Using Maximal Slices as a Solution to the ‘Output Variable’ Problem for Calculating Slice-Based Cohesion Metrics

James Hamilton
Department of Computing
Goldsmiths, University of London
United Kingdom
Email: james.hamilton@gold.ac.uk

Sebastian Danicic
Department of Computing
Goldsmiths, University of London
United Kingdom
Email: s.danicic@gold.ac.uk

***Abstract*—Software metrics are widely used to quantify software quality; cohesion is one such metric. It is a measure of the ‘inter-relatedness’ of code. Sliced-based cohesion metrics use program slicing to calculate the cohesion of a software module by using ‘output variables’ as slicing criteria. The definition of ‘output variables’, however, varies in different studies. Unfortunately, these different definitions lead to different values for the metrics. We solve this problem by introducing standardised versions (independent of the ‘output variables’) of previous definitions of the slice-based metrics.**

Our approach computes metrics based only on the *maximal* slices in a software module. These can be computed automatically without the need to specify which points contain ‘output variables’.

We call the slicing criteria that generate maximal slices, in a software module, ‘crucial points’. Empirical evidence suggests that crucial points represent points of interest in a program and strongly correlate with previous definitions of ‘output variables’. We believe that using crucial points instead of output variables in slice-based metrics may lead to a more intuitively accurate measurement of cohesion.

I. Introduction

Program metrics have long been used in an attempt to quantify code quality, for example for use in refactoring [19], improving software [8] and finding bugs [14]. A software metric usually assigns some number to a certain quality of a program. Cohesion metrics [23] attempt to quantify the inter-relatedness of code in a program module. A highly cohesive module suggests that it’s statements are highly inter-related and perform one job; whereas a poorly cohesive module may be performing multiple jobs. Ideally, program modules should have a high cohesion corresponding to a modular design - where each task is separated into individual modules - which reduces complexity and allows programmers to work on modules in isolation [18].

There are several cohesion metrics which may defined using program slicing, or other information about a program module.

Program slicing [22] involves the calculation of a subset of a program which affects some point of interest, known as the slicing criteria. Program slicing is used to aid program understanding and debugging by reducing the size of the code which has to be examined. For example, computing a backward slice on line 9 in the sum/product example (listing 2, page 3) results in the slice containing lines $\{9, 7, 6, 5, 3, 2, 1\}$. Program slicing can also be used for testing, code verification, restructuring and in the calculation of program metrics.

A. Slice-Based Cohesion Metrics

Sliced-based cohesion metrics attempt to determine the cohesiveness of a program module by calculating the intersection of slices. Using slices to calculate cohesion metrics works well due to the relatedness definition of cohesion. If statements in a module are related it is likely that their slices will share many statements. On the other hand, if there are multiple tasks taking place in a module it is likely that the intersection will be small or empty.

Weiser [22] defined three metrics related to cohesion, which have been further refined and expanded by others [7, 9, 12, 13]:

tightness
: the ratio of the size of the slice intersection to the total module length

mininmum coverage
: the ratio of the smallest slice to the total module length

coverage
: the ratio of the average slice size to the total module length

maximum coverage
: the ratio of the largest slice size to the total module length

overlap
: the average ratio of the intersection of all slices to slice size

The definition of a program module under consideration in previous empirical studies [10, 11], and this study, is a procedure, but could also be, for example, a class, a package, whole program etc.

II. The Output Variable Problem

The calculation slice-based cohesion metrics poses a problem: what are the slicing criteria? Previously, the concept of an 'output variable' has been used. The definition of an 'output variable' varies between studies; this is a problem for replicating or comparing such studies.

Green et al. [5] compared 11 papers which used 'output variables' and came to the conclusion that there are four main types of 'output variables': "the return value of a function; global variables modified by a fucntion; parameters passed by reference to, and modified by, a function; and any variables printed, written to a file, or otherwise output to the external environment" [5]. Most of the previous studies in the area of slice-based metrics have used CodeSurfer [4] which works with C programs. The last category, according to Green et al., is the most difficult to capture with CodeSurfer (as it is the most difficult to define).

