Measuring Functional Cohesion

James M. Bieman and Linda M. Ott

Abstract— **We examine the functional cohesion of procedures using a data slice abstraction. Our analysis identifies the data tokens that lie on more than one slice as the "glue" that binds separate components together. Cohesion is measured in terms of the relative number of *glue tokens*, tokens that lie on more than one data slice, and *super-glue tokens*, tokens that lie on all data slices in a procedure, and the *adhesiveness* of the tokens. The intuition and measurement scale factors are demonstrated through a set of abstract transformations.**

Index Terms— **Software measurement, cohesion, program slices, measurement theory**

I. Introduction

COHESION is an attribute of a software unit or module that refers to the "relatedness" of module components. A highly cohesive software module is a module that has one basic function and is indivisible — it is difficult to split a cohesive module into separate components.

Virtually every software engineering text describes cohesion as an important factor of design quality. If cohesion is an important attribute of software design quality, we should be able to recognize when a module exhibits cohesion, and, ideally, we should be able to quantify the amount of cohesion in a module. Such cohesion measures can help developers design modules with greater cohesion.

Module cohesion can be classified using an ordinal scale that includes *coincidental*, *logical*, *temporal*, *procedural*, *communicational*, *sequential*, and *functional* cohesion [39, ch. 7]. Using this model, a module exhibits one of these seven cohesion categories. The cohesion categories vary in desirability ranging from the most desirable (functional cohesion) to the least desirable (coincidental cohesion). All of these cohesion categories indicate the extent of the "functional strength" of a module, the contribution of module parts towards performing one task [10, pp. 199–200].

Our aim is to develop quantitative measures of functional cohesion, the most desirable of these functional strength cohesion categories. According to Yourdon and Constantine, every element in a module exhibiting functional cohesion "is an integral part of, and is essential to, the performance of a single function" [39, p. 127]. In their model, a module is either functionally cohesive or not. In contrast, we are developing techniques that indicate the extent to which a module approaches the ideal of functional cohesion.

Note that one can also evaluate cohesion from the perspective of data abstraction [23, pp. 169–171]. Fenton describes this *abstract* or *data* cohesion as a different notion of cohesion with a different set of measurement attributes [10, p. 200]. In this paper, we address functional cohesion; we defer the treatment of abstract or data cohesion to future work.

Measurement techniques used in the physical sciences guide us in our development of functional cohesion measures. Aspects of functional cohesion are internal product attributes related to properties of programs [11]. Our objectives include the development of (1) a good model of functional cohesion, and (2) measures that use the model to quantify functional cohesion.

For cohesion measures 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 such tools should be based on the models. For example, QUALMS [38] is based on the flow graph model, and the test coverage measurement tools of Bieman and Schultz [4], [5] are based on the *standard representation* model [2]. We use a *slice abstraction* of a program based on *data slices* to model cohesion [26].

A program slice is the portion of program text that affects a specified program variable [35]. A variation on program slices can model and measure functional cohesion [28]. Procedure cohesion measures must indicate the cohesion that is expressed in the program text. We cannot measure semantic relations between program components that cannot be identified from the program text alone. Note that functional cohesion is actually an attribute of individual procedures or functions, rather than an attribute of a separately compilable program unit or module (depending on the programming language, modules may include several procedures and declarations). We will use the term "procedure" to refer to both procedures and functions.

We develop cohesion measures in terms of the slice model, and validate the measures by demonstrating that they are consistent with expected cohesion model orderings and determining their scale properties. Thus, we appeal to the *representation condition* of measurement theory [10, pp. 25–26],[11], which requires that our intuition about the relative quantity of functional cohesion is preserved by a

Travel support for this work provided in part by a Faculty Development Grant from Michigan Technological University. J. Bieman's research is partially supported by the NASA Langley Research Center, Colorado Advanced Software Institute (CASI), Computer Technology Associates (CTA) and Storage Technology Inc. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology teaching and research at universities in Colorado for the purpose of economic development.

J. Bieman is with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523. Email: bieman@cs.colostate.edu, (303)491-7096, Fax: (303) 491-2466.

L. Ott is with the Department of Computer Science, Michigan Technological University, Houghton, MI 49931, Email: linda@cs.mtu.edu, (906) 487-2187, Fax: (906) 487-2933.

cohesion measure. To be measurable on an ordinal scale, an attribute of cohesion must impart an ordering on the model. That is, the model of a procedure with "more" of one cohesion attribute must be ranked (according to the attribute ordering) higher than the model of a procedure with "less" of the attribute [24].

A measure is specified as a mapping from the model to a quantitative value. Such a measure must be consistent with the cohesion ordering. One way to demonstrate that a measure is consistent with the ordering is to evaluate the effect of code modifications to the model and the measures. We focus on the direction of the changes to cohesion measurements resulting from relatively simple code modifications. The direction of measurement changes provides a ranking of relative levels of cohesion before and after a code change. Our analysis also demonstrates the scale properties and the arithmetic operations that can be applied to the measurement values [41, ch. 4].

The role of experimentation in software measurement research is to map structural measures back to process goals such as fewer defects, increased maintainability, etc. But, before we can conduct effective empirical research, we must first have sound measures [1]. Thus, our goal here is to develop measures that accurately reflect the concept of cohesion.

The paper has the following organization. In Section II, we define the abstractions used to model functional cohesion. In Section III, we examine the cohesion attributes and measures, and Section IV evaluates the scale properties of the measures. In Section V, we provide some examples of procedures, cohesion orderings, and cohesion measures. Section VI is a review of related work. Our conclusions are given in Section VII.

II. Cohesion Abstractions

In our analysis, functional cohesion is based on procedure outputs. Each output "object" (output parameter, modified global variable, or file), represents one component of a procedure's functionality. We identify the components of a procedure that contribute to particular outputs. Although a procedure may perform computation that does not produce outputs, outputs of some kind are generally the externally visible manifestation of functionality. We do not address the cases where activities that do not produce outputs are the real functionality, for example modules whose main functionality is to produce a time delay. In the case of procedures with multiple outputs, we see how closely the program parts that contribute to different outputs are bound. Using this approach, procedures with only one output exhibit maximum functional cohesion.

