An Introduction to Slice-Based Cohesion and Coupling Metrics

Pam Green
Peter C.R. Lane
Austen Rainer
Sven-Bodo Scholz

June 2009

Technical Report No.488
School of Computer Science
University of Hertfordshire

Contents

Abstract

This report provides an overview of slice-based software metrics. It brings together information about the development of the metrics from Weiser's original idea that program slices may be used in the measurement of program complexity, with alternative slice-based measures proposed by other researchers. In particular, it details two aspects of slice-based metric calculation not covered elsewhere in the literature: output variables and worked examples of the calculations.

First, output variables are explained, their use explored and standard reference terms and usage proposed. Calculating slice-based metrics requires a clear understanding of 'output variables' because they form the basis for extracting the program slices on which the calculations depend. This report includes a survey of the variation in the definition of output variables used by different research groups and suggests standard terms of reference for these variables. Our study identifies four elements which are combined in the definition of output variables. These are the function return value, modified global variables, modified reference parameters and variables printed or otherwise output by the module.

Second, slice-based metric calculations are explained with the aid of worked examples, to assist newcomers to the field. Step-by-step calculations of slice-based cohesion and coupling metrics based on the vertices output by the static analysis tool CodeSurfer® are presented and compared with line-based calculations.

1 Introduction

There are various forms of static source code analysis which are aimed at improving software quality. Examples include finding security vulnerabilities [4], bug detection [8, 18, 40] and locating regions of code which may benefit from reengineering [7, 23]. The reduction of code complexity is another area which can be aided by static source code analysis. Yourdon and Constantine [39] suggest that increasing modularisation of program code decreases its complexity. Static analysis techniques can provide code quality metrics to aid code modularisation both by highlighting complex sections of code and by quantifying improvements.

Two measures of a program's modularity are cohesion and coupling. The cohesion of a module is a measure of the degree to which the functions that the module performs are related. Ideally a module will contain a single function or a very closely related set of functions. Cohesion is described by Page-Jones as "the measure of the strength of functional relatedness of the elements within a module." [32, Chap.6].

Stevens et al. [35] proposed a set of seven qualitative levels of cohesion based on the associations between processing elements in a module; an element being defined as a set of statements in the module performing one task. These levels, in descending order of cohesion strength, are functional, sequential, communicational, procedural, temporal, logical and coincidental. Stevens et al. consider the cohesion levels to be non-linear in scale; functional cohesion is deemed to be much stronger than the other levels, with coincidental and logical cohesion very much weaker than the intermediate levels.

Coupling is a measure of the degree of interdependence between the modules in a system. Page-Jones [32] suggests that faults and changes are less likely to be propagated through the program by the smaller number of connections which are associated with low coupling between modules. The recommended best practice is for a system to have high cohesion and low coupling, reducing errors [34] and simplifying maintenance and understanding [16]. Ideally, highly cohesive code will not be strongly coupled and vice versa [39].

Examples of traditional measures of coupling are Henry and Kafura's complexity metric [15] or the Shepperd complexity measure [19], both of which are based on the information flow between modules. Information flow is measured in terms of the variables defined in one module and used in another.

In 1979 Mark Weiser [37] introduced the concept of program slicing, taking the way that programmers debug their code as his inspiration. His idea was to remove any lines of code not affecting the values computed at the point under consideration, as long as the removal did not impede the running of the program. Slicing algorithms act on intermediate representations of the program. Automatic slicing tools parse the program code to create one or more of these representations, such as control flow graphs (CFG), abstract syntax trees (AST) or system dependence graph (SDG). Weiser sliced using data flow analysis; modern tools generally trace the transitive control and data dependences by traversal of the SDG.

A slice is taken in respect of a variable or set of variables at a point in the program. The simplest form of slicing is static slicing; this includes all statements affecting or affected by the variable(s) at the point of interest, for all possible inputs to the program. There are other types of slicing which aim to reduce the size of slices; for example, dynamic slicing considers a particular set of input values, which may reduce the slice, as branches of the program not executed when using these inputs will be excluded; amorphous slicing reduces the slice by the use of transformations to the program. For each of these, in simple terms, backward slicing will result in the set of program statements, or points, which affect the calculation of the value of the variable at the point of interest. Forward slicing finds the set of program statements, or points, which will be affected by computation at the point of interest. See, for example, [2, 6, 9, 13, 36] for more detailed information on slicing.

Program slices offer a basis for measuring module cohesion. The slices used in calculating slice-based cohesion metrics are derived from the individual tasks performed by a module. Metrics based on these slices provide a means of quantifying the interaction between the computational strands of the module, and thus its cohesion. Meyers and Binkley report that their longitudinal study of slice-based cohesion metrics indicates that the metrics quantify overall code quality [29, p.2].

A slice-based metric also offers a finer-grained measure for coupling than traditional measures such as Henry and Kafura's. The former measures how much of the code in an interprocedural slice is external to the module, while the latter considers only whether a connection exists between modules [14].

The tasks of a module are represented by the variables changed or output in its execution. The slices used in the calculation of slice-based metrics are taken in respect of these variables at the last point of change, or at the point of output. The variables on which the slices are based are known as "output variables". However, there appears to be no standard definition or use of these variables in slice-based metrics and their description by different research groups varies. Section 2 explores this lack of consistency and offers a suggestion for standardising references to, and use of, output variables.

The rest of the document is in two main parts: Section 3 focuses on slice-based cohesion metrics and Section 4 on slice-based coupling metrics. Each of these sections has two main purposes. The first is to provide a background to the development of the metrics, and introduce the methods for calculating slice-based metrics proposed by different research groups. The second is to give a clear explanation of the metric formulae and clarify their calculation with worked examples based on a sample program.

This report covers two aspects of slice-based metric calculation not found elsewhere in the literature. First, output variables are explained, their use explored and standard reference terms and usage proposed. Second, slice-based metric calculations are explained with the aid of worked examples to assist newcomers to the field.

Note that the words 'module' and 'procedure' are used interchangably in this document.

2 Output variables

Research into slice-based metrics requires a clear understanding of ‘output variables’, sometimes referred to as ‘principal variables’. These variables form the basis for extracting the program slices necessary for the calculation of slice-based metrics. Unfortunately, existing definitions lack consistency; the results of an investigation aiming to establish a uniform point of reference is presented in this section. Section 2.1 shows the literature selected for analysis. The elements which are combined to describe output variables are discussed in Section 2.2, followed by an analysis of their use in Section 2.3. Section 2.4 summarises.

2.1 Sources

To survey the definitions of ‘output variables’ in the literature on program slicing, the following strategy was used to select publications on the subject. A search was made of the IEEE, ACM and Google Scholar bibliographic databases using the terms “output variables”, “principal variables”, “program slicing” and “slice-based metrics” separately. The papers resulting from this search were used as the basis for further search. This included papers referenced by and referencing the selected papers and those written by authors of the selected papers. The results were filtered to remove those papers not describing output variables and those where the author had duplicated a description found in one of their previous papers. This strategy resulted in the selection of the eleven publications shown in Table 1.

Table 1: Paper index numbers (P-n) used to refer to the reviewed papers in sections 2.2 and 2.3

P-n	Author	Date	Title	Ref.	Page
P-1	Ott and Thuss	1989	The relationship between slices and module cohesion.	[30]	201
P-2	Lakhotia	1993	Rule-based approach to computing module cohesion.	[22]	37
P-3	Bieman and Ott	1994	Measuring functional cohesion	[1]	645
P-4	Karstu	1994	An examination of the behaviour of slice based cohesion measures.	[20]	7-8
P-5	Harman et al.	1995	Cohesion metrics.	[11]	3
P-6	Harman	1997	Cleaving together - program cohesion with slices	[10]	36
P-7	Meyers and Binkley	2004	A longitudinal and comparative study of slice-based metrics	[27]	2
P-8	Meyers and Binkley	2004	Slice-based cohesion metrics and software intervention.	[29]	257
P-9	Pan et al.	2005	Bug classification using slicing metrics.	[33]	32
P-10	Meyers and Binkley	2007	An empirical study of slice-based cohesionand coupling metrics.	[28]	5
P-11	Binkley et al.	2007	Empirical study of optimization techniques for massive slicing.	[3]	3

2.2 The elements

Consideration of the eleven selected publications and of similar work by these authors leads us to identify four ‘elements’ underlying the definitions found in these papers. These are: the return value of a function; global variables modified by the function; parameters passed by reference to, and modified by, a function; and any variables printed, written to file, or otherwise output to the external environment.

