Effects of Software Changes on Module Cohesion

Linda M. Ott
Dept. of Computer Science
Michigan Technological University
Houghton, MI 49931

James M. Bieman
Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523

Abstract

We use program slices to model module cohesion. For our purposes, a slice is a projection of program text that includes only the data tokens relevant to one output. We define six cohesion metrics in terms of these slices, and evaluate the effects of classes of module changes on these metrics. We find that the effects on cohesion metrics are notably more predictable when the changes result from adding code rather than from moving code. In general, the effects that software changes have on the cohesion metrics match our intuition.

1 Introduction

Changes made during software maintenance can have negative effects on the internal structure of a software system. As changes are made over time, the software can become more difficult to understand and maintain. Maintainers will surely benefit from tools to help evaluate the effects of a change on the structural integrity of a software system.

Module cohesion is one software attribute that can be affected by changes. A cohesive module has one basic function. Changes can introduce auxiliary functionality to existing program units resulting in less cohesive modules. We use a variation on program slices, introduced by Weiser [24], to model and measure cohesion [18].

The effects of a change on cohesion are not always obvious. A slice profile is one heuristic tool that can help a maintainer visualize the cohesion in a module [17]. By seeing the effect of a change on module cohesion, a maintainer can get intuitive feedback on the impact of the change. In this paper, we analyze the effects of changes on a set of cohesion metrics. These metrics provide a numeric view of the effect of a change, and can allow a maintainer to evaluate the relative effect of alternative changes. Quantitative metrics allow comparisons between alternative maintenance (and design) decisions, and allow a maintainer to monitor the effect of his or her actions on the software structure.

We focus on the direction of the changes to cohesion metrics resulting from relatively simple code modifications. The direction of metric changes provides a ranking of relative levels of cohesion before and after a code change. Such a ranking helps provide a basic understanding of cohesion attributes [16], and demonstrates the scale properties and the arithmetic operations that can be applied to the metric values [28].

For metrics to provide meaningful measurements they must be rigorously defined, accurately reflect well understood software attributes, and be based on models that capture these attributes [1]. The measures should be specified independently from the measurement tools, and, in fact, such tools should be based on the models. Example tools include QUALMS [27], which is based on the flowgraph model, and the set of test coverage measurement tools developed by Bieman and Schultz [3, 4] based on the *standard representation* model [2]. We use a slice model of a program to develop measures of module cohesion, and then evaluate the effect of code changes to the measures.

The paper has the following organization. In Section 2, we define program slicing and a program model based on slices, and use this model to define several cohesion metrics. In Section 3, we evaluate the effects of code changes on the slice based program models. Section 4 reports on the effects of code addition changes on the cohesion metrics, and Section 5 reports on the more unpredictable metric effects from moving code. Our conclusions are given in Section 6.

2 Program Slicing

Slicing is a method of program reduction introduced by Weiser [24, 25, 26]. A *slice* of a module at statement s with respect to variable v is the set of all statements and predicates that might affect the value of v at s. Slices were proposed as potential debugging tools and program understanding aids. They have since been used in a broader class of applications (e.g., debugging parallel programs [5], maintenance [6, 8, 17], and testing [10, 11, 15, 20]).

Weiser's algorithm for computing slices is based on data flow analysis. It is suggested in [21] that a *program dependence graph* representation could be used to compute slices more efficiently and precisely. An algorithm for computing slices using a *program dependence graph* representation is presented by Horwitz, Reps, and Binkley [9, 22]. A slice is obtained by walking backwards over the program dependence graph to obtain all nodes which have an effect on the value of the variable of interest. Similarly, a *forward slice* [9] can be obtained by walking forward over the program dependence graph to obtain all nodes which are affected by the value of a variable. The algorithm based on the program dependence graph is more restricted than Weiser's in the sense that it will only compute a slice for variable v at statement s if v is defined or used in statement s. Both *intra*procedural slices and *inter*procedural slices can be computed. *Intra*procedural slices are restricted to within a single procedure while *inter*procedural slices cross the boundaries of procedural calls.