Cohesion is a measure of how strongly-related the statements within a program module are. Imagine a million print statements of the form in listing 1 where we are printing the result of a function call which has a side-effect that changes the value of x depending on it's previous value. Intuitively, the example in listing 1 is very cohesive because each such statement is dependent on all the previous statements in the sequence.

```
int f(int k) {
  x = k + x;
  return x;
}

void main() {
  print(f(0) + x);
  print(f(1) + x);
  ...
  print(f(1000000) + x);
}
```

Listing 1. extreme example

The main problem with computing slice-based metrics is deciding which slices to include in the calculations. If we wanted to compute slice-based cohesion metrics for the example in listing 1 previous studies would have chosen the points where variables were output as the slice points. In this case every line in the main method, as x is printed on every line.

If we slice at each print statement in order to compute the various metrics we will end up with very extreme values. The tightness will be approximately 0, mincoverage 0, coverage 0.5, maxcoverage 1.

This is an example where maximal slices give use exactly what we want because slicing on the last line will give a maximal slice; resulting in a tightness of 1. It could also happen that there are no 'output variables', according to some definition, within a module; but using maximal slices every module has at least one maximal slice. Maximal slices cope with extreme cases but also cope with sensible cases as well.

It turns out that the values of the metrics are highly sensitive to the choice of slices used in their computation. Many authors have grappled with the problem of which slices to include in order to come up with meaningful values for the metrics. We feel that the choices appear somewhat arbitrary and are in some sense trying to second-guess what the slices of interest are.

The aim of this paper is to get around this problem by giving a definition that is language independent, which gives sensible values for these metrics without having to make arbitrary decisions about which slices to include. We solve the problem of the varying definitions of 'output variable' in this paper by defining slice-based cohesion metrics based on maximal slices.

III. Maximal Slices

We propose the use of maximal slices as the basis of the definition of slice-based cohesion metrics. In our definition there is no need to define the concept of an 'output variable' which allows us to provide a standard definition of slice-based cohesion metrics.

In this section we think of a module M as the set containing the statements in the module, for example, we write $|M|$ to represent the number of statements in a module.

Definition 1 (The Set of all Slices in a Module): Given a module M we define $\text{Slices}(M)$ to be the set of all slices of M.

Definition 2 (Maximal Slice): A maximal slice is a slice that is not a subset of any other slice.

Definition 3 (The Set of all Maximal Slices): Given a module M we define $\text{maxSlices}(M)$ to be the set of all maximal slices of M. Formally,

$$S \in \text{maxSlices}(M) \iff \begin{cases} S \in \text{Slices}(M) \\ \text{and} \\ \forall k \in \text{Slices}(M), \\ \qquad S \subseteq k \implies S = k. \end{cases}$$

Definition 4 (Crucial Point): A crucial point is slicing a criterion which produces a maximal slice.

Definition 5 (The Intersection of all Maximal Slices): Given a module M we write $\mathcal{I}(M)$ for the intersection of all maximal slices. Formally,

$$\mathcal{I}(M) = \bigcap_{S \in \text{maxSlices}(M)} S$$

Definition 6 (Tightness): The tightness is the ratio of the intersection of all maximal slices to the size of the module. The tightness of a module M is defined by the following formula:

$$\text{tightness}(M) = \frac{|\mathcal{I}(M)|}{|M|}$$

Definition 7 (Minimum Coverage): The minimum coverage is the ratio of size of the smallest maximal slice to the size of the module. The minimum coverage of a module M is defined by the following formula:

$$\text{mincoverage}(M) = \frac{\min\{|S| : S \in \text{maxSlices}(M)\}}{|M|}$$

Definition 8 (Coverage): The coverage is the ratio of the average maximal slice to the size of the module. The coverage of a module M is defined by the following formula:

$$\text{coverage}(M) = \frac{\sum\limits_{S \in \text{maxSlices}(M)} |S|}{|M||\text{maxSlices}(M)|}$$