Table 2 provides short code examples for each of these different elements. In each example the output variable is shown in bold text and an intraprocedural backward slice on the output variable will include all of the statements in the function. Except in the print example, the variable ‘total’, external to the function, is the one modified. In the print example, the local variable ‘sum’ is output to the external environment by printing.

F. Function return value.

In the example (Table 2, column 1), the value of the variable ‘sum’ is returned to the calling function; this is the function return value.

G. Modified global variable.

Variables with scope outside a function which are modified by that function are considered to be modified global variables. The example given (Table 2, column 2), shows the variable ‘total’, defined outside of the function, being changed in the function add.

P. Printed output.

The third column in Table 2 provides an example of a printed variable. In this case the value of the variable ‘sum’ is printed to the standard output. This category of output variable also contains, for example, variables written to file, variables controlling graphical output and variables output to control an external device.

R. Reference parameter.

Reference parameters are those whose address is passed to a function. The contents of this address may then be modified by the function, as shown in columns 4 and 5 of Table 2. This is useful where a function alters more than one value, or to save passing large data objects, such as arrays, as parameters. In column 4 the pointer ‘ptr’ is set to the value of the address of the variable ‘total’. The pointer is passed to the function, where the value ‘*ptr’ means the contents of the address held in ‘ptr’, thus ‘total’ is modified. In the last column in Table 2, the address of ‘total’ is passed directly; this facility is available in C++ but not in C.

2.3 Analysis

There is a seven-year gap between earlier (1-6) and later (7-10) papers. This is partly because relevant publications in the intervening years were by authors whose papers are already included and whose definitions are similar in meaning (for example, Harman et al. [12], which echoes Harman [10]) and partly due to renewed interest in the study of slice-based metrics enabled by the use of CodeSurfer®, a mature commercial tool for the analysis of C or C++ programs, based on their dependence graphs [5].

Earlier papers (1-6) do not mention function returns specifically, but some allude to it obliquely, for example, Harman, paper 4, “this value must be visible outside the function” [10] or Ott and Thuss, paper 1, “If slices are obtained for all of the module’s outputs, their union will completely encompass the module (unless it contains unused or “dead” code).” [30] Later papers (7-10) all use clear terms such as “a function’s return value”.

Papers 1, 3, 4, 5 and 6 include ‘printed’ output variables in their descriptions, for example, Karstu, paper 4, “also included in outputs would be any direct output from the module to files or devices” [20] or Bieman and Ott, paper 3, “any single value explicitly output to a file (or user output)” [1]. Later papers,

Table 2: Sample C code to sum the n elements of an array, showing examples of each of the ‘elements’

F. Function return	G. Modified globals.	P. Printed or similar.	R. Reference parameter 1.	R. Reference parameter 2.
. . .	int total = 0;	. . .	. . .	. . .
int main() . . . . . . total = add (sizes, n); . . .	int main() . . . . . . add (sizes, n); . . .	int main() . . . . . . add (sizes, n); . . .	int main() . . . int *ptr = & total; add (sizes, n, ptr); . . .	int main() . . . . . . add (sizes, n, total); . . .

int add (int sizes[], int n)
{
int sum = 0;
int j;
for (j = 0; j < n; j++)
{
sum = sum + sizes[j];
}
return **sum**;
}
	int add (int sizes[], int n)
	{
	int j;
	for (j = 0; j < n; j++)
	{
	total = total + sizes[j];
	}
	}
		int add (int sizes[], int n)
		{
		int sum = 0;
		int j;
		for (j = 0; j < n; j++)
		{
		sum = sum + sizes[j];
		}
		printf (”total=%d”, **sum**);
		}
			int add (int sizes[],
			int n, int *ptr)
			{
			int sum = 0;
			int j;
			for (j = 0; j < n; j++)
			{
			sum = sum + sizes[j];
			}
			***ptr** = *ptr + sum;
			}
				int add (int sizes[],
				int n, int &total)
				{
				int j;
				for (j = 0; j < n; j++)
				{
				total = total + sizes[j];
				}
				}

7 and 8, include the printed variables as examples, but paper 10 explains that these are not included in the calculation of the metrics.

Global variables feature in all of the papers except for paper 7, however the other Meyers and Binkley papers, 8 and 10, do include the words “those globals modified by the function” [28, 29]. The descriptions for this are generally straightforward, for example, Bieman and Ott, paper 3, “an assignment to a global variable” [1].

Direct mention is made of reference parameters as output variables in papers 1, 2, 5 and 6, while Karstu, paper 4, says “all data tokens that will be accessible from the outside of the module that have been modified by the module: e.g. variable parameters, global variables” [20]. Of the later papers, only 9 and 11 mention reference parameters.

Table 3 summarises the elements found in each of the papers. Each row represents one paper; if the definition given in a paper includes an element, the corresponding column in the table is marked. The last column, combination, gives a new number to each different combination of the elements found. The four elements are used in seven different combinations, three of which are repeated.

Papers 1, 2, 3 and 5 are theoretical, paper 6 is a magazine article and 4, 7, 8, 9 and 10 are based on empirical studies; of these, all except for the work reported in paper 4 use CodeSurfer as the analysis tool.

There are many functions provided by CodeSurfer to aid the analysis process, one of which allows the extraction of both the function return values and the global variables modified by the function. The comparative ease of isolating these two output variable types offers a simple explanation for the choice of output variables by both the research groups using this tool. The set of modified global variables generated by the analysis process will include “generated formal parameters representing a set of globals modified in a function immediately or transitively” [5]. Therefore parameters passed by reference will, if modified, be in the set returned by the CodeSurfer function used to extract function returns and modified globals.

Ott and Thuss [30], Karstu [20] and Harman [10] offer a definition which includes all of the identified elements. Of these, only Karstu performed a practical study, made possible by the slicing tools

Table 3: Combinations of elements described in the papers.

Paper Reference P-n	Biblio-graphy ref.	Author	Date	Function return F	Global variables G	Variables printed P	Reference parameters R	Comb-ination No.
P-1	[30]	Ott and Thuss	1989	F	G	P	R	1
P-2	[22]	Lakhotia	1993		G		R	2
P-3	[1]	Bieman and Ott	1994	F	G	P		3
P-4	[20]	Karstu	1994	F	G	P	R	1
P-5	[11]	Harman et al.	1995		G	P	R	4
P-6	[10]	Harman	1997	F	G	P	R	1
P-7	[27]	Meyers and Binkley	2004	F		P		5*
P-8	[29]	Meyers and Binkley	2004	F	G	P		3*
P-9	[33]	Pan et al.	2005	F	G		R	6
P-10	[28]	Meyers and Binkley	2007	F	G			7
P-11	[3]	Binkley et al.	2007		G	P	R	4

∗Paper 10 offers clarification of the work in papers 7 and 8, explaining that, in spite of the examples given, the studies did not encompass those variables printed or similarly output.

he developed, however these have certain limitations [28], for example “pointers are treated as regular variables” [20, p.17].

It could be argued that the inclusive definition of output variables would provide a sound basis for further work involving slice-based metrics, as this represents the intention of Ott and Thuss [30] to analyse all of the module except for ‘dead’ code. By using CodeSurfer, it is possible to include three of the four elements with comparative ease; however the printed, or similarly output, variables are more difficult to capture.

Most examples in the slicing literature have been with programs written in C/C++. CodeSurfer also works with C/C++ programs. Of interest is whether this analysis would potentially transfer to other programming languages or paradigms. One difficulty here is that terminology is not fixed. For example, Java only permits ‘pass by value’, but as every reference to an object is through a pointer, the effect is similar to C++’s use of a reference parameter; objects passed as references to Java methods may be directly modified. This situation holds for other languages permitting side-effects, such as Lisp, Python, Ruby and Scheme; these languages support all four of the elements. Languages from other paradigms, such as functional programming, do not offer all the elements; for example Haskell provides function returns [F] and printed variables [P] only. However, these elements appear to provide a useful structure for comparing the definitions of output variables between projects and across languages.

2.4 Summary

This study shows that the definition of an output variable could be broken down into a set of four elements: function return value, modified global variable, ‘printed’ variable, and modified reference parameter. The authors of eleven previous papers have defined output variables in different ways. This analysis shows that each paper’s definition can be captured in terms of four elements. The different combinations of the elements presented in the papers as definitions for output variables are tabulated.