A. Program Slices

Slicing is a method of program reduction introduced by Weiser [35], [36], [37]. A *slice* of a procedure at statement s with respect to variable v is the sequence 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 [7], maintenance [13], [15], [25], and testing [17], [18], [22], [29]).

Weiser's algorithm for computing slices is based on data flow analysis. It is suggested in [27] that a *program dependence graph* representation can 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 [16], [31]. 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* [16] 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.

We derive cohesion measures directly from slices rather than dependence graphs. Slices promote a more intuitive analysis since they are based on program text. Our measurement theory approach requires that a measure be consistent with intuition, and including program text in our abstraction eases intuitive analysis.

B. Data Slices

In [37], Weiser defined several slice based measures. Longworth [21] first studied their use as indicators of cohesion. In [30] and [33], Thuss eliminates certain inconsistencies noted by Longworth through the use of *metric slices*. A metric slice takes into account both *uses* and *used by* data relationships; 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 measures, 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 *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 procedure. We consider a change of interest to be any change which could have an effect on the cohesiveness of a procedure. 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 the variable used in a given context. Each of these changes would result in a change to at least one data slice. (This is in contrast to a metric slice, where if a statement is modified, the actual statements in the slice might not change.)

Informally, we view a *data slice* for a data token, v, as the sequence 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 compute a data slice for each output of a procedure. An "output" is any single value explicitly output to a file

```
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;
```

Fig. 1. Data slice for *SumN*. Items included in the slice are contained within boxes.

SumN	ProdN	Statement
		procedure SumAndProduct
1	1	(N : integer;
1	1	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;

Fig. 2. Data Slice profile for *SumAndProduct*. The number of data tokens included in the data slice for *SumN* and *ProdN* is indicated in columns 1 and 2 respectively.

(or user output), an output parameter, or an assignment to a global variable. An output tuple with multiple components is considered to be multiple outputs. Since we are interested in the cohesion of the whole procedure, we use a concept similar to that of *end-slices* [19]. The "backward" slices are computed from the end of the procedure[1] and the "forward" slices are computed from the "top"s of the backward slices.

Fig. 1 displays an example of a data slice embedded in a program. The slice for *SumN* in Fig. 1 is a sequence of data tokens:

$$N_1 \cdot SumN_1 \cdot I_1 \cdot SumN_2 \cdot 0_1 \cdot I_2 \cdot 1_2 \cdot N_2 \cdot SumN_3 \cdot SumN_4 \cdot I_3$$

where each T_i indicates the i'th data token for T in the procedure. Note that in the slice for *SumN*, the subscript in "1_2" indicates that the token is the second occurrence of data token "*1*" in the procedure. We can also compute the slice for *ProdN*:

$$N_1 \cdot ProdN_1 \cdot I_1 \cdot ProdN_2 \cdot 1_1 \cdot I_2 \cdot 1_2 \cdot N_2 \cdot ProdN_3 \cdot ProdN_4 \cdot I_4$$

We can profile the data slices in a procedure to give a sense of the relationships among data slices. Fig. 2 shows an example of a data slice profile. We indicate, in the column for a slice variable, the number of data tokens in that line that are included in the slice. This profile was derived from an earlier method developed for visualizing slices [25], [28], [33].

C. Slice Abstractions

Our analysis of functional cohesion is developed using an abstract model of procedures based on data slices. The *Slice Abstraction* models each procedure as a set of data slices, and a data slice as a sequence of data tokens. Essentially, we strip away all of the non-data tokens from a procedure and include only the data tokens in the abstraction.

The slice abstraction for the *SumAndProduct* procedure of Fig. 1 and Fig. 2 is:

$$SA(SumAndProduct) = \{N_1 \cdot SumN_1 \cdot I_1 \cdot SumN_2 \cdot 0_1 \cdot I_2 \cdot 1_2 \cdot N_2 \cdot SumN_3 \cdot SumN_4 \cdot I_3,$$
$$N_1 \cdot ProdN_1 \cdot I_1 \cdot ProdN_2 \cdot 1_1 \cdot I_2 \cdot 1_2 \cdot N_2 \cdot ProdN_3 \cdot ProdN_4 \cdot I_4 \ \}$$

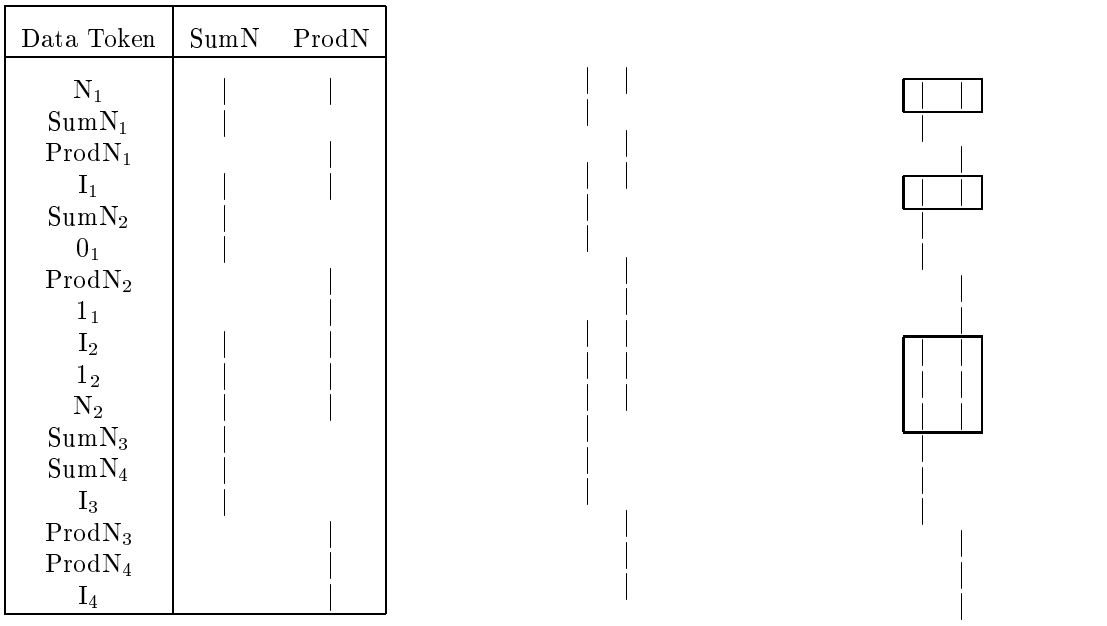
Fig. 3(a) provides another view of a slice abstraction of the *SumAndProduct* procedure. The names of the data tokens are listed in the first column of Fig. 3(a). A "|" in the second and third column indicates that the indicated data token is part of the data slice for the named output.