2.1 Metric Data Slices

In [26], Weiser defined several slice based metrics. Longworth [14] first studied their use as indicators of cohesion. In [19, 23], certain inconsistencies noted by Longworth are eliminated through the use of *metric slices*. A metric slice takes into account both the *uses* and *used by* data relationships [7]; that is, they are the union of Horwitz et.al.'s backward and forward slices.

In order to analyze the effects of changes on slice metrics, we modify this concept of metric slices to use data tokens (i.e., variable and constant definitions and references) rather than statements as the basic unit. We call these slices *metric data slices*.

Using data tokens as the basis of the slices ensures that all changes of interest will cause a change in at least one slice of a module. We consider a change of interest to be any change which could have an effect on the cohesiveness of a module. An example of a change that is *not* of interest is changing some operator to a different operator. Examples of changes of interest include adding code, deleting code, or changing a variable name. Each of these changes would result in a change to at least one metric data slice. (This is in contrast to a metric slice, where if a statement is modified, the number of statements in the slice might not change.)

Informally, we view a *metric data slice* for a data token, v, as the set of all data tokens in the statements that comprise the "backward" and "forward" slices of v. We use *intra*procedural slicing since we are interested in examining the cohesiveness of each procedure as a separate entity.

We view a software system as a set of *slice modules* where each slice module is a set of metric data slices. A metric data slice is computed for each output of the module. Since we are interested in the cohesion of the whole module, we use a concept similar to that of *end-slices* [12, 13]. The "backward" slices are computed from the end of the module [1] and the "forward" slices are computed from the "top"s of the backward slices.

M will be used to represent the set of slice modules for module m. Figure 1 contains an example of a metric data slice.

2.2 Data Slice Profiles

Slice profiles were developed to aid in visualizing the relationships among the slices generated for a module [18, 23]. A slice profile for procedure *SumAndProduct* is given in Figure 2. Each column with a variable name heading in the slice profile corresponds to a slice of that variable. All rows in the profile marked with a vertical bar "|" are statements included in a slice for a particular variable, otherwise the row is blank. The "Statement" column contains the source statement. For example, the column with heading *SumN* in Figure 2 corresponds to the slice for *SumN*. This slice consists of all statements containing a vertical bar in the column for *SumN*.

Although they do not completely capture metric data slices, we can modify slice profiles to give a sense of the relationships among metric data slices. To do this, we indicate in the column for a slice variable, the number of data tokens in that line that are included in the slice. Figure 3 shows an example of a metric data slice profile.

2.3 Data Slice Metrics

In his original work on slicing, Weiser proposes several metrics. Longworth [14] and Thuss [23] show that Weiser's metrics are related to cohesion. Here we redefine the metrics in terms appropriate for metric data slices.

For notational convenience, let V_m be the set of variables used by module m and let V_O be a subset of V_m containing only the output variables of module m. Output variables include parameters and globals that are modified, variables that are written by the module, and the return value of a function. A slice module M for module m is defined by

$$M = \{SL_i \mid v_i \epsilon V_O\} \qquad (1)$$

where the symbol SL_i is the metric data slice obtained for v_i. Let $SL_{int}(M)$ be the intersection of all the SL_i.

$$SL_{int}(M) = \bigcap_{i=1}^{|M|} SL_i. \qquad (2)$$

The size of a module, m, denoted size(m), is the total number of data tokens in m. Since slices have been defined here as sets, $|SL_i|$ is the number of data tokens in the slice SL_i.

Coverage is a comparison of the size of the slices to the size of the module. Low *Coverage* values generally result from modules with numerous short slices, and may be an indication of several distinct processing elements and, therefore, low cohesion. *Coverage* is defined as the mean slice size divided by the module size.

$$Coverage(M) = \frac{1}{|M|} \sum_{i=1}^{|M|} \frac{|SL_i|}{size(m)} \qquad (3)$$

Overlap measures the average ratio of data tokens common to all the slices. A high *Overlap* might indicate high code interdependence since most slices span

[1] That is from the *FinalUse* nodes as described in [9]

```
procedure SumAndProduct( N : integer; var SumN, ProdN : integer );
var
  I : integer;
begin
  SumN := 0;
  ProdN := 1;
  for I := 1 to N do begin
    SumN := SumN + I;
    ProdN := ProdN * I
  end
end;
```

Figure 1: Metric data slice for *SumN* . Items included in the slice are contained within boxes.