There does not appear to be a standard to determine which elements, if not all, should be used in a definition of ‘output variables.’ Certainly, the available tools for obtaining these variables affect which definition is convenient at any time. However, what is clear is that for source-code analysis to offer meaningful lessons between and across projects, it is important to define clearly which of the elements are being used as the output variables.

The effect of the choice of output variables is partly dictated by the nature of the program being analysed. For example, the slice-based metrics for a program without global variables will be unaffected if the ‘global’ element is excluded from the set of variables used for slicing. However, it is difficult to imagine a program which does not have some output stream; therefore the decision about exclusion of the variables ‘printed’, or otherwise written to the external environment, is more important.

3 Slice-based cohesion metrics

There are two parts to this section. Section 3.1 traces the development of slice-based cohesion metrics and details the formulae used in their calculation. Worked examples, illustrating the calculation of slice-based cohesion metrics for a small program, follow in Section 3.2.

3.1 Background and Development

This section traces the history of the development of slice-based cohesion metrics. As part of his work on slicing as a debugging tool, Weiser [38] suggested that metrics based on program slices might bear a relationship to program complexity; his proposals are given in Section 3.1.1. Weiser's ideas were investigated and developed by Longworth in his Master's thesis [24], for which his advisor was Linda Ott; details of this work are in Section 3.1.2. Subsequently, Ott worked with Thuss [31] to refine the set of metrics; their studies and revised set of metrics are described in Section 3.1.3. Overlap, one of the measures, was redefined by each of these sets of researchers and this is discussed in Section 3.1.4. A description of two alternative slice-based cohesion measures, adhesion, proposed by Bieman and Ott, and statement level cohesion, by Krinke are given in Sections 3.1.5 and 3.1.6 respectively. Meyers and Binkley's empirical studies of slice-based cohesion metrics [26, 27, 29, 28] are outlined in Section 3.1.7. The metrics used in these studies and most used in research since the mid-1990's are those formulated by Ott and Thuss; these are discussed in Section 3.1.8 and the formulae presented in Table 6.

3.1.1 Weiser

In 1981 Mark Weiser introduced the concept of program slicing, taking his inspiration from the way in which programmers debug their code. His idea was to remove any lines of code not affecting the values computed at the point under consideration, while leaving an executable program.

While considering uses for slicing other than an aid to debugging, Weiser proposed five "slicing-based program metrics" of complexity, based on the relationships between module size, slice sizes and the intersection of these slices, suggesting that they "may be particularly meaningful, compared to such measures as McCabe's or Halstead's" [38, p.447]. The measures he described were coverage, overlap, clustering, parallelism and tightness. Weiser's descriptions of his proposed slicing-based metrics:

"Coverage	compares the length of slices to the length of the entire program. Coverage might be expressed as the ratio of mean slice length to program length. A low coverage value, indicating a long program with many short slices, may indicate a program which has several distinct conceptual purposes.
Overlap	is a measure of how many statements in a slice are found only in that slice. This could be computed as the mean of the ratios of non-unique to unique statements in each slice. A high overlap might indicate very interdependent code.
Clustering	reveals the degree to which slices are reflected in the original code layout. It could be expressed as the mean of the ratio of statements formerly adjacent to total statements in each slice. A low cluster value indicates slices intertwined like spaghetti, while a high cluster value indicates slices physically reflected in the code by statement grouping.
Parallelism	is the number of slices which have few statements in common. Parallelism could be computed as the number of slices which have a pair wise overlap less than a certain threshold. A high degree of parallelism would suggest that assigning a processor to execute each slice in parallel could give a significant program speed-up.

Tightness	measures the number of statements which are in every slice, expressed as a ratio over the total program length. The presence of relatively high tightness might indicate that all the slices in a subroutine really belonged together because they all shared certain activities." [38, p.447]

A summary of the formulae for Weiser's three metrics related to cohesion: coverage, overlap and tightness is shown in Table 4.

Table 4: Weiser's [38] coverage, overlap and tightness formulae

Metric	**Ratio**	
Coverage	Mean of	$\frac{slice\ size}{module\ size}$
Overlap	Mean of	$\frac{unique\ statements\ in\ slice}{non-unique\ statements\ in\ slice}$
Tightness		$\frac{slice\ intersection\ size}{module\ size}$

3.1.2 Longworth

In his master's thesis, Longworth [24] investigated Weiser's proposed complexity metrics and whether they provided a useful measure of cohesion within a program. The two ideas of cohesion and complexity are linked: cohesion levels are based on the relationship between processes in a module and this relationship is an element of the complexity of the system.

He pointed out that parallelism has a different nature to the other metrics, in that it is not related to either complexity or to modular cohesion. He also discusses the weakness of parallelism as a guide to the usefulness of parallel processing because the type of statements which can be decomposed are not reflected by the measure. Clustering is disregarded as a measure of cohesion as "an aspect of spatial arrangement only" [24, p.33] rather than a measure of functional relatedness.

Longworth was therefore interested in discovering whether coverage, overlap[1] and tightness give an indication of cohesion. He initially studied a set of thirteen programs, selected from literature about data flow patterns. These programs were necessarily small (14-86 lines) to allow slicing 'by hand'. Weiser's five slice-based metrics were calculated for each of the programs. These metrics and the cohesion levels that the programs illustrated were compared, leading Longworth to conclude that coverage provides a good indicator for either high or low levels of cohesion, while overlap and tightness can be considered to confirm the level suggested by coverage. To explore these relationships further, Longworth conducted two more studies. One used different versions of two other programs, varying the level of code dependency in each. In the other, complex programs were decomposed into stages and analysed; these studies supported his initial findings.

3.1.3 Ott and Thuss

The metrics proposed by Weiser and investigated by Longworth were further interpreted and extended by Ott and Thuss [31]. Essentially, Weiser sliced on every variable where it occurred in a procedure and Longworth sliced on every variable once, at the end point of the procedure. Ott and Thuss base

[1]note that in his study, Longworth redefined overlap as the mean of the ratios of non-unique statements to slice length

their metrics only on slices taken for each of the output variables, which are variables selected to capture the behaviours of the module, details of which may be found in Section 2. They introduce an amended method for slicing, the metric slice, to remove the inconsistencies found by Longworth between the metrics and cohesion levels. A metric slice is a backward slice computed on a variable from the end point of the procedure, combined with forward slices computed from definitions of the variable in the backward slice.

Ott and Thuss added minimum and maximum coverage to their suite of metrics, the reason for this is not stated in their paper, but may be because coverage was shown by Longworth to be most related to the cohesion levels that he used. MinCoverage is the ratio of the size of the smallest slice, and MaxCoverage the largest slice, to the size of the module. They also chose to exclude clustering and parallelism from their suite of metrics. Their measurement of overlap differs from both Weiser's and Longworth's, in that the ratio used is that of the slice intersection to slice size. Ott and Thuss's metrics are given in Table 6 on page 15.

3.1.4 The evolution of overlap

The evolution of the definition of overlap, shown in Table 5, brings differences to the outcome of the calculation of this metric. Weiser's proposal may result in a value greater than one: for example, consider a module with two slices, one of four statements and one of six. If three of these statements are shared, then overlap will be the mean of $\frac{3}{1}$ and $\frac{3}{3}$ = 2. Weiser's overlap is also undefined if there are no unique statements in the slice. The other metrics fall in the range 0-1, as does overlap when calculated by the amended methods suggested by Longworth or by Ott and Thuss. The Ott and Thuss variation will generally result in a smaller value than Longworth's method, as the number of statements shared by all slices will usually be smaller than the number of non-unique statements, unless there are fewer than three output variables. It can be noted that Bieman and Ott's strong functional adhesion (see Section 3.1.5) has a similar nature to the Ott and Thuss overlap because both are computed using elements (tokens or vertices) in all of the slices; likewise weak functional adhesion is similar to Longworth's overlap, as each uses elements which are shared by at least two slices.