Definition 9 (Maximum Coverage): The maximum coverage is the ratio of size of the largest maximal slice to the size of the module. The maximum coverage of a module M is defined by the following formula:

$$\text{maxcoverage}(M) = \frac{\max\{|S| : S \in \text{maxSlices}(M)\}}{|M|}$$

Definition 10 (Overlap): The overlap is the average ratio of the intersection of all maximal slices to slice size. The overlap of a module M is defined by the following formula:

$$\text{overlap}(M) = \frac{\sum\limits_{S \in \text{maxSlices}(M)} \frac{|\mathcal{I}(M)|}{|S|}}{|\text{maxSlices}(M)|}$$

We believe that maximal slices capture the most interesting parts of a program; previous studies have attempted to gather the same useful information by selecting slicing criteria. Slice-based cohesion metrics slices are calculated by using a set of slices - the ‘output variables’ are not important, the slices are. If a slice is maximal it would suggest that the statements in that slice perform one task; whereas a slice that is a subset of another slice could be thought of as performing a sub-task of a larger task.

Disjoint maximal slices would suggest that the set of statements in each slice are performing completely separate tasks; for example, if we calculate maximal slices for a jar file containing multiple class files we may find completely disjoint maximal slices if each class is an unrelated program. Ideally, the intersection of maximal slices of a program method should be large, corresponding to a highly inter-related set of program statements. However, if we are considering Java classes or packages we would consider a smaller intersection between them good, corresponding to the modularity of object-oriented programming.

In order to calculate the values of the slice-based cohesion metrics, backward slices are computed for each statement in a module, starting with the last statement. Maximal slices are computed from the set of slices and used to calculate the metrics.

For example, there are 2 maximal slices for listing 2 - $\{9, 7, 6, 5, 3, 2, 1\}$ and $\{8, 7, 6, 4, 3, 2, 0\}$; these are the only slices that are not subsets of other slices. The crucial points are lines 8 and 9 as these give the maximal slices. Notice that the crucial points *contain ‘output variables’* (i.e. they are variables printed to the console in these lines). The length of the module is 10 and the intersection of the maximal slices is $\{7, 6, 3, 2\}$, so the size of the intersection is 4. The smallest and largest maximal slices are both 7.

```
0: sum = 0;
1: product = 0;
2: i = 0;
3: while(i < 10) {
4:  sum = sum + i;
5:  product = product * i;
6:  i = i + 1;
7: }
8: print sum
9: print product
```

Listing 2. sum/product example - psuedocode

Now, using our definitions we can calculate cohesion metrics for this example:

$$\begin{aligned}
\text{tightness(example)} &= \frac{4}{10} \\
\text{mincoverage(example)} &= \frac{7}{10} \\
\text{coverage(example)} = \frac{7+7}{10 \times 2} &= \frac{7}{10} \\
\text{maxcoverage(example)} &= \frac{7}{10} \\
\text{overlap(example)} = \frac{\frac{4}{7} + \frac{4}{7}}{2} &= \frac{4}{7}
\end{aligned}$$

These metrics involve taking the intersections of slices so including small slices in our set will necessarily lead to small values of these metrics. If slice points are arbitrarily chosen to calculate metrics and it happens that very small slices are included then, when averaged over the program, all large programs will have, for example small tightness and small overlap - tightness and overlap are proportional to the intersection.

Maximal slices are special because any code that occurs in a maximal slice cannot affect code outside a maximal slice; this guarantees that “all points of interest” that may be affected by the slice will be included in the slice. The pleasing “closed property” of maximal slices make them an ideal choice for computing slice-based metrics, removing the need for previous arbitrary choices. Maximal slices are slices which have “maximal effect”.

IV. Empirical Study

We have used the Indus Java program slicer [15, 16] to perform an empirical study of 378 small Java programs taken from SourceForge.net. Indus performs analysis and slicing on the Jimple [20] intermediate-representation provided by the Soot [21] framework. Jimple is a fully typed three-address code representation of Java/Java bytecode.