<table>
<tr><th>SumN</th><th>ProdN</th><th>Statement</th></tr>
<tr><td>|</td><td>|</td><td>procedure SumAndProduct(N : integer; var SumN, ProdN : integer);</td></tr>
<tr><td>|</td><td>|</td><td>var</td></tr>
<tr><td>|</td><td>|</td><td> I : integer;</td></tr>
<tr><td>|</td><td>|</td><td>begin</td></tr>
<tr><td>|</td><td></td><td> SumN := 0;</td></tr>
<tr><td></td><td>|</td><td> ProdN := 1;</td></tr>
<tr><td>|</td><td>|</td><td> for I := 1 to N do begin</td></tr>
<tr><td>|</td><td></td><td>   SumN := SumN + I;</td></tr>
<tr><td></td><td>|</td><td>   ProdN := ProdN * I</td></tr>
<tr><td>|</td><td>|</td><td> end</td></tr>
<tr><td>|</td><td>|</td><td>end;</td></tr>
</table>

Figure 2: Slice profile for ***SumAndProduct***. Statements included in the slice for ***SumN*** are indicated with a "|" in column ***SumN*** of the profile. Similarly, the slice for ***ProdN*** is indicated in column ***ProdN***.

SumN	ProdN	Statement
2	2	procedure SumAndProduct(N : integer; var SumN, ProdN : integer);
		var
1	1	I : integer;
		begin
2		SumN := 0;
	2	ProdN := 1;
3	3	for I := 1 to N do begin
3		SumN := SumN + I;
	3	ProdN := ProdN * I
		end
		end;

Figure 3: Metric Data Slice profile for *SumAndProduct*. The number of data tokens included in the metric data slice for *SumN* are indicated with a number in column *SumN* of the profile. The metric data slice for *ProdN* is indicated in column *ProdN*.

the entire module, and implies a high degree of interrelationship and high cohesion. *Overlap* was originally defined by Weiser as the average of the ratio of non-unique to unique statements in each slice. Because *Overlap* is undefined if there are no unique statements in a slice, Longworth [14] redefined *Overlap* in terms of the total number of statements in each slice. Similarly, we define *Overlap* as the ratio of the number of data tokens in the intersection of all of the slices to the size of each slice.

$$Overlap(M) = \frac{1}{|M|} \sum_{i=1}^{|M|} \frac{|SL_{int}(M)|}{|SL_i|} \quad (4)$$

Tightness is based on the number of data tokens included in every slice. High *Tightness* values tend to indicate a high degree of data relationships within the module, possibly indicating a highly functional module. *Tightness* is expressed as a ratio of the number of data tokens in the intersection of all the slices over the module size.

$$Tightness(M) = \frac{|SL_{int}(M)|}{size(m)} \quad (5)$$

Parallelism indicates the number of slices with little in common, and may reflect the number of "unrelated" processing elements within a module. *Parallelism* is expressed as the number of slices having a pairwise overlap with all of the other slices less than or equal to a threshold, τ.

$$Parallelism(M) = |\{SL_i \text{ such that } |SL_i \bigcap SL_j| < \tau \text{ for all } j \neq i\}| \quad (6)$$

MinCoverage is the size of the shortest slice as a ratio to the module size. High *MinCoverage* values indicate that the shortest slice requires most of the data tokens in the module, and therefore, that all of the slices interact.

$$MinCoverage(M) = \frac{1}{size(m)} \min_i |SL_i| \quad (7)$$

MaxCoverage is the size of the longest slice as a ratio to the module size. High *MaxCoverage* values indicate that at least one slice requires most of the data tokens in the module.

$$MaxCoverage(M) = \frac{1}{size(m)} \max_i |SL_i| \quad (8)$$

Values of *Coverage*, *Overlap*, *Tightness*, *MinCoverage* and *MaxCoverage* will range from 0 to 1, with the assumption that higher values indicate more cohesive modules. Values of *Parallelism* will range up from 0; the higher values indicate that the module contains multiple unrelated or only slightly related tasks.