Table 5: The evolution of overlap: in each case, overlap is the mean of the ratios described

Author	Numerator	Denominator
Weiser	non-unique statements	unique statements
Longworth	non-unique statements	slice length
Ott and Thuss	slice intersection	slice length

3.1.5 Bieman and Ott

Also of interest is a slice-based method for measuring functional cohesion suggested by Bieman and Ott [1]. The basis for these calculations is the presence in the slices of data tokens rather than lines of code or vertices. Data tokens are the definitions of and references to variables and constants. Not only are the data tokens which are common to all the slices in the module (superglue tokens) considered, but also any tokens which are shared by more than one slice (glue tokens). The strong functional cohesion metric is defined as the number of superglue tokens as a proportion of the total tokens for the module. Similarly, the weak functional cohesion measure is the number of glue tokens (including superglue tokens) as a proportion of the total tokens. The adhesiveness of a glue token can be measured by the number of slices in which it appears as a proportion of the total number of slices, while non-glue tokens, that is a token appearing in only one slice, have zero adhesiveness. The adhesiveness of a module is then the mean adhesiveness for all the data tokens in the module. A worked example is shown in Table 13, page 22. This method appears not to have been investigated in recent studies, though Harman et al. [11] draw parallels between this method and their investigation into the use of leaves, nodes or tokens as the basis for slice-based metrics calculations.

3.1.6 Krinke

Krinke [21] suggests two methods for calculating cohesion measures at statement level. One is a slice-based metric, in which the cohesion for a statement is the sum of the sizes of the slices in which the statement appears as a proportion of the sum of the sizes of all slices in the module. The other is a chop-based measure; a chop consists of the transitive dependences, or the influence of one set of points on another, between a source and a target point in a program. Chops are taken between each of the input variables (procedure arguments and global variables) and each output variable for the procedure. The cohesion for a statement is the number of chops in which the statement appears as a proportion of the total number of chops for the module, which is the product of the number of input variables and the number of output variables.

3.1.7 Meyers and Binkley

The introduction of analysis tools such as CodeSurfer [5], which enable automated slicing, has allowed more wide-ranging research into slice-based cohesion metrics. Meyers and Binkley undertook the broadest study of slice-based cohesion metrics [28, 29], which researches baseline metric values, their relationship with each other, with size-based metrics, and with code quality. The metrics they calculated were those detailed by Ott and Thuss, with the addition of Weiser's parallelism. However, the metric slice used by Ott and Thuss is replaced by the two-pass algorithm of Horwitz et al. [17] in these studies, because metric slices are "likely to be larger than a traditional slice with a corresponding drop in usefulness" [29, p.9].

Meyers and Binkley analysed 63 programs covering a wide range of sources, applications and styles, to establish baseline metric values. They performed longitudinal analysis of two programs showing that the metrics correspond to code quality. When comparing the metric values with each other, they found a very strong correlation between tightness and minCoverage and strong correlation between minCoverage and overlap; tightness and overlap; maxCoverage and coverage. These correlations suggest that it may not be necessary to calculate all of the metrics to characterise the code.The metrics were also compared with the number of lines of code in the modules, which "show that slice-based metrics provide a unique view of a program and are not a proxy for size-based metrics" [28, p.3]

3.1.8 Formulae

The formulae for slice-based cohesion metrics shown in Table 6, page 15, are those proposed by Ott and Thuss and subsequently used in research by, for example, Meyers and Binkley [29], Harman et al. [11] and Majumdar et al. [25]. While Weiser based his ideas on the number of lines, Ott and Thuss' calculations are based on variable referent executable statements (VRES). Since the advent of slicing tools, such as CodeSurfer, based on program dependency graphs, it has become general practice to calculate these values in terms of the vertices of the graph, though a study by Harman et al [11] compared the metrics using these two bases with those based on tokens or distinct variables or on the leaves or the nodes in the AST (abstract syntax tree). Their analysis of two short (5 line) sample functions providing an example of support for the idea of using leaves, nodes or tokens. They also suggest that the conservative nature of slicing algorithms will produce over-large values for the cohesion measures.

Although the cohesion metrics of Ott and Thuss are the most commonly used in research projects, it is interesting to note their variation and continuing development. For example, the bug classification study by Pan et al. [33] introduced a different set of slice-based metrics broadly based on the ideas behind "conventional" metrics. These aim to provide similar information to conventional cohesion and coupling metrics at a finer-grained level, being based on the vertices in a slice rather than lines of code. For example, the "conventional" metric CountInput, which is the number of inputs to a function, is echoed by their "globalInput" measure which counts all inputs to a function, both local and global.

Table 6: Slice-based cohesion metrics

Terms

Name	Notation	Meaning
Module	M	Unit of code under analysis, for example, a procedure.
Module variables	V_M	All the variables in the module.
Output variables	V_O	The output variables in the module. $V_O \subseteq V_M$.
Output variable (i)	v_i	One of the output variables. $v_i \in V_O$.
Slice on v_i	SL_i	A slice on output variable v_i.
Slice intersection	SL_{int}	Intersection of slices for each of the output variables. $\bigcap_i SL_i$.

Formulae

Metric	Formula	Description	Meaning
Coverage(M)	$\frac{\sum_{i=1}^{\lvert V_O\rvert} \frac{\lvert SL_i\rvert}{\lvert M\rvert}}{\lvert V_o\rvert}$	$\frac{sum\ of\ slice\ sizes}{\#(output\ variables) \times module\ size}$	Mean of the ratios of each slice size to the size of the module.
MinCoverage(M)	$\frac{min_i \lvert SL_i\rvert}{\lvert M\rvert}$	$\frac{minimum\ slice\ size}{module\ size}$	The ratio of the size of the smallest slice to the module size.
MaxCoverage(M)	$\frac{max_i \lvert SL_i\rvert}{\lvert M\rvert}$	$\frac{maximum\ slice\ size}{module\ size}$	The ratio of the size of the largest slice to the module size.
Overlap(M)	$\frac{\sum_{i=1}^{\lvert V_O\rvert} \frac{\lvert SL_{int}\rvert}{\lvert SL_i\rvert}}{\lvert V_o\rvert}$	$\frac{\sum_i \frac{intersection\ size}{slice\ size_i}}{\#(output\ variables)}$	Mean of the ratios of intersection size to the size of each slice in the module.
Tightness(M)	$\frac{\lvert SL_{int}\rvert}{\lvert M\rvert}$	$\frac{size\ of\ slice\ intersection}{module\ size}$	The ratio of the slice intersection size to that of the module.

3.2 Worked examples

This section comprises an example program with worked examples based on the code. The program on which the calculations are based is discussed in Section 3.2.1. The use of both lines of code and vertices in calculating Ott and Thuss's metrics are shown here; line-based calculations in Section 3.2.2 and vertex-based in Section 3.2.5. Weiser's parallelism is discussed in Section 3.2.3 and Bieman and Ott's adhesion calculations given in Section 3.2.4. A discussion follows in Section 3.3.

3.2.1 Example program

The example program, shown in Table 12, page 21, calculates the slice-based metrics defined by Ott and Thuss for a set of slice sizes, module size, slice intersection size and number of output variables. There are three procedures, 'main', 'calcMetrics' and 'overlap'. The five output variables are 'tightness', 'coverage', 'min-coverage', 'max-coverage' and 'overlap'. Each variable is allocated a single letter key, shown at the bottom of the table. These letters are used to mark the lines which appear in the backward slice on the variable at the last point at which it occurs in the program. Line numbers are given as plain numbers for the procedure 'calcMetrics', with the prefix 'm' for 'main' and the prefix 'o' for 'calcOverlap'. To simplify the line-based calculations, lines containing only a brace have been ignored.

3.2.2 Line-based calculations

This section provides worked calculations of slice-based cohesion metrics for the example program using lines of code as a measure. Table 7 on page 16, displays the information extracted from the program which is required for the calculations. For each module, the information shown is: size, the output variables and their location, the lines in the slice taken on the variable at that location, a count of these lines, and the intersection of lines in all slices in the module. Table 8 on page 17 details the calculations.