We find an uncluttered view of slice abstractions without labels useful for visualizing important attributes of functional cohesion in slice abstractions. Fig. 3(b) is an unlabeled view of the slice abstraction of the *SumAndProduct* procedure. When analyzing functional cohesion, it is important to know when one token is in more than one data slice, but the actual names of the tokens are not important. The slice abstractions from two completely different procedures can have the same cohesion properties, and look identical when viewed in the unlabeled form.

D. Glue, Super-glue, and Stickiness

As Fig. 3(a) and Fig. 3(b) show, several of the data tokens are common to more than one data slice. Data tokens N_1, I_1, I_2, 1_2, and N_2 are in the data slice for *SumN* and the data slice for *ProdN*. Such tokens, common to more than one data slice in a slice abstraction, are the connections between the slices. We say that these tokens are

[1]We use the *FinalUse* nodes of [16] as the end of a procedure.



(a) *SA(SumAndProduct)* (b) Unlabeled View (c) *Glue* tokens highlighted.

Fig. 3. Three Views of *SA(SumAndProduct)*

the "glue" that binds the slices. Thus, we define the *glue* in a slice abstraction of a procedure P, *G(SA(P))*, as the set of data tokens that lie on more than one data slice in *SA(P)*. A *glue token* is a token that lies on more than one data slice. We also consider all of the tokens in an abstraction with only one slice to be glue tokens. Fig. 3(c) shows *SA(SumAndProduct)* with the glue tokens enclosed in boxes. Although there are two "|" symbols on each row of glue tokens in Fig. 3(c), there is actually only one token for each row.

It is useful to identify the data tokens that are common to every data slice in a procedure. These tokens are the *super-glue* tokens, and *SG(SA(P))* denotes the set of data tokens that lie on **all** data slices in *SA(P)*. The notion of super-glue tokens is especially useful in slice abstractions with more than two data slices. Note that $SG(SA(P)) \subseteq G(SA(P))$ — all super-glue tokens are also glue tokens. If $|SA(P)| \leq 2$ then $SG(SA(P))=G(SA(P))$. Note that all of the data tokens in a procedure with only one slice are super-glue tokens.

Fig. 4 shows a 3-slice abstraction with glue and superglue tokens. This abstraction has two super-glue tokens and five glue tokens (super-glue is still glue). One of the tokens glues S_1 to S_2, one glues S_2 to S_3, and one glues S_1 to S_3. The super-glue tokens bind all three slices together. Six of the tokens lie on only one data slice and are not glue tokens.

The distribution of glue and super-glue tokens indicates how tightly bound the individual slices are, since the effect of glue tokens is to bind slices. Individual glue tokens can have a varying effect on cohesion based on the number of slices that they bind. Thus, we can describe the relative *stickiness* or *adhesiveness* of a glue token. The notion of

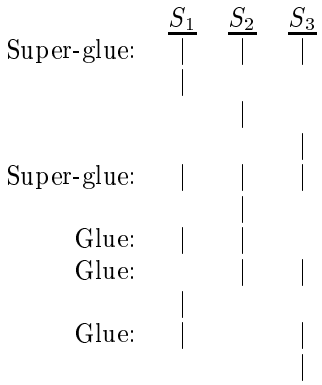


Fig. 4. A 3-slice *SA* with *glue* and *super-glue*.

token adhesiveness can characterize the adhesiveness property of an entire procedure or slice abstraction. We use the concepts of glue, super-glue, and adhesiveness to develop functional cohesion measures.

III. Functional Cohesion Attributes and Measures

A. Definition of Measures

We define functional cohesion attributes and measures in terms of slice abstractions, data tokens, glue and super-glue. We also use the set of data tokens in a slice abstraction a, denoted *tokens(a)*, and the set of data tokens in procedure p, denoted *tokens(p)*. In general, *tokens(p)* = *tokens(SA(p))*. However, if a value is computed that does not contribute to any output (usually a program anomaly), then there may be data tokens that do not lie on any slice and *tokens(SA(p))* $\subset$ *tokens(p)*. Note that each appear-

ance of a data token in a program is counted as a different token, and each token can be in more than one data slice.

Metrics based on the relative number of glue and super-glue tokens are intuitive and can easily be defined in terms of slice abstractions. According to Yourdon and Constantine [39, pp. 127–130], a procedure with functional cohesion is one in which all parts are cohesive. This view recognizes only the strongest functional cohesion and is consistent with the use of the super-glue tokens as the basis for defining cohesion attributes and measures. Thus, we define *strong functional cohesion (SFC)* as the ratio of super-glue tokens to the total number of data tokens in a procedure p:

$$SFC(p) = \frac{|SG(SA(p))|}{|tokens(p)|} \tag{1}$$

The *SFC* is a measure of the minimal functional cohesion in a procedure. SFC is very similar to the *Tightness* measure defined by Ott and Thuss [30]. However *Tightness* is defined in terms of statements shared by slices rather than data tokens.