Many of the crucial points correspond to the concept of 'output variable' as used in previous studies – the difference being that we did not have to specify what an 'output variable' is to find them.

Table I shows a selection of crucial point statement types found in our study of 378 Java programs. We have chosen the listed types by considering, intuitively, the types of statements in Java that might correspond to the 'output variables' in previous studies. The types listed correspond to approximately 70% of the overall statements designated as crucial points.

Nearly 100% of Java Writer/Stream method calls (print(ln), close, flush, write) were designated crucial points. These methods cause some kind of output and therefore are the most obvious methods to be in definitions of 'output variables'.

Thread related, System.exit() and Runtime.exec() method calls affect the environment outside of the module and should therefore be considered 'output variables'; nearly all of these were found to be crucial points.

A high percentage of modified instance variables (fields) were designated as crucial points. These include both the current object and other objects used within a method; e.g. "this.foo = 1", "obj.foo = 2". Java uses *pass-by-value* where a value passed as a parameter is a reference to an object therefore modified instance variables of other objects, used in a method, are considered crucial points. Similarly a high percentage of class variables (static fields) have been designated crucial points.

Another of the types of 'output variable' in previous studies were return values. Our study found that 73.46% of return statements returning variables were found to be crucial points.

The "or similar" part of "printed or similar" type of 'output variables' is the most difficult to define. We found that 72.17% of method invocations with void return types and 1 or more parameters were considered crucial points; such a method is likely to cause some external changes, such as print or modify objects, because it doesn't return any value. A method that returns a value is less likely to have side-effects. As a more concrete example, 97.73% of 'setters' were found to be crucial points compared to only 23.44% of 'getters'[1].

We have shown examples which coincide with previous definitions of output variables. However, it is not important using our technique to determine whether or not a specific method call is or is not an output method. The advantage of our technique is that we use maximal slices to determine the interesting parts of a program for us.

Each method will have at least 1 crucial point and the average number of crucial points per method, in our study, is 4.76. The average length of a method is 13.57 Jimple statements. Java programs, compared with other languages such as C, contain a high number of methods with a small length (e.g. $<$ 3 Jimple statements) due to the use of 'getters' and 'setters' in object-oriented programming, and default constructors. This skews results when averaging slice-based cohesion metrics of methods in a program (as previous studies have done) and makes it hard to compare the metrics to other languages. Our study revealed that 38% of methods contain fewer than 3 Jimple statements.

[1]We consider a 'setter' to be of the form 'void set[alphanumeric]*(parameter+)' and a 'getter' 'nonvoid get[alphanumeric]*()'

Type	CPs	!CPs
*Stream print(ln)	100%	0%
*Writer print(ln)	100%	0%
*Stream close()	100%	0%
*Writer close()	100%	0%
*Stream flush()	100%	0%
*Writer flush()	100%	0%
*Stream write()	100%	0%
*Writer write()	99.7%	0.3%
printStackTrace	100%	0%
Thread start	100%	0%
Thread stop	100%	0%
Thread sleep	100%	0%
Thread currentThread	100%	0%
System.exit()	100%	0%
Runtime.exec()	80%	20%
modified instance vars (*this*)	93.21%	6.79%
modified instance vars (others)	89.5%	10.5%
modified class vars (same class)	80.28%	19.72%
modified class vars (other class)	62.16%	37.84%
returns (with vars)	73.46%	26.54%
returns (with constants)	72.8%	27.2%
Reflect invoke	100%	0%
*Exception initialisers	96.58%	3.42%
void method invocations	72.17%	27.83%
setter invocations	97.73%	2.27%
getter invocations	23.44%	76.56%

TABLE I
SELECTED JAVA STATEMENTS THAT ARE/ARE NOT CRUCIAL POINTS

V. CONCLUSION AND FUTURE WORK

We have used maximal slices to redefine slice-based cohesion metrics without the use of the previously ambiguous concept of 'output variables'. This standard definition is an improvement because it will allow further studies in this area to be more easily comparable.