Table 7: Analysis of line-based intraprocedural slices

Module		**Slice taken**				**Intersection**	
name	size	on output variable	at line	lines in the slice	total	lines	total
calcMetrics	20	tightness	16	{16, 11, 5, 1}	4		
		coverage	17	{17, 12, 10, 7, 6, 5, 4, 1}	8		
		min-coverage	18	{18, 14, 8, 7, 6, 2, 1}	7		
		max-coverage	19	{19, 15, 9, 7, 6, 3,1}	7		
		overlap	20	{20, 13, 5, 1}	4	{1}	1
calcOverlap	6	return value	o6	†{o6, o5, o4, o3, o2, o1}	6	{o1-o6}	6
main	6	none			0		0
				†*on* refers to line *n* in calcOverlap			

Table 8: Line-based calculation of slice-based cohesion metrics

calcMetrics						
Tightness	=	$\frac{size\ of\ slice\ intersection}{module\ size}$	=	$\frac{1}{20}$	=	0.05
Coverage	=	$\frac{sum\ of\ slice\ sizes}{\#(output\ variables) \times module\ size}$	=	$\frac{30}{5 \times 20}$	=	0.30
Min-coverage	=	$\frac{minimum\ slice\ size}{module\ size}$	=	$\frac{4}{20}$	=	0.20
Max-coverage	=	$\frac{maximum\ slice\ size}{module\ size}$	=	$\frac{8}{20}$	=	0.40
Overlap	=	$\frac{\sum_i \frac{intersection\ size}{slice\ size_i}}{\#(output\ variables)}$	=	$\frac{\frac{1}{4}+\frac{1}{8}+\frac{1}{7}+\frac{1}{7}+\frac{1}{4}}{5}$		
			=	$\frac{51}{56} \times \frac{1}{5}$	=	0.18

calcOverlap						
Tightness	=	$\frac{size\ of\ slice\ intersection}{module\ size}$	=	$\frac{6}{6}$	=	1
Coverage	=	$\frac{sum\ of\ slice\ sizes}{\#(output\ variables) \times module\ size}$	=	$\frac{6}{1 \times 6}$	=	1
Min-coverage	=	$\frac{minimum\ slice\ size}{module\ size}$	=	$\frac{6}{6}$	=	1
Max-coverage	=	$\frac{maximum\ slice\ size}{module\ size}$	=	$\frac{6}{6}$	=	1
Overlap	=	$\frac{\sum_i \frac{intersection\ size}{slice\ size_i}}{\#(output\ variables)}$	=	$\frac{\frac{6}{6}}{1}$	=	1

3.2.3 Parallelism, a line-based example

Parallelism is not stated as one metric value, but a series giving the number of slices having no more than a given number of lines (or statements) in common with other slices; stated as parallelism (M, τ), where M is the module and τ is the maximum number of lines (or statements) shared with other slices. Table 9 shows the lines shared by more than one slice in the module calcMetrics in the sample program. The slices on "tightness" and "overlap" have two lines in common with any other slice (lines 1 and 5 in each case). The slice based on "coverage" and "minCoverage" have 1 line in common with all the other slices, 1 line shared with 3 other slices and 2 lines shared with 2 others. The slice based on "maxCoverage" has 1 line shared with all slices and 2 shared with 2 other slices. Therefore parallelism (calcMetrics,1) = 0; parallelism (calcMetrics,2) = 2; parallelism (calcMetrics,3) = 3; parallelism (calcMetrics,4) = 5.

Table 9: Lines of code common to more than one slice in the module calcMetrics, for parallelism calculation

	Line number				
Slice based on	1	5	6	7	Total
coverage	x	x	x	x	4
min-coverage	x	x	x	x	4
overlap	x	x			2
tightness	x	x			2
max-coverage	x		x	x	3

3.2.4 Bieman and Ott's adhesion measure

As described in Section 3.1.5, Bieman and Ott's cohesion measure, adhesion, is based on data tokens. Data tokens are definitions of, and references to variables and constants. Any token appearing in only one slice has zero adhesiveness, unless there is only one slice in the module, when it has a value of one. Other tokens' adhesiveness is calculated as the ratio of the number of slices in which the token appears to the total number of slices. Calculation of the adhesion of the sample program modules is shown in Table 13.

3.2.5 Vertex-based calculations

In this section, slice-based cohesion metrics based on vertices are calculated for the example program. The information is obtained by use of a Scheme script with CodeSurfer. More information on the vertices can be found in Table 17 on page 30. The layout is similar to that in Section 3.2.2 with Table 10 displaying the information required for calculations extracted from the program. For each module: size; the output variables and their location; the vertices in the slice taken on the variable at that location; a count of these vertices; and the intersection of vertices in all slices in the module. Table 11, page 19, details the calculations and offers a comparison with line-based figures.

Table 10: Analysis of vertex-based intraprocedural slices

Module		**Slice taken**				**Intersection**	
name	size	on output variable	at vertex	vertices in the slice	total	vertices	total
calcMetrics	55	tightness	51	{51,31,24,15,8,7,6,5,1}	9		
		coverage	49	{60,59,49,33,25,23,22,21,18,16,14,10,9,6,5,4,3,1}	18		
		min-coverage	47	{62,60,57,47,35,29,23,22,19,18,12,10,9,6,5,4,3,1}	18		
		max-coverage	45	{61,60,58,45,37,30,23,22,20,18,13,10,9,6,5,4,3,1}	18		
		overlap	43	{56,55,54,53,43,39,28,26,17,10,9,8,7,4,3,1}	16	{1}	1
calcOverlap	19	return value	18	{20,19,18,15,14,13,12,11,10,8,7,6,5,4,3,1}	16	{as slice}	16
main	19	none			0		0

Table 11: Vertex-based calculation of slice-based cohesion metrics, also showing the result of excluding "system" vertices[a] and comparable line-based results.

calcMetrics					Vertex based	Exc.sys[a]	Line based
Tightness	=	$\frac{size\ of\ slice\ intersection}{module\ size}$	=	$\frac{1}{55}$	= 0.018	0.019	.05
Coverage	=	$\frac{sum\ of\ slice\ sizes}{\#(output\ variables)\times module\ size}$	=	$\frac{79}{5\times55}$	= 0.287	0.304	0.3
Min-coverage	=	$\frac{minimum\ slice\ size}{module\ size}$	=	$\frac{9}{55}$	= 0.164	0.173	0.2
Max-coverage	=	$\frac{maximum\ slice\ size}{module\ size}$	=	$\frac{18}{55}$	= 0.327	0.346	0.4
Overlap	=	$\frac{\Sigma_i \frac{intersection\ size}{slice\ size_i}}{\#(output\ variables)}$	=	$\frac{\frac{1}{9}+\frac{1}{18}+\frac{1}{18}+\frac{1}{18}+\frac{1}{16}}{5}$			
			=	$\frac{49}{144}\times\frac{1}{5}$	= 0.068	0.068	0.18
calcOverlap							
Tightness	=	$\frac{size\ of\ slice\ intersection}{module\ size}$	=	$\frac{16}{19}$	= 0.842	1	1
Coverage	=	$\frac{sum\ of\ slice\ sizes}{\#(output\ variables)\times module\ size}$	=	$\frac{16}{1\times19}$	= 0.842	1	1
Min-coverage	=	$\frac{minimum\ slice\ size}{module\ size}$	=	$\frac{16}{19}$	= 0.842	1	1
Max-coverage	=	$\frac{maximum\ slice\ size}{module\ size}$	=	$\frac{16}{19}$	= 0.842	1	1
Overlap	=	$\frac{\Sigma_i \frac{intersection\ size}{slice\ size_i}}{\#(output\ variables)}$	=	$\frac{\frac{16}{16}}{1}$	= 1	1	1

[a]Where "system" vertices, such as body, return and exit, are excluded, or the union of the vertices in the slices for the module is used, the module lengths for calcMetrics and calcOverlap become 52 and 16 respectively.

3.3 Discussion: slice-based cohesion

This section traces the development of slice-based cohesion metrics, detailing not only the metrics suggested by Weiser and developed by Longworth and Ott and Thuss, but also alternative metrics devised by both Bieman and Ott, Krinke and Harman.

The worked examples provide illustration of slice-based cohesion metrics calculations for Weiser's parallelism, Bieman and Ott's adhesion and Ott and Thuss's five metrics. The latter are shown both in line-based and vertex-based form. Line-based calculations are given for ease of understanding, vertex-based calculations to show the result of using a slicing tool, which is the only feasible method of slicing anything but 'toy' programs. It is possible to use CodeSurfer to provide the set of lines in which the vertices in the slices occur, which, while cumbersome with version 1.9p7 of CodeSurfer, is simpler with the newer version, 2.1p1.

As the module overlap has only one function, the expected value of each of the metrics is 1. Line-based measures reflect this, but vertex-based measures do not, see Table 11, page 19. This is because some of the vertices; for example, vertices 9, 16 and 17, the body, return and exit vertex respectively, see Table 17, page 30; while included in the number of vertices, will not be included in any slice. Exclusion of these vertices from the calculations will result in the more intuitive value of 1 for the metrics. There are other vertex types, particularly those generated by CodeSurfer for internal use, such as auxiliary vertices, which should also be considered for exclusion from metric calculations. An alternative, and equivalent, method for calculating the metrics is to use the union of the vertices in the slices as the "module size". Regardless of the strategy chosen, comparisons between metric values calculated in the same way will be meaningful.