We can also measure cohesion in terms of the glue tokens in a slice abstraction. Such a measure can be more sensitive than a measure based on only the super-glue tokens — it can indicate that adding something may "glue" together previously non-cohesive elements even if the token does not "glue" together **all** of the slices. Such functional cohesion indicates a "weaker" type of cohesion than indicated by the super-glue tokens. Thus we define *weak functional cohesion (WFC)* as the ratio of glue tokens to the total number of tokens in a procedure. For procedure p:

$$WFC(p) = \frac{|G(SA(p))|}{|tokens(p)|} \tag{2}$$

Another way to measure cohesion is in terms of the *adhesiveness* of glue tokens. The *adhesiveness* is related to the relative number of slices that each token "glues" together. Thus, a token that "glues" together four slices in a five slice procedure is more adhesive than a token that "glues" together two or three slices. We can define the adhesiveness, α, of token t in procedure p as follows:

$$\alpha(t,p) = \begin{cases} \dfrac{\#\ slices\ in\ p\ containing\ t}{|SA(p)|} & if\ t \in G(SA(p)) \\ 0 & otherwise \end{cases} \tag{3}$$

The overall adhesiveness, A, of an SA is the average adhesiveness of the data tokens in a procedure:

$$A(p) = \frac{\sum_{t \in tokens(p)} \alpha(t,p)}{|tokens(p)|} \tag{4}$$

Equivalently, overall adhesiveness can be computed as a ratio of the amount of adhesiveness to the total possible adhesiveness. That is, for procedure p:

$$A(p) = \frac{\sum_{t \in G(SA(p))} \#\ slices\ containing\ t}{|tokens(p)| \cdot |SA(p)|} \tag{5}$$

In the examples in the following subsection, we compute A using equation (5), since equation (5) is easier to apply.

Adhesiveness should indicate the relative strength of the glue in a procedure. Adhesiveness is most closely related to the *coverage* measure of Ott and Thuss [30]. It should be particularly sensitive to the cohesion resulting from glue tokens that lie on more than two slices, but do not lie on all slices.

All of these cohesion measures (*strong functional cohesion*, *weak functional cohesion*, and *adhesiveness*) range in value from zero to one. They have a value of zero when a procedure has more than one output and exhibits none of the cohesion attribute indicated by a particular measure. A procedure with no super-glue tokens, no tokens that are common to all data slices, has zero *strong functional cohesion* — there are no data tokens that contribute to all outputs. A procedure with no glue tokens, that is no tokens common to more than one data slice (in procedures with more than one data slice), exhibits zero *weak functional cohesion* and zero *adhesiveness* — there are no data tokens that contribute to more than one output. The *strong functional cohesion* and *adhesiveness* are at a maximum value of one for procedures in which all of the data tokens are super-glue tokens — all data tokens affect all outputs. *Weak functional cohesion* of a procedure is one if all data tokens are glue tokens — all data tokens affect more than one output in procedures with more than one slice.

B. Examples

The cohesion measures can be applied to the *SumAndProduct* procedure. *SA(SumAndProduct)* has two slices with 17 tokens and 5 glue tokens. Each glue token is a super-glue token since *SA(SumAndProduct)* has only two data slices. Thus,

$$WFC(SA(SumAndProduct)) = \mathrm{SFC}(\mathrm{SA}(\mathrm{SumAndProduct})) = \frac{5}{17} = .294$$

Adhesiveness is calculated as follows:

$$A(SA(SumAndProduct)) = \frac{5 \cdot 2}{17 \cdot 2} = .294$$

since there are five glue tokens and each glue token lies on two slices. The denominator is the total number of tokens times the number of slices. We see that in this two slice example procedure all three cohesion measures give the same value. This is not surprising since the *WFC* and *A* measures gain sensitivity on multi-slice procedures — all glue tokens are also super-glue tokens on a one or two slice procedure.

The *WFC* and *SFC* of the 3-slice abstraction in Fig. 4 will differ since some of the glue tokens are not super-glue. Out of a total of 11 tokens, this abstraction has 5 glue tokens of which 2 are super-glue. Thus

$$WFC(SA(Fig.\ 4)) = 5/11 = .455$$

and

$$SFC(SA(Fig.\ 4)) = 2/11 = .182$$

Since there are two tokens on three slices and three tokens on two slices, adhesiveness is calculated as follows:

$$A(SA(Fig.\ 4)) = \frac{2 \cdot 3 + 3 \cdot 2}{11 \cdot 3} = \frac{12}{33} = .36$$

Adhesiveness and the strong and weak cohesion measures are based solely on the number of slices and data tokens in a procedure, and the number of glue and super-glue tokens.

C. Relationships between the Measures

By examining the definitions, we can determine relationships among the three proposed measures. Since $SG(SA(P)) \subseteq G(SA(P))$, it follows that $|SG(SA(p))| \leq |G(SA(P))|$. Thus, using (1) and (2) we can see that for a given procedure p:

$$SFC(p) \leq WFC(p). \tag{6}$$

We see that:

$$SFC(p) \leq A(p) \tag{7}$$

by noticing that $\alpha(t,p) = 1$ using Definition (3) for all $t \in SG(SA(p))$ and therefore, the numerator in (4) is at least as large as the numerator in (1). Similarly, since $\alpha(t,p) \leq 1$ for all $t \in G(SA(P))$, using (2) and (4), we see that:

$$A(p) \leq WFC(p) \tag{8}$$

Thus, we have:

$$SFC(p) \leq A(p) \leq WFC(p) \tag{9}$$

Finally, we see that $A(p)$ is more "sensitive" than either $WFC(p)$ or $SFC(p)$ to differences in the amount of program cohesion. If we fix the size of programs considered, that is, $|tokens(p)|$, and we fix the number of slices considered, that is, $|SA(p)|$, we see that $WFC(p)$ and $SFC(p)$ can assume at most $|tokens(p)|$ values. $A(p)$, on the other hand, can assume $|tokens(p)| \cdot (|SA(p)| - 1)$ values.

IV. Discussion of Scale Properties

Fenton defines the term "validation" as "the process of ensuring that the measure is a proper numerical characterization of the claimed attribute" [10, p. 82]. This kind of validation is very difficult when the attribute to be measured is loosely understood. We need to rely on human intuition to determine the relative levels of our cohesion properties, to see if they are consistent with the measurement values. Zuse shows how to determine what type of scale software measures assume [41, ch. 4], [42], [6]. In this paper, we combine the methods of Fenton and Zuse to validate the cohesion measures in terms of intuitive notions of cohesion and to determine the scale properties of the measures. First, we show that the measures assume an ordinal scale that matches our intuition concerning the cohesion attributes that are measured. Then, we evaluate the measures in terms of the requirements of a ratio scale.