3 Effects of changes on slices

We use an abstract model of a software system to examine how changes affect metric data slices. Using this model, a software system is a set of *slice modules*, where each slice module is a set of metric data slices, and a metric data slice is a set of data tokens. A slice module, M, is a set of metric data slices, since all metric data slices based on particular outputs are unique. Minimally, the data token used to output a value (this value may be a record, array or other aggregate structure) belongs to only one metric data slice. Since each metric data slice is based on one *output*, each slice in slice module M has some data tokens that are not part of any other metric data slice in M.

Modifications to a module affect the metric data slices in the slice module representation of the module. We classify these effects from common changes to the slice module M representation of module m. Our analysis includes only changes involving the addition, movement, deletion, or changes to data definitions or references. The analysis does not evaluate the effects of changes to non-data tokens, i.e., changing a ">" to a "<" in a decision. We assume that the changes are semantically meaningful and non-trivial, and, thus, we do not address changes that cannot affect the output of a program, such as adding or changing comments or adding "dead code", that is, code that cannot be reached during any execution. Our analysis is based on incremental changes to a module. We view major changes as sequences of incremental changes. Incremental changes to module m will make the following incremental changes to the slices in the slice module representation M:

Adding code to m: Adding code to m results in either adding a metric data slice to M or extending an existing metric data slice. Adding an additional output will add a metric data slice. Adding an output will not affect other metric data slices (we assume that adding an output with side effects is a sequence of two incremental changes). Adding code that does not include adding an output must add data tokens to one or more existing metric data slices, thus extending these metric data slices.

Moving code: Moving code cannot add any new metric data slices, since a new slice requires adding a new output which requires the addition of some code. However, moving code can extend one or more metric data slices by moving an output so that additional data tokens are in the slice defined by the output. An output can also be moved so that some data tokens are no longer in the metric data slice defined by the output, resulting in a smaller metric data slice. A token can be moved so that it becomes part of a metric data slice defined by a particular output thus extending that metric data slice. Code can also be moved out of a metric data slice, resulting in a smaller metric data slice. One incremental code movement change can result in a combination of these effects on slices. A change that consists of moving a data token from one location to another in a module can result in extending one or more metric data slices and shortening one or more metric data slices.

Deleting code: Deleting an output results in removing the metric data slice defined by the output from m. Deleting non-output data tokens shortens all of the metric data slices that contain the deleted data token.

Change code: We treat changed code as a sequence of incremental changes — delete, then add data tokens.

Thus, any incremental change to module m results in one or more of the following changes to the metric data slice structure of its slice module representation M:

- Add a metric data slice to M
- Extend one or more metric data slices in M
- Shorten a metric data slice in M
- Remove a metric data slice from M

We now evaluate the effects on the slice-based cohesion metrics from software changes.

4 Effects on slice metrics from adding code

Adding code to a module m results in adding or extending one or more metric data slices to the slice module representation M of module m. We examine how adding or extending a metric data slice affects the slice metrics introduced in Section 2. In the discussion, the changed representation is described as M'. Removing code has the inverse effect of adding code.

4.1 Adding one metric data slice to M

A metric data slice is added to M by adding code which includes one new output data token to m.

Coverage. Adding a metric data slice SL to M can either increase or decrease the *Coverage* depending on the relationship between the size of the new slice and a function of the sizes of the new and old modules to the *Coverage* of the original module. Through relatively simple algebraic transformations, we find that the *Coverage(M') > Coverage(M)* if and only if

$$\frac{|SL|}{|M'| \cdot size(m') \; - \; |M| \cdot size(m)} \; > \; Coverage(M)$$

Overlap. *Overlap* may go up or down, depending on whether the added slice has more overlap than *Overlap(M)*. If the added slice SL includes all of the data tokens in the intersection of the metric data slices in M, $SL_{int}(M)$, then *Overlap(M') > Overlap(M)* if and only if

$$\frac{|SL_{int}(M)|}{|SL|} \; > \; Overlap(M)$$

If the added slice SL does not include all of the data tokens in the intersection of M, $SL_{int}(M)$, then we find that *Overlap(M') > Overlap(M)* if and only if

$$\frac{|SL_{int}(M')|}{|SL|} \; > \; Overlap(M) + \sum_{i=1}^{|M|} \frac{|SL_{int}(M)| - |SL_{int}(M')|}{|SL_i|}$$

Tightness. Adding a slice to M lowers the *Tightness* in the changed module. *Tightness(M') < Tightness(M)* because there are more slices in M' and the number of data tokens common to all slices cannot increase when a new slice is added. (The numerator cannot increase, while the denominator must increase.)