Baseline levels of slice-based cohesion metrics calculated in vertices are provided by Meyers and Binkley [29, 26, 27, 28], though it should be noted that these figures were obtained using CodeSurfer version 1.6 and differ from those produced by a more up-to-date version of the tool. There are many different ways to configure the build with a slicing tool; for example, library functions may be included or excluded, pointer analysis may be handled in different ways or single static assignment form may be invoked. This highlights the need to use the same method, tool, settings and script to produce comparable data.

Table 12: Example code: calculation of slice-based cohesion metrics

Line ref.	Code	Slice label
	`#include <stdio.h>`	
	`void calcMetrics (int sizes[], int module, int slint, int vo);`	
	`double calcOverlap (int sizes[], int slint, int vo);`	
m1.	`int main()`	
	`{`	
m2.	`    int sizes[6] = 12,6,8,8,10,6;`	c,o,n,x
m3.	`    int module = 20;`	t,c
m4.	`    int slint = 4;`	t,o
m5.	`    int vo = 6;`	c,o
m6.	`    calcMetrics (sizes, module, slint, vo);`	t,c,o,n,x
	`}`	
1.	`void calcMetrics (int sizes[], int module, int slint, int vo)`	t,c,o,n,x
	`{`	
2.	`    double min = 99999.9;`	n
3.	`    double max = 0.0;`	x
4.	`    int sumslice = 0;`	c
5.	`    double tightness, coverage, overlap;`	t,c,o
6.	`    int i;`	n,x,c
7.	`    for (int i = 0; i <vo; i++)`	n,x,c
	`    {`	
8.	`        if (sizes[i] <min) min = sizes[i];`	n
9.	`        if (sizes[i] >max) max = sizes[i];`	x
10.	`        sumslice = sumslice + sizes[i];`	c
	`}`	
11.	`    tightness = (float) slint / module;`	t
12.	`    coverage = (float) sumslice / (vo × module);`	c
13.	`    overlap = calcOverlap (sizes, slint, vo);`	o
14.	`    min = min / module;`	n
15.	`    max = max / module;`	x
16.	`    printf (”tightness = %f \ n”, tightness);`	t
17.	`    printf (”coverage = %f \ n”, coverage);`	c
18.	`    printf (”min-coverage = %f \ n”, min);`	n
19.	`    printf (”max-coverage = %f \ n”, max);`	x
20.	`    printf (”overlap = %f \ n”, overlap);`	o
	`}`	
o1.	`double calcOverlap (int sizes[], int slint, int vo)`	o
	`{`	
o2.	`    float total = 0.0;`	o
o3.	`    int j;`	o
o4.	`    for (j = 0; j <vo; j++)`	o
	`{`	
o5.	`        total = total + (float) slint/sizes[j];`	o
	`}`	
o6.	`    return total/vo;`	o
	`}`	

Key: Line references: **mn** is line n in ‘main’, **n** is line n in ‘calcMetrics’ and **on** line n in ‘overlap’.

Slice labels mark lines included in the slice on the variable denoted by the following letters:
t tightness
c coverage
o overlap
n min-coverage
x max-coverage

Table 13: Bieman and Ott's adhesion: every occurrence of a variable or constant definition or reference is a data token. A token appearing in more than one slice has an adhesion value of the number of slices it is in as a proportion of all the slices in the module; otherwise the value is zero, unless there is only one slice, when it is one.

n	vertex	variable	no.of slices	adhesion		n	vertex	variable	no.of slices	adhesion
1	3	sizes	4	0.8		33	28	overlap	1	0
2	4	sizes	4	0.8		34	29	min	1	0
3	5	module	4	0.8		35	29	min	1	0
4	6	module	4	0.8		36	29	module	1	0
5	7	slint	2	0.4		37	30	max	1	0
6	8	slint	2	0.4		38	30	max	1	0
7	9	vo	4	0.8		39	30	module	1	0
8	10	vo	4	0.8		40	43	overlap	1	0
9	12	min	1	0		41	45	max	1	0
10	13	max	1	0		42	47	min	1	0
11	14	sunslice	1	0		43	49	coverage	1	0
12	15	tightness	1	0		44	51	tightness	1	0
13	16	coverage	1	0		45	53	vo	1	0
14	17	overlap	1	0		46	54	slint	1	0
15	18	i	1	0		47	55	sizes	1	0
16	19	min	1	0		48	57	sizes	1	0
17	19	99999.9	1	0		49	57	i	1	0
18	20	max	3	0.6		50	57	min	1	0
19	20	0	1	0		51	58	sizes	1	0
20	21	sumslice	1	0		52	58	i	1	0
21	21	0max	1	0		53	58	max	1	0
22	22	i	3	0.6		54	59	sumslice	1	0
23	22	0	3	0.6		55	59	sumslice	1	0
24	23	i	3	0.6		56	59	sizes	1	0
25	23	vo	3	0.6		57	59	i	1	0
26	24	tightness	1	0		58	60	i	1	0
27	24	slint	1	0		59	61	max	1	0
28	24	module	1	0		60	61	sizes	3	0.6
29	25	coverage	1	0		61	61	i	1	0
30	25	sumslice	1	0		62	62	min	1	0
31	25	vo	1	0		63	62	sizes	1	0
32	25	module	1	0		64	62	i	1	0
			subtotal	8.6					total	9.2
			calcMetrics						mean adhesion	0.143

n	vertex	variable	no.of slices	adhesion		n	vertex	variable	no.of slices	adhesion
1	3	sizes	1	1		12	14	vo	1	1
2	4	sizes	1	1		13	15	total	1	1
3	5	slint	1	1		14	15	vo	1	1
4	6	vo	1	1		15	16	total	1	1
5	7	vo	1	1		16	16	vo	1	1
6	8	total	1	1		17	19	total	1	1
7	10	j	1	1		18	19	total	1	1
8	11	0.0	1	1		19	19	slint	1	1
9	12	j	1	1		20	19	sizes	1	1
10	12	total	1	1		21	19	j	1	1
11	14	j	1	1		22	20	j	1	1
			subtotal	11					total	22
			calcOverlap						mean adhesion	1

4 Slice-based coupling metrics

This section explains slice-based coupling metrics from both a theoretical and a practical perspective. The formulae shown in the theoretical Section, 4.1.1, are based on those given in Harman et al [14]. The practical sections include the algorithm used in a Scheme implementation for calculating these metrics in Section 4.1.2. Worked examples are shown in Section 4.2, using the program code in Table 12, page 21, with calculations based both on lines of code in Section 4.2.1 and on vertices in Section 4.2.2. A discussion on slice-based coupling follows in Section 4.3.

4.1 Background

In 1995 Harman et al. designed a slice-based coupling metric as an alternative to Henry and Kafura's information flow. They suggest that basing the metric on slices "refine the idea of information flow which has traditionally been associated with coupling measurement" [14, p.1]. While information flow notes only the presence of flow between one function and another, slice-based measures provide the number of nodes in a slice which are from another function, giving the notion of the "bandwidth" of the coupling. The flow from one module to another is calculated in terms of the vertices in the slices on each of the output variables in the module. Coupling between two modules is calculated from the normalised values of the flow in each direction.

4.1.1 Definitions

Harman et al. define the flow between two modules f and g in a program p, containing both f and g, ($FF^{p}_{(f,g)}$), as the relative amount of the body of f which contributes to the computation of the result(s) of g.

The formula given is:

$$FF^{p}_{(f,g)} = \frac{\#\left((\bigcup_{v \in p(g)} SS(p,[v],E(g))) | N(f) \right)}{N(f)}$$

where:
SS(p,[v],E(g)) means the slice on program p with respect to a set of output variables [v] at the last line of the module g
p(g) denotes the output (principal) variables of module g
N(f) denotes the number of nodes in module f

which, with reference to the example given, may be translated as:

1. Find the output variables for the function g.
2. Take full backward slices on each of these variables.
3. Form a set of vertices from the union of these slices.
4. Find the number of these vertices which are in the function f.
5. Find the total number of vertices in function f, (N(f))

6. $FF^p_{(f,g)}$ is the result of step 4 divided by the result of step 5.