A. Cohesion Measures and the Ordinal Scale

For a real-valued ordinal scale measure of cohesion attributes to exist, our intuition about these attributes, called "empirical relations" or "viewpoints", must satisfy three axioms: reflexivity, transitivity, and completeness [40], [41, p. 47], [42], [6]. These are the requirements of a *weak order*. From [40], we define a cohesion *viewpoint* as binary relations, $\star>$, $\star\approx$, and $\star\geq$ on programs $\mathcal{P}$ where:

$P_1 \star> P_2$	P_1 is more "cohesive" than P_2
$P_1 \star\approx P_2$	P_1 and P_2 are equally "cohesive"
$P_1 \star\geq P_2$	$P_1 \star> P_2$ or $P_1 \star\approx P_2$

for $P_1, P_2 \in \mathcal{P}$.

It is not possible to give a general definition of cohesion viewpoints. Rather we can use a subset of the above relations called an *elementary viewpoint*. An elementary viewpoint is defined in terms of a finite set of transformations on a program representation. A complete set of *elementary transformations* can be used to generate every possible instance of a program representation from a base representation. To show that a measure is on an ordinal scale, we need to show that it is consistent with a complete set of elementary transformations, since the set represents the cohesion viewpoint. Thus, we evaluate the "functional cohesion orderings" of procedures in terms of intuitively obvious effects of program modifications on functional cohesion. We model the changes in terms of an ordering of slice abstractions. In this analysis, we assume that it is the "shape" of slice abstractions that is critical, so two completely different procedures can have the same functional cohesion attributes. We use unlabeled views of slice abstractions as depicted in Fig. 3(b) to demonstrate the necessary attributes and transformations.

A.1 Slice Abstraction Transformations

Functional cohesion orderings can be developed in terms of a set of elementary transformations of slice abstractions. We seek a set of transformations that can generate the set of all slice abstractions, and provide an ordering. The transformations are developed inductively.

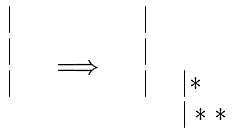
Base case: A one slice procedure:

A one slice procedure is entirely cohesive, and should have the highest possible *SFC*, *WFC*, and *A*. All three of our measures satisfy intuition here. *SFC*, *WFC*, and *A* give their maximum value of 1 for a one slice procedure.

Transformations:

1. Add one slice. There are two ways to add a slice:
   (a) Add functionality by adding a new output to the program. This requires adding at least one new

output token.[2]



The new output is not on any of the previous slices. Thus at least one new non-glue token is added.

(b) Output existing functionality. This can be accomplished by changing a non-output token into an output token. The following change to C-like pseudo-code is an example of such a transformation:

```
y=x ⟹ printf(y=x)
```

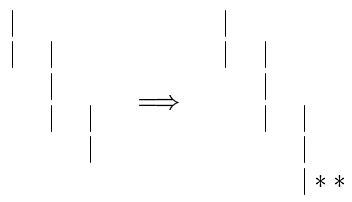
A simple change to the parameters in a Pascal program can also cause existing functionality to become a new output:

```
x:integer ⟹ var x:integer
```

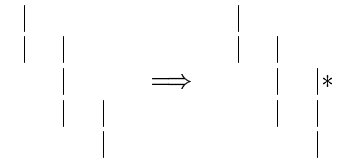
With such transformations, a new slice can be created without adding any new tokens.

2. Extend n slices by adding one token to them. This added token may be a token that is either

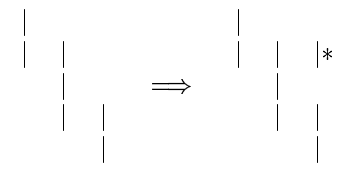
(a) not in any of the slices in the slice abstraction (i.e., a new token):



(b) a token already in one or more of the other slices in the slice abstraction, but not in all of the other slices:



(c) or, a token already in all of the other slices:



A token can be added to a slice without adding new code by moving the token within a procedure to a location that puts it in the scope of the slice.

[2]We use "$*$" to indicate a token added to a slice that is not new to the procedure. We use "$**$" to indicate when an added token is new to the program; it is a token that is not on any other slice.

This set of transformations is complete — we can build all slice abstractions using the base case and repetitions of the two transformations. Removing and shortening slices are inverse operations to the add and extend operations.

A.2 Effect of Transformations on the Metrics

In this section, we summarize the effects of the transformations introduced above on the cohesion measures that we have defined. See the appendix for the detailed arguments. For consistency with the appendix, we will refer to the initial abstraction as a and the abstraction after a transformation as a'.

Strong Functional Cohesion. When adding a new slice to a, $SFC(a') \leq SFC(a)$. This is consistent with our intuition that adding functionality tends to decrease the cohesiveness of a procedure. When extending slices, we find that $SFC(a') > SFC(a)$ only when the number of super-glue tokens has increased. Thus, the effects of the transformations match our intuition that the strong functional cohesion components include only elements that contribute to all the functionality computed by the procedure.

Weak Functional Cohesion. When we add functionality to a procedure by adding a new output, we increase cohesion only when the net effect is to "glue" previously non-cohesive parts creating a higher percentage of glue tokens. When we output existing functionality without adding new tokens, $WFC(a') \geq WFC(a)$. When extending a slice, WFC can remain unchanged, increase or decrease, depending on whether the new token is already "glue", is new "glue" or is not "glue", respectively.

Adhesiveness. When we add functionality to a procedure by adding a new output, A can increase or decrease. If we add only non-glue tokens, then A will decrease. If we add at least some glue tokens, the effect on A depends upon the amount of "glue" added, the size of the procedure and the size of the slice being added. When we extend a slice of a multiple slice procedure, A will increase if we add a superglue token and will decrease if we add a non-glue token. If we add a glue token (which is not also superglue), the effect on A depends upon the ratio of the number of slices that the new token lies on, and the total number of slices in the abstraction. If we extend a slice without adding any tokens, then normally A will increase. A remains unchanged only if we extend a slice by rearranging code to include token(s) that were not previously in any slice.