Parallelism. Here, we assume a τ of 0, thus, parallel slices can have no common data tokens. *Parallelism* can increase or decrease depending on whether the new code intersects with old code. *Parallelism* increases only if the added slice is completely independent from all of the slices in M. *Parallelism* will decrease if the added slice includes data tokens that are on any of the slices in M that do not share any data tokens with other slices in M.

MinCoverage. *MinCoverage(M') < MinCoverage(M).* If $|SL| >$ size of the shortest slice in M, *MinCoverage* decreases since the *size(m') > size(m).* If $|SL| <$ the shortest slice in M, then *MinCoverage* decreases even more.

MaxCoverage. If $|SL| <$ size of the longest slice in M, *MaxCoverage(M') < MaxCoverage(M)* since the *size(m') > size(m).* If $|SL| >$ size of the longest slice then it depends on the number of new data tokens added to M.

4.2 Extending one or more metric data slice(s) in M

Adding one additional data token to m can result in extending one or more metric data slices.

Coverage. If the change extends one metric data slice SL in M, *Coverage* will normally decrease, that is, *Coverage(M') < Coverage(M). Coverage(M') = Coverage(M)* if all of the metric data slices in M are independent, that is, they have maximum *Parallelism*. The only situation in which it is possible for *Coverage* to increase when adding one metric data slice, is when there are data tokens in m which are not in any metric data slice, that is, when there is dead code in module m.

If more than one metric data slice is extended, then *Coverage* is more likely to increase. Assuming that we extend t metric data slices, each by one data token, then *Coverage(M') > Coverage(M)* if and only if

$$Coverage(M) < \frac{t}{|M|}$$

Overlap. Assume that $|M| > 1$ and that at least one slice $SL \in M$ is not changed. In this case, *Overlap(M') < Overlap(M)* because the size of all of the changed slices has increased, while the data tokens common to all slices is unchanged.

If the added code results in adding a data token to each metric data slice in M, then *Overlap(M') > Overlap(M)* since both the number of data tokens common to all metric data slices and the size of all metric data slices is increased by one.

Tightness. Assume that $|M| > 1$ and that at least one slice $SL \in M$ is not changed. In this case, *Tightness(M') < Tightness(M)* because the *size(m') > size(m)* while the data tokens common to all slices is unchanged.

If the added code results in adding a data token to each metric data slice in M, then *Tightness(M') > Tightness(M)*, since both the numerator and denominator of the *Tightness* calculation is increased by one.

Parallelism. Again, we assume a τ of 0, thus, parallel metric data slices can have no common data tokens. *Parallelism* will either remain unchanged or be reduced, *Parallelism(M') ≤ Parallelism(M).* *Parallelism(M') = Parallelism(M)* if the change extends only one metric data slice. If an added data token extends two or more slices that shared no tokens in M, then *Parallelism(M') < Parallelism(M).*

MinCoverage. If the added code extends the shortest metric data slice in M then, *MinCoverage(M') > MinCoverage(M).* Otherwise, *MinCoverage(M') < MinCoverage(M)* since *size(m') > size(m)* and the size of the shortest metric data slice is unchanged.

MaxCoverage. If the added code extends the longest metric data slice in M then, *MaxCoverage(M') > MaxCoverage(M).* Otherwise, *MaxCoverage(M') < MaxCoverage(M)* since *size(m') > size(m)* and the size of the longest metric data slice is unchanged.

5 Effects on slice metrics from moving code

The movement of one data token within a module can result in no significant change in all of the metric data slices. However, such a change may extend and/or shorten one or more metric data slices (see Section 3). Thus, it is very difficult to identify a priori the effects on the metrics of code movement changes. We examine the effect of moving one data token on the data slice metrics.

Coverage. Moving a code segment containing one data token can raise the *Coverage* if the net effect is to increase the size of more data slices than are reduced. *Coverage* is reduced if the net effect is to shorten the metric data slices.