Introducing a new symbol $N(f \to g)$ to mean the number of vertices (nodes) in a module, f, which are in the union of the slices on the output variables of the module g, and using Harman et al's notation for the total number of vertices in a function f, N(f), this can be stated as:

$$FF^p_{(f,g)} = \frac{N(f \to g)}{N(f)}$$

The coupling between two functions f and g, is then defined as[2]

Coupling(f,g) =

$$\frac{FF^p_{(f,g)} \times N(f) + FF^p_{(g,f)} \times N(g)}{N(f) + N(g)}$$

Rewritten:

$$\frac{\frac{N(f \to g) \times N(f)}{N(f)} + \frac{N(g \to f) \times N(g)}{N(g)}}{N(f) + N(g)}$$

Cancelling:

$$\frac{N(f \to g) + N(g \to f)}{N(f) + N(g)}$$

Meyers and Binkley [28] generalise this coupling between modules to give the coupling for a module as the weighted average of its coupling with each of the other modules in the program, giving the formula for coupling of a module f as:

Coupling (f) =

$$\frac{\Sigma_i coupling(f, g_i) \times |g_i|}{\Sigma_i |g_i|}$$

Or:

$$\frac{\Sigma_i \frac{[N(f \to g_i) + N(g_i \to f)] \times N(g_i)}{N(f) + N(g_i)}}{\Sigma_i N(g_i)}$$

4.1.2 Implementation algorithm

1. Collect the sizes of each procedure, in (non-system generated) vertices.
2. Total these sizes.
3. For each procedure, calculate the coupling with each other procedure:
   (a) Find the output variables for the procedure.
   (b) Find the sets of vertices representing slices on these variables.
   (c) Make a union of these sets.

[2]idea from [14], amended after discussion with Mark Harman, Sebastian Danicic (21.3.2007)

(d) Calculate the total number of vertices in this union from each procedure.

(e) Store for use in step 4

4. For each procedure, f, calculate coupling(f):

    (a) Calculate the sum of lengths of other procedures by subtracting own length from total of all lengths (step 2).

    (b) Calculate the coupling between the function in question and each other function, (g_i), $coupling(f, g_i)$ x length g_i.

    (c) Sum these figures and divide by sum of the lengths of all $g_i s$. (step 4a)

4.2 Worked Examples

This section shows practical coupling calculations, both for lines of code and for vertices. Whereas cohesion is calculated from intraprocedural elements of the slices, coupling is concerned with the extraprocedural elements. These are detailed in Tables 14, for line-based calculations, and Table 15, for vertices. The calculations are for the same code used for the cohesion calculations, that in Table 12, page 21. Also included for information and to aid understanding of the vertices in a system dependence graph is a list, in Table 17, showing detail of the vertices obtained by querying CodeSurfer.

4.2.1 Line-based calculations

Table 14 is in 2 parts; the top shows details for each slice in the modules, the lower shows summaries by module. For each module (g), the size of the module and the output variables in the module are listed on the left of the table. In the detail section, for each of the other modules (f), the lines from the slice on the output variables which are in that module are shown in the column headed 'Line numbers'. The union of these lines is shown in the summary section, with the number of these lines for use in the coupling calculations. The calculations follow the steps listed in section 4.1.1. This structure is repeated for vertex-based calculations in Section 4.2.2.

Table 14: Modules, sizes, output variables and extraprocedural lines in the slices on those variables

Module (g)	**Module size**	**Output variables**	**Module (f)**	**Line numbers**	**Total lines**
calcMetrics	20	tightness	main	{m1, m3, m4, m6}	(4)
		coverage	main	{m1, m2, m3, m6}	(4)
(detail)		min-coverage	main	{m1, m2, m6}	(3)
		max-coverage	main	{m1, m2, m6}	(3)
		overlap	main	{m1, m2, m4, m5, m6}	(5)
			calcOverlap	{o1, o2, o3, o4, o5, o6}	(6)
calcMetrics	20	all output variables	main	{m1, m2, m3, m4, m5, m6}	6
(summary)			calcOverlap	{o1, o2, o3, o4, o5, o6}	6
calcOverlap	6	return value	main	{m1, m2, m4, m5, m6}	5
			calcMetrics	{1,5,13,20}	4
main	6	none	calcMetrics	none	0
			calcOverlap		0

$$N(\text{main} \rightarrow \text{calcMetrics}) = 6$$
$$N(\text{calcOverlap} \rightarrow \text{calcMetrics}) = 6$$

$$N(\text{main} \rightarrow \text{calcOverlap}) = 5$$
$$N(\text{calcMetrics} \rightarrow \text{calcOverlap}) = 4$$

$$N(\text{calcMetrics} \rightarrow \text{main}) = 0$$
$$N(\text{calcOverlap} \rightarrow \text{main}) = 0$$

Coupling (main, calcMetrics)	$= 0 + 6 / 6 + 20$	$= 3/13$	$= 0.2308$
Coupling (main, calcOverlap)	$= 0 + 5 / 6 + 6$	$= 5/12$	$= 0.4167$
Coupling (calcMetrics, calcOverlap)	$= 6 + 4 / 20 + 6$	$= 5/13$	$= 0.3846$

Coupling (main) $= [(3/13) \times 20 + (5/12) \times 6] / (20 + 6)$
$= [60/13 + 30/12\ ] / 26$
$= 185 / 676$
$= 0.274$

Coupling (calcMetrics) $= [(3/13) \times 6 + (5/13) \times 6] / (6 + 6)$
$= [18/6 + 30/13] / 12$
$= 23/52$
$= 0.442$

Coupling (calcOverlap) $= [(5/12) \times 6 + (5/13) \times 20] / (20 + 6)$
$= [30/12 + 100/13] / 26$
$= 265/676$
$= 0.392$

4.2.2 Vertex-based calculations

This section follows the structure given in Section 4.2.1. Two additional tables are provided to clarify the nature and content of the vertices, Tables 16 and 17. The first of these tables lists the id numbers for non-system generated vertices for each module, with totals; the second gives more information about each of these vertices. This information is obtained by querying CodeSurfer.

Table 15: Modules, sizes, output variables and extraprocedural vertices in slices

Module (g)	**Module size**	**Output variables**	**Module (f)**	**Vertex ids**	**Total lines**
calcMetrics	55	tightness	main	{1,5,6,9,10,12,19,20}	(8)
		coverage	main	{1,4,5,7,8,9,11,12,18,20,21}	(11)
(detail)		min-coverage	main	{1,4,5,7,8,9,11,12,18,20,21}	(11)
		max-coverage	main	{1,4,5,7,8,9,11,12,18,20,21}	(11)
		overlap	main	{1,4,6,7,8,10,11,12,18,19,21}	(11)
			calcOverlap	{1,3,4,5,6,7,8,10,11,12,13,14,15,18,19,20}	(16)
calcMetrics (summary)	55	all output variables	main	{1,4,5,6,7,8,9,10,11,12,18,19,20,21}	14
			calcOverlap	{1,3,4,5,6,7,8,10,11,12,13,14,15,18,19,20}	16
calcOverlap	19	return value	main	{1,4,6,7,8,10,11,12,18,19,21}	11
			calcMetrics	{1,3,4,7,8,9,10,26,53,54,55}	11
main	19	none	calcMetrics	none	0
			calcOverlap		0

N(main → calcMetrics)	= 14
N(calcOverlap → calcMetrics)	= 16
N(main → calcOverlap)	= 11
N(calcMetrics → calcOverlap)	= 11
N(calcMetrics → main)	= 0
N(calcOverlap → main)	= 0

Coupling (main, calcMetrics)	= 14 + 0 / 55 + 19	= 14/74	= 0.2895
Coupling (main, calcOverlap)	= 11 + 0 / 19 + 19	= 11/38	= 0.1892
Coupling (calcMetrics, calcOverlap)	= 11 + 16 / 55 + 19	= 27/74	= 0.3649

Coupling (main) = [(14/74) × 55 + (11/38) × 19] / (55 + 19)
= 0.215

Coupling (calcMetrics) = [(14/74) × 19 + (27/74) × 19] / (19 + 19)
= 0.277

Coupling (calcOverlap) = [(11/38) × 19 + (27/74) × 55] / (55 + 19)
= 0.346

Table 16: The CodeSurfer vertex id numbers for the vertices in each procedure

Module	Vertex numbers (excluding system generated vertices)	Total vertices
main	{1,3,4,5,6,7,8,9,10,11,12,14,15,16,17,18,19,20,21}	(19)
calcMetrics	{1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,28,29,30,31, 33,35,37,39,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62}	(55)
calcOverlap	{1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}	(19)

4.3 Discussion: slice-based coupling metrics

Although the cohesion metrics have been well studied and documented, the coupling metrics have received less attention; in fact, the authors were unaware of an error in the formula printed in the original paper from the time of publication in 1995 until recently. Meyers and Binkley's [28] longitudinal study of slice-based coupling values for two programs, 'gnugo' and 'barcode' show that while intra-project comparisons of slice-based coupling metrics can provide information about the design of a program, inter-project comparisons may be of less use due to the difference in design styles between projects.