A.3 Evaluation of Orderings and Cohesion Metrics

To validate that the three measures, *SFC, WFC,* and A, assume an ordinal scale we need to demonstrate that the orderings imposed by the measures are consistent with the elementary viewpoints of the associated cohesion attributes. Such a conclusion relies heavily on intuition, since elementary viewpoints are defined in terms of subjective views of cohesion. Our main goal here is to demonstrate that the measures are consistent with intuition. At the very least, we are convinced that the orderings imposed by the measures are not counterintuitive. The measures are on an

ordinal scale to the extent that the orderings imposed by the measures match the users (of the measures) intuition concerning the elementary viewpoints of cohesion.

B. Cohesion Measures and the Ratio Scale

To perform multiplication and division on measurement values, the measures must assume a ratio scale. Thus we evaluate our functional cohesion measures in terms of the requirements for ratio scale measurement.

One way to demonstrate that a measure is on the ratio scale involves adding a program composition operator "$\circ$" to the relational system used in an ordinal scale evaluation. A composition operator takes two slice abstractions and combines them to create a new slice abstraction. Adding $\circ$ to the cohesion viewpoint of Section IV-A, gives us a relational system $(\mathcal{P}, \star\geq, \circ)$. Zuse [41, p. 49–50] shows that a measure is on a ratio scale if the measure is a real valued function m, is on an ordinal scale, and the following axioms hold:

$$P_1 \star\geq P_2 \Leftrightarrow m(P_1) \geq m(P_2)$$

$$m(P_1 \circ P_2) = m(P_1) + m(P_2)$$

The first axiom requires that m be consistent with the intuitive ordering of the procedure imposed by the attribute being measured. The second axiom requires that m be additive.

Meaningful composition operators are necessary to use Zuse's method of verifying that a measure assumes a ratio scale. In the extended version of this paper [3], we define two composition operators. One operator ties the output of the slices in one abstraction to the inputs of another abstraction. The second operator assumes no interactions between the two merged abstractions.

The requirement that $m(P_1 \circ P_2) = m(P_1) + m(P_2)$ is not satisfied using either of the two composition operators. This is because the size attribute $|tokens(p)|$, the number of tokens in the procedure, is in the denominator of the calculation for all three of the cohesion measures (*SFC,* *WFC,* and A). Under the two composition operators, the measures are not additive, and, thus, do not assume a ratio scale.

Gustafson, Tan, and Weaver argue that composition operators for the complex models (such as slice abstractions) used to define structural measures do "not make sense" because programmers rarely merge programs [14]. As an alternative to the analysis based on composition operators, we can use an intuitive argument that the functional cohesion measures do not assume a ratio scale. Multiplication makes sense for ratio scale measures. Thus, if the functional cohesion measures are on a ratio scale, we should be able to argue that one procedure (or slice abstraction) is twice as cohesive as another. We can find slice abstractions *s1* and *s2*, where *SFC(s1) = 2SFC(s2)*, *WFC(s1) = 2WFC(s2)*, or $A(1) = 2A(2)$. However, we find no justification (other than the measures themselves) for claiming that any *s1* is *twice* as cohesive as *s2*. The notion of doubling cohesion is not intuitive, and multiplying cohesion values does not seem to be meaningful. Thus, we find no evidence that the functional cohesion measures assume a ratio scale.

V. Examples

In this section, we examine a few small code segments to illustrate the differences among the three proposed cohesion measures. The figures in this section use slice profiles (as in Fig. 2) showing the entire procedure text rather than slice abstractions showing only data tokens to make it easier to visualize the connection between program text and slices. As described in Section II-B, the slices in the examples are the union of the backward and forward slices based on the output variables.

The first example uses a procedure that transforms a value in one of two ways depending on the initial value. A flag that indicates which of the two transformations was used is also returned. Fig. 5 contains a slice profile and cohesion measurements for this *Decode* procedure. In this case the three measures give equivalent values. The cohesion measurements are always equivalent for two slice procedures since in such cases $G(SA(p)) = SG(SA(p)$. The .53 measurement values indicate that approximately half of the tokens lie on both slices.

The three cohesion measurements are lowered when the procedure is modified by adding an output variable that is not connected to the slices of the original outputs. The modified procedure, *Decode2*, is in Fig. 6. *Decode2* was created by adding a variable *count* to the original procedure *Decode*. It is a global variable that may indicate the number of times that *Decode2* is called. *SFC(Decode2)* is zero, and clearly indicates the existence of some noncohesive components in the procedure — the slice for output variable *count* does not include any tokens that lie on the slices for the other outputs. *WFC(Decode2)* has dropped to .42 and $A(Decode2)$ has dropped further down to .28. Of *WFC* and A, A is more dramatically affected by adding the noncohesive component.

Figures 7, 8, and 9 demonstrate how the measures behave when functionality is combined. Procedure *LookUp* in Fig. 7 is a table lookup routine which returns a password and address associated with a key, and a boolean flag which indicates a successful search. As can be seen in Fig. 7, the three cohesion measures give relatively high values for this procedure, *WFC(LookUp)* = 1.0, *A(LookUp)* = .90, and *SFC(LookUp)* = .70. Most of the data tokens affect or are affected by the three outputs.