Overlap and Tightness. The change will increase *Overlap* and *Tightness* if a data token that is not in the intersection of the metric data slices is moved so that all of the metric data slices are extended. *Overlap* and *Tightness* are reduced if the change results in a smaller intersection.

Parallelism. *Parallelism* will either remain unchanged, be increased or reduced. The effect on *Parallelism* depends on whether code is moved in a manner that affects the number of data tokens that metric data slices share with other slices.

MinCoverage and MaxCoverage. The effect of the change on *MinCoverage* and *MaxCoverage* depends on whether the shortest or longest metric data slice respectively is increased or decreased.

6 Conclusions

In this paper, we develop a slice model of programs and redefine slice metrics based on our model. Previous work demonstrates the relationship between slice metrics and module cohesion. Using our new model we can analyze how changes to a program module are reflected in the slice based metrics, and we get some additional understanding of the relationship between these metrics and cohesion.

Table 1 summarizes the effects of adding code on each of the slice metrics. We find that each of the metrics exhibits unique behavior, yet the metrics' response to adding code matches the intuitive meaning of separate attributes of module cohesion. For example, when adding a slice, *Coverage* will increase or decrease depending on how cohesive (in terms of *Coverage*) the new slice is when compared to the cohesiveness (also in terms of *Coverage*) of the original code. The effects on *Overlap* that result from adding a slice are similar to the effects on *Coverage*, which is expected due to the similarities of their definitions. The magnitude of the effects on *Tightness* of adding a slice also depends on the relationship of the new slice to the existing code. However, *Tightness* always decreases when a new slice is added. *Tightness* is a metric that is especially sensitive to the number of outputs computed by a module (each slice represents one output). The number of outputs, and their interdependence, is an attribute of cohesion that is indicated by the *Tightness* metric. *Coverage, Overlap,* and *Tightness* all tend to decrease when one or more slices are extended. Unless all slices are extended by one code addition, a slice extension change will tend to add functionality that is not connected to the other slices, thus decreasing attributes of cohesion. The *Parallelism* metric is essentially an "anti-cohesion" metric. *Parallelism* is high when slices are independent. Thus, adding a slice is likely to reduce *Parallelism* if the new slice shares any data tokens with existing slices. After a code extension change is made, if more than one slice is affected, then the sharing of data tokens reduces *Parallelism*. The effects on *MinCoverage* or *MaxCoverage* depend, as expected, on whether the longest or shortest slice is modified.

We find that determining general effects on the cohesion metrics resulting from moving code is a difficult problem. Moving code can affect an arbitrary number of slices. Our analysis of this problem provides support for maintenance programmers who would rather add new code than move a line of an existing system. The metrics should be especially helpful when a code movement change is contemplated, since our analysis shows that the effects on cohesion from moving code is very unpredictable.

The effects on the cohesion metrics from module changes do seem to match our intuition concerning the expected effects on particular cohesion attributes. Thus, effects on cohesion attributes can be monitored during maintenance using these metrics. Such cohesion monitoring should aid in managing the effects of maintenance activities.

We plan to continue to analytically evaluate the relationship between cohesion attributes and the cohesion metrics. This analysis is necessary to demonstrate that the metrics impose an ordering on modules and systems that matches our intuitive understanding of cohesion. We also need to more fully understand the properties of the metrics to insure that we perform valid statistical analyses on metric data.

We also plan empirical studies to confirm that the cohesion metrics can be useful maintenance tools. We have prototype cohesion measurement tools that can analyze cohesion in Pascal programs. Analyzers for C, C++ and Ada programs are planned. With these tools, data from industry, and input from software maintenance professionals, we hope to demonstrate the effectiveness of the slice based cohesion approach.

Acknowledgements

The authors thank the Department of Computer Science at Colorado State University for providing Dr. Ott with the facilities and environment that resulted in a productive sabbatical year including the completion of this paper. The authors also thank the anonymous referee who provided valuable criticisms of an earlier version of this paper.

References

[1] A.L. Baker, J.M. Bieman, N. E. Fenton, A. C. Melton, and R.W. Whitty. A philosophy for software measurement. *Journal of Systems and Software*, 12(3):277–281, July 1990.

[2] J. Bieman, A. Baker, P. Clites, D. Gustafson, and A. Melton. A standard representation of imperative language programs for data collection and software measures specification. *The Journal of Systems and Software*, 8(1):13–37, January 1988.