5 Summary

This document has three main purposes. First, it provides an analysis of the output, or principal, variables which are used as the basis for computing slices. Second, it presents an explanation of slice-based cohesion and coupling metrics and their development. Last, the calculation of slice-based metrics is explained using step-by-step worked examples based on a small sample program.

Our study shows there to be four elements which are combined in the definition of output variables. These are the function return value, modified global variables, modified reference parameters and variables printed or otherwise output by the module.

The development of slice-based cohesion metrics by different researchers is traced and the ideas for slice-based coupling metrics are introduced. To clarify the calculation of these metrics, worked examples of each metric is shown, based both on lines of code and on vertices.

This report is a collection of information and explanations aiming to be an introductory guide to slice-based metrics and their calculation.

Table 17: Description of vertices, from CodeSurfer's analysis of the program. This table shows the id and kind allocated to the vertices, together with the characters from the source code related to each vertex.

vertex id	vertex kind	characters
main		
1	entry	main()
3	body	main()
4	declaration	int sizes[6]
5	declaration	int module
6	declaration	int slint
7	declaration	int vo
8	expression	sizes[6] = 12,6,8,8,10,6
9	expression	module = 20
10	expression	slint = 4
11	expression	vo = 6
12	call-site	calcMetrics()
13	expression	j = 0
14	expression	}
15	return	}
16	exit	}
17	formal-out	main()
18	actual-in	vo
19	actual-in	slint
20	actual-in	module
21	actual-in	sizes
calcOverlap		
1	entry	calcOverlap()
3	formal in	int sizes[]
4	declaration	int sizes[]
5	formal in	int slint
6	declaration	int slint
7	formal in	int vo
8	declaration	int vo
9	body	calcOverlap()
10	declaration	double total
11	declaration	int j
12	expression	total = 0.0
14	control-point	for (j <vo
15	expression	return total / vo
16	return	return total / vo
17	exit	}
18	formal-out	calcOverlap()
19	expression	total = total + (float) slint / sizes[j]
20	expression	j++
calcMetrics		
1	entry	calcMetrics()
3	formal-in	int sizes[]
4	declaration	int sizes[]
5	formal-in	int module
6	declaration	int module
7	formal-in	int slint
8	declaration	int slint
9	formal-in	int vo
10	declaration	int vo
11	body	calcMetrics()
12	declaration	double min
13	declaration	double max
14	declaration	int sumslice
15	declaration	double tightness
16	declaration	double coverage
17	declaration	double overlap
18	declaration	int i
19	expression	min = 99999.9
20	expression	max = 0.0
21	expression	sumslice = 0
22	expression	i = 0
23	control-point	for (i <vo
24	expression	tightness = (float) slint / module
25	expression	coverage = (float) sumslice / (vo ∗ module)
26	call-site	calcOverlap()
28	expression	overlap = calcOverlap ()
29	expression	min = min / module
30	expression	max = max / module
31	call-site	printf ()
33	call-site	printf ()
35	call-site	printf ()
37	call-site	printf ()
39	call-site	printf ()
41	return	}
42	exit	}
43	actual-in	overlap
44	actual-in	"overlap = % f \n "
45	actual-in	max
46	actual-in	"max-coverage = % f \n "
47	actual-in	min
48	actual-in	"min-coverage = % f \n "
49	actual-in	coverage
50	actual-in	"coverage = % f \n "
51	actual-in	tightness
52	actual-in	"tightness = % f \n "
53	actual-in	vo
54	actual-in	slint
55	actual-in	sizes
56	actual-out	calcOverlap ()
57	control-point	if (sizes[i] <min)
58	control-point	if (sizes[i] >max)
59	expression	sumslice = sumslice + sizes[i]
60	expression	i++
61	expression	max = sizes[i]
62	expression	min = sizes[i]

References

[1] J. M. Bieman and L. M. Ott. Measuring functional cohesion, 0098-5589. *Software Engineering, IEEE Transactions on*, 20(8):644–657, 1994.

[2] D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[3] D. Binkley, M. Harman, and J. Krinke. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30(1):3, 2007.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[5] CodeSurfer®. Grammatech Inc. http://www.grammatech.com/products/codesurfer.

[6] A. de Lucia. Program slicing: Methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Florence, Italy, Nov. 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–177, New York, NY, USA, 2000. ACM.

[8] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[9] K. Gallagher and D. Binkley. Program slicing. In *24th IEEE International Conference on Software Maintenance, Frontiers of Software Maintenance*, Beijing, China, October 2008. ICSM, IEEE.

[10] M. Harman. Cleaving together - program cohesion with slices. *EXE magazine*, 11(8):35–42, 1997.

[11] M. Harman, S. Danicic, Y. Sivagurunathan, B. Sivagurunathan, and B. Jones. Cohesion metrics. In *8th International Software Quality Week (QW'95)*, San Francisco, CA, USA, 1995.

[12] M. Harman, N. Gold, R. Hierons, and D. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 11, Washington, DC, USA, 2002. IEEE Computer Society.

[13] M. Harman and R. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[14] M. Harman, M. Okulawon, B. Sivagurunathan, and S. Danicic. Slice-based measurement of function coupling. In *8th International Software Quality Week (QW'95)*, San Francisco CA, 1995. paper 4-T-4.

[15] S. Henry and D. Kafura. Software structure metrics based on information flow. In *Software Engineering, IEEE Transactions on , vol.SE-7, no.5*, pages 510–518, 1981.

[16] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 392–411, New York, NY, USA, 1992. ACM.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–60, 1990.

[18] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[19] D. C. Ince and M. J.Shepperd. An empirical and theoretical analysis of an information flow-based system design metric. In *ESEC*, pages 86–99, 1989.

[20] S. Karstu. *An examination of the behaviour of slice based cohesion measures.* Msc, Michigan Technological University, 1994.

[21] J. Krinke. Statement-level cohesion metrics and their visualization. In *SCAM 2007 Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, Paris, France, 2007. IEEE Computer Society Press.

[22] A. Lakhotia. Rule-based approach to computing module cohesion. In *Proceedings 15th ICSE*, pages 35–44, Baltimore, Maryland, USA, 1993. IEEE Computer Science Press.

[23] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[24] H. Longworth. Slice based program metrics. Master's thesis, Computer Science, Michigan Technical University, Michigan, USA, 1984.

[25] A. Majumdar, S. Drape, and C. Thomborson. Metrics-based evaluation of slicing obfuscations. *Information Assurance and Security*, 0:472–477, 2007.

[26] T. Meyers and D. Binkley. A large scale empirical study of slice-based metrics. In *MASPLAS'04 Mid Atlantic Student Workshop on Programming Languages and Systems*, Seton Hall University, 2004.

[27] T. Meyers and D. Binkley. A longitudinal and comparative study of slice-based metrics. In *10th International Software Metrics Symposium*, Chicago, USA, 2004.

[28] T. Meyers and D. Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Methodology (TOSEM)*, 17(1):1–27, 2007.

[29] T. M. Meyers and D. Binkley. Slice-based cohesion metrics and software intervention. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, page 256, 2004.

[30] L. M. Ott and J. Thuss. The relationship between slices and module cohesion. In *Software Engineering, 11th International Conference on*, 1989.

[31] L. M. Ott and J. J. Thuss. Slice based metrics for estimating cohesion. In *IEEE-CS International Metrics Symposium*, pages 78–81, 1993.

[32] M. Page-Jones. *The practical guide to structured systems design: 2nd edition*. Yourdon Press, Upper Saddle River, NJ, USA, 1988.

[33] K. Pan, S. Kim, and J. E. James Whitehead. Bug classification using program slicing metrics. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 31–42, Washington, DC, USA, 2006. IEEE Computer Society.

[34] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, 1991.

[35] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[36] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[37] M. Weiser. *Program Slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, AnnArbor, MI, 1979.

[38] M. Weiser. Program slicing. In *5th international conference on Software engineering*, pages 439–449, San Diego, California, USA, 1981. IEEE Press.

[39] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design.* Prentice Hall, 1979.

[40] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.