The use of maximal slices is appropriate because we believe that a maximal slice captures an interesting part of a program; any code that occurs in a maximal slice cannot affect code outside the maximal slice. Intersecting maximal slices represent inter-related program statements, while disjoint maximal slices suggest multiple tasks are being performed. For future work we will define new metrics in terms of maximal slices which we believe will compute slice-based cohesion metrics more intuitively, especially in the context of object-oriented programming.

A fruitful avenue of research that we are now investigating is representing a program module as a graph of slices. The visualisation of these graphs may give a better intuitive insight into the cohesive properties of programs rather than a simple value. A further advantage is that it gives the opportunity of defining metrics in terms of standard graph metrics, such as communities [1].

These visualisation might also be useful for locating software watermarks [3, 6] or malware [2, 17] within programs - allowing us to define better software watermarks, or easily remove malware.

References

[1] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, October 2008. ISSN 1742-5468. doi: 10.1088/1742-5468/2008/10/P10008.

[2] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. pages 12–12, Washington, DC, 2003. USENIX Association.

[3] Christian Collberg. Software watermarking: Models and dynamic embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT*, January 1999.

[4] GrammaTech, Inc. CodeSurfer, 2011. URL http://www.grammatech.com/.

[5] Pam Green, Peter C R Lane, Austen Rainer, and S Scholz. An Introduction to Slice-Based Cohesion and Coupling Metrics. Technical Report 488, University of Hertfordshire, 2009.

[6] James Hamilton and Sebastian Danicic. A Survey of Static Software Watermarking. In *Proceedings of the World Congress on Internet Security 2011*, pages 114–121, London, 2011. IEEE.

[7] M. Harman, S. Danicic, Y. Sivagurunathan, and B. Jones. Cohesion Metrics, 1995.

[8] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2006. ISBN 3540244298.

[9] H. Longworth. *Slice Based Program Metrics*. Masters, Michigan Technical University, 1984.

[10] Timothy M. Meyers and David Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–27, December 2007. ISSN 1049331X. doi: 10.1145/1314493.1314495.

[11] T.M. Meyers and David Binkley. Slice-based cohesion metrics and software intervention. In *11th Working Conference on Reverse Engineering*, pages 256–265. IEEE Computer Society, November 2004. ISBN 0-7695-2243-2. doi: 10.1109/WCRE.2004.34.

[12] L.M. Ott and J.J. Thuss. The Relationship Between Slices And Module Cohesion. In *11th International Conference on Software Engineering*, pages 198–204. IEEE, 1989. ISBN 0-8186-8941-2. doi: 10.1109/ICSE.1989.714420.

[13] L.M. Ott and J.J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings First International Software Metrics Symposium*, pages 71–81. IEEE Comput. Soc. Press, 1993. ISBN 0-8186-3740-4. doi: 10.1109/METRIC.1993.263799.

[14] Kai Pan, Sunghun Kim, and James Whitehead. Bug Classification Using Program Slicing Metrics. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 31–42. Ieee, 2006. ISBN 0769523536. doi: 10.1109/SCAM.2006.6.

[15] V.P. Ranganath. *Scalable and accurate approaches for program dependence analysis, slicing, and verification of concurrent object oriented programs*. Phd, KANSAS STATE UNIVERSITY, 2007.

[16] V.P. Ranganath. Indus - Java Program Slicer, 2009. URL http://indus.projects.cis.ksu.edu/.

[17] M. A Siddiqui. *Data Mining Methods for Malware Detection*. PhD thesis, Florida, 2008.

[18] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and G.L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, January 2002. ISSN 1350-1917. doi: 10.1046/j.1365-2575.2002.00117.x.

[19] Ricky E Sward, AT Chamillard, and D Cook. Using Software Metrics and Program Slicing for Refactoring. *The Journal of Defense Software Engineering*, (July), 2004.

[20] R. Vallee-Rai and L.J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, Montreal, Quebec, Canada, 1998.

[21] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[22] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0897911466.

[23] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, February 1979. ISBN 0138544719.