In Fig. 8, we combine procedure *LookUp* with procedure *Decode* from Fig. 5 to create procedure *LookUp2*. The procedures are combined such that *Decode* operates on the same data used by *LookUp*. The cohesion measurement values for this procedure are *WFC(LookUp2)* = .83, *A(LookUp2)* = .69, and *SFC(LookUp2)* = .43. The original procedure *Decode* is intuitively less cohesive than procedure *LookUp*. In this combined case, *WFC* and A fall between their values for the two original procedures, while *SFC* has a value that is below the value of either of the original procedures. *SFC* tends to drop dramatically, when non-cohesive components are added.

value	small	
1	1	procedure Decode(var value: integer;
1	1	var small: boolean);
		begin
2	2	if value < 5000 then begin
4		value := value * 8 mod 10;
2	2	small := true
		end
		else begin
3		value:=value mod 10;
2	2	small := false
		end;
		end;

$$WFC(Decode) = \frac{8}{15} = .53$$

$$A(Decode) = \frac{8 \cdot 2}{15 \cdot 2} = .53$$

$$SFC(Decode) = \frac{8}{15} = .53$$

Fig. 5. A slice profile and cohesion measurements for a simple procedure

value	small	count	
1	1		procedure Decode2(var value: integer;
1	1		var small: boolean;
		1	var count: integer);
			begin
2	2		if value < 5000 then begin
4			value := value * 8 mod 10;
2	2		small := true
			end
			else begin
3			value:=value mod 10;
2	2		small := false
			end;
		3	count := count +1;
			end;

$$WFC(Decode2) = \frac{8}{19} = .42$$

$$A(Decode2) = \frac{8 \cdot 2}{19 \cdot 3} = .25$$

$$SFC(Decode2) = \frac{0}{19} = 0.0$$

Fig. 6. A slice profile and cohesion measurements for a noncohesive procedure.

Procedures *LookUp* and *Decode* are again combined in Fig. 9 creating procedure *LookUp3*. This time we combine the procedures such that *Decode* operates on data that is distinct from the data used by *LookUp*. For this combined procedure, *WFC(LookUp3)* = .83, *A(LookUp3)* = .43, and *SFC(LookUp3)* = 0.0. *SFC* clearly indicates with a value of zero that there are no data tokens that are common to all of the slices. *WFC* does not distinguish between *LookUp2* and *LookUp3* — according to *WFC* the two procedures are equally cohesive. *A* does indicate that *LookUp3* is less cohesive than *LookUp2*, however, unlike *SFC*, *A* also indicates that there are some cohesive components.

These two examples show that A rather than WFC more accurately matches our intuition concerning the cohesiveness of a procedure which contains several functional components. This is true, in general. For a more detailed analysis of the sensitivity of the cohesion measures, see the extended version of this paper [3].

VI. Related Work

Our current efforts are based on earlier work using slice based measures as indicators of cohesion [21], [33], [28], [30]. Longworth [21] and Thuss [33], [28] examined the potential of measures proposed by Weiser [35] as indicators of cohesion. Ott and Thuss first noted the visual relationship that existed between the slices of a module and its cohesion as depicted in a slice profile [28]. The insights gained from this earlier work were instrumental in developing the data slice model of cohesion and cohesion measures presented here.

Other researchers have also examined the problem of measuring cohesion including Emerson [8], [9], Lakhotia [20], Troy and Zweben [34], and Selby and Basili [32].

A. Emerson's work

Emerson bases his cohesion measure on a control flow graph representation of a module [8], [9]. The graph contains a node for each statement in the module that contains a variable. After construction of the graph, a reference set is constructed for each variable in the module which indicates the nodes in the control flow graph that reference that variable. A flow subgraph, $<R>$, is computed for each reference set, R, as the minimal subgraph of F which contains every complete path in F that passes through an element of R. This is equivalent to generating the set of vertices which are either reachable from an element of R or from which an element of R is reachable. A cohesion value is computed for each reference set as the ratio of the cyclomatic complexity of $<R>$ times the size of R to the cyclomatic complexity of F times the size of F. The cohesion of a module is then computed as the mean of the cohesion values of the reference sets for each variable in the module. The values for Emerson's complexity measure range from 0 to 1. Discrimination levels are suggested to map these values to three levels of cohesion: data cohesion, control cohesion, and superficial cohesion.

Emerson indicates that his flow graph and reference set constructs are related to slicing [9]. Emerson computes flow subgraphs based on generating all vertices which are either reachable from an element of R or from which an element of R is reachable. Thus, these flow graphs are more closely related to metric slicing than Weiser's original definition of slicing [35]. Weiser only used "backwards slices" while Emerson's subflowgraph is clearly related to both forwards and backwards slicing.

The measure defined by Emerson is somewhat analogous to the *coverage* measure defined in [28]. (*coverage* is the average of the ratios of the lengths of each slice to the module length.) Emerson's measure is the average of the ratios of the size of each reference set (weighted by the cyclomatic complexity of the subgraph generated from the reference set) to the size of the flow graph (weighted by the cyclomatic complexity of the flow graph). Emerson computes

success	passwd	address	
3	3	3	procedure LookUp(A: Table; Size: integer; key: keytype;
1	1	1	var success: boolean;
1	1		var passwd: integer;
1		1	var address: string);
			begin
2	2	2	i := 1;
2	2	2	success:= false;
3	3	3	while not success and i <= Size do
3	3	3	if A.name[i] = key then
			begin
2	2	2	success := true;
3	3		passwd := A.value[i];
3		3	address := A.add[i];
			end
			else
3	3	3	i := i + 1;
			end;

$$WFC(LookUp) = \frac{27}{27} = 1.0$$

$$A(LookUp) = \frac{8 \cdot 2 + 19 \cdot 3}{27 \cdot 3} = .90$$

$$SFC(LookUp) = \frac{19}{27} = .70$$

Fig. 7. A table lookup procedure.

success	passwd	address	
3	3	3	procedure LookUp2(A: Table; Size: integer; key: keytype;
1	1	1	var success: boolean;
1	1		var passwd: integer;
1		1	var address: string);
			begin
2	2	2	i := 1;
2	2	2	success:= false;
3	3	3	while not success and i <= Size do
3	3	3	if A.name[i] = key then
			begin
3	3		passwd := A.value[i];
2	2		success := true;
3		3	address := A.add[i];
			end;
			else
3	3	3	i := i + 1;
2	2		if passwd < 5000 then begin
	4		passwd := passwd * 8 mod 10;
2	2		success := true;
			end
			else begin
	3		passwd := passwd mod 10;
2	2		success := false;
			end
			end;

$$WFC(LookUp2) = \frac{33}{40} = .83$$

$$A(LookUp2) = \frac{16 \cdot 2 + 17 \cdot 3}{40 \cdot 3} = .69$$

$$SFC(LookUp2) = \frac{17}{40} = 0.43$$

Fig. 8. A table lookup procedure combined with a decode procedure such that both use the same data.