[3] J. Bieman and J. Schultz. Estimating the number of test cases required to satisfy the all-du-paths testing criterion. *Proc. Software Testing, Analysis and Verification Symposium (TAV3-SIGSOFT89)*, pages 179–186, December 1989.

[4] J. Bieman and J. Schultz. An empirical evaluation (and specification) of the all-du-paths testing criterion. *Software Engineering Journal*, 7(1):43–51, January 1992.

Table 1: Summary of Effects of Adding Code on Slice Metrics

Metric	Adding a New Slice	Extending One or More Slices
Coverage	Depends on *Coverage* of new slice	Decreases under normal circumstances
Overlap	Depends on *Overlap* of new slice	Decreases unless all slices are extended
Tightness	Decreases	Decreases unless all slices are extended
Parallelism	Decreases unless new slice is independent	One slice extended, then no change Otherwise, decreases
MinCoverage	Depends on size of new slice	Depends on slice extended
MaxCoverage	Depends on size of new slice	Depends on slice extended

[5] J.-D. Choi, B. Miller, and P. Netzer. Techniques for debugging parallel programs. Technical Report 786, Univ. Wisconsin-Madison, 1988.

[6] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Software Engineering*, 17(8):751–761, 1991.

[7] Matthew S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, 1977.

[8] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Trans. Programming Languages and Systems*, 11(3):345–386, 1989.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Programming Languages and Systems*, 12(1):35–46, 1990.

[10] B. Korel and J. W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[11] B. Korel and J. W. Laski. Stad – a systerm for testing and debugging: User perspective. In *Proc. 2nd Workshop on Software Testing, Verification and Analysis*, 1988.

[12] Arun Lakhotia. Insights into relationships between end-slices. Technical Report CACS TR-91-5-3, University of Southwestern Louisiana, September 1991.

[13] Arun Lakhotia and Jagadeesh Nandigam. Computing module cohesion. Technical Report CACS TR-91-5-4, University of Southwestern Louisiana, November 1991.

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

[15] H. D. Longworth, L. M. Ottenstein [Ott], and M. R. Smith. The relationship between program complexity and slice complexity during debugging tasks. In *Proc. IEEE COMPSAC*, pages 383–389, 1986.

[16] A.C. Melton, D.A. Gustafson, J.M. Bieman, and A.L. Baker. A mathematical perspective for software measures research. *Software Engineering Journal*, 5(5):246–254, 1990.

[17] Linda M. Ott. Using slice profiles and metrics during software maintenance. In *Proc. 10th Annual Software Reliability Symposium*, pages 16–23, 1992.

[18] Linda M. Ott and Jeffrey J. Thuss. The relationship between slices and module cohesion. In *Proc. 11th International Conference on Software Engineering*, pages 198–204, 1989.

[19] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. Technical Report CS-91-4, Dept. Computer Science, Michigan Technological Univ., November 1991. Also published as Technical Report CS-91-124 Dept. Computer Science, Colorado State Univ.

[20] Linda M. Ott and Jeffrey J. Thuss. Using slice profiles and metrics as tools in the production of reliable software. Technical Report CS-92-8, Dept. Computer Science, Michigan Technological Univ., April 1992. Also published as Technical Report CS-92-115 Dept. Computer Science, Colorado State Univ.

[21] K. J. Ottenstein and L. M. Ottenstein [Ott]. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, 1984. See also SIGPLAN Notices, 19,5, 177-184.

[22] T. Reps and W. Yang. The semantics of program slicing and program integration. In *Proc. of the Colloquim on Current Issues in Programming Languages*, pages 360–374, 1989. *Lecture Notes in Computer Science*, Vol. 352, Springer-Verlag, New York, NY.

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

[24] M. D. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[25] M. D. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[26] M. D. Weiser. Program slicing. *IEEE Trans. Software Engineering*, 10(4):352–357, 1984.

[27] L. Wilson and L. Leelasena. The QUALMS program documentation. Technical Report Alvey Project SE/69, SBP/102, South Bank Polytechnic, London, 1988.

[28] H. Zuse. *Software Complexity Measures and Methods.* W. de Gruyter, Berlin, 1991.