success	passwd	address	value	small	
3	3	3			procedure LookUp3(A: Table; Size: integer; key: keytype;
1	1	1			var success: boolean;
1	1				var passwd: integer;
1		1			var address: string;
			1	1	var value: integer;
			1	1	var small: boolean);
					begin
2	2	2			i := 1;
2	2	2			success:= false;
3	3	3			while not success and i <= Size do
3	3	3			if A.name[i] = key then
					begin
3	3				passwd := A.value[i];
2	2				success := true;
3	3	3			address := A.add[i];
					end;
					else
3	3	3			i := i + 1;
			2	2	if value < 5000 then begin
			4		value := value * 8 mod 10;
			2	2	small := true; end
					else
			3		value := value mod 10;
			2	2	small := false; end
					end;

$$WFC(LookUp3) = \frac{35}{42} = .83$$

$$A(LookUp3) = \frac{15 \cdot 2 + 20 \cdot 3}{42 \cdot 5} = .43$$

$$SFC(LookUp3) = \frac{0}{42} = 0.0$$

Fig. 9. A table lookup procedure combined with a decode procedure such that both use distinct data.

reference sets and subgraphs for each variable while *coverage* is based only on slices for output variables. Although there is an apparent relation between these two measures, the precise meaning of Emerson's measure is unclear. In particular, the effect of multiplying the reference set by the cyclomatic complexity is to mask the view of cohesion. Cyclomatic complexity is a control flow measure, and combining the measures of different attributes weakens the discriminating power of a measure [24]. In contrast, our slice based cohesion measures are based on intuitively sound abstractions that are designed to isolate functional cohesion attributes from other factors.

B. Lakhotia's work

Lakhotia developed a method for computing cohesion based on an analysis of the variable dependence graphs of a module [20]. Pairs of outputs are examined to identify any data or control dependences that exist between the two outputs. Rules are provided for determining the cohesion of the pairs. For example, "two variables have sequential cohesion if one has data dependence on the other." The cohesion of a module is then defined to be "functional if it has only one output variable; it is undefined if it has no output variables; else it is the lowest cohesion of all pairs of the output variables of the module." Through examples Lakhotia argues that this method closely matches the original classifications (*coincidental*, *logical*, *temporal*, *procedural*, *communicational*, *sequential*, and *functional*) of cohesion [39, pp. 108]. Rather than develop an algorithmic mechanism to determine the original levels of cohesion, our objective is to quantify the amount of functional cohesion. Thus, in certain situations we will obtain differing results. For example, our measures will indicate that a significant part of a module is highly cohesive. In contrast, Lakhotia's method will indicate the lowest type of cohesion demonstrated by the module. Only a module with a single output exhibits functional cohesion in Lakhotia's model. This is equivalent to identifying functional cohesion only in the cases when $SFG(P) = 1$. We are able to generate relative levels of functional cohesion using our measures.

C. Other work related to cohesion

Two other studies examine cohesion indicators rather than attempting to measure cohesion directly. Troy and Zweben examined the quality of structured designs using, in part, some design cohesion indicators [34]. They used

- The number of effects listed in the design document;
- The number of effects other than I/O errors;
- The maximum *fan-in* to any one box in the structure chart, that is, the number of lines emanating upward from that box;
- The average *fan-in* in the structure chart; and

- The number of possible return values

as indicators of cohesion. They did not find evidence of a clear relationship between these measures and the "quality" of the software. Quality is measured here by the number of source code modifications. These negative results may mean that cohesion is not related to number of source code modifications or that these measures are not indicative of cohesion. Troy and Zweben did not attempt to show a relationship between these measures and cohesion.

Selby and Basili examined a measure based on data interactions, called data bindings, as a basis for computing the cohesion and coupling of the components of a system [32]. Routines are placed into clusters based on the data bindings and the coupling of a cluster with other clusters is determined. A ratio of the cluster coupling factor to the internal strength of a cluster is computed. An experiment indicated that clusters with a high ratio had the most errors and the highest error correction efforts. Selby and Basili also did not attempt to show a relationship between their measure and cohesion.

VII. Conclusions

Using principles from measurement theory, we derive a set of three functional cohesion measures. First, we develop an abstraction of procedures to isolate intuitive attributes of functional cohesion. This abstraction is based on data slices of procedures. Using the data slice abstraction, we define the concept of *glue* and *super-glue* data tokens. We also introduce the concept of data token *adhesiveness*. Using the slice abstraction and the concept of glue, super-glue and adhesiveness, we derive the measures. *Strong functional cohesion (SFC)* is based on the relative number of super-glue tokens in a procedure. *SFC* is the measure most closely related to the original definition of functional cohesion of Yourdon and Constantine [39, ch. 7]. *Weak functional cohesion (WFC)* is based on the relative number of glue tokens in a procedure and includes some notion of Yourdon and Constantine's weaker categories of cohesion. *Adhesiveness* is based on the relative "stickiness" of the glue tokens in a procedure, and is the measure that is most sensitive to minor program modifications.

We show that the measures satisfy the requirements of an ordinal scale to the extent that the orderings imposed by a set of simple transformations match our intuition concerning functional cohesion. We are not able to demonstrate that the measures are on a ratio scale. The measures are not additive under two possible composition operations, and the multiplication of cohesion values is not intuitive. As a result, one can use ordinal scale computations when analyzing measurement values, but ratio scale computations are not justified. Thus, analyses requiring a median value are meaningful, but a statistical analysis that requires a mean may not be valid.

We show analytically that, for a given procedure p, $SFC(p) \leq A(p) \leq WFC(p)$. We also show, through a series of examples, that *Adhesiveness* appears to be the most sensitive and potentially most useful of the proposed measures.

We do not show that our functional cohesion measures can predict software process attributes such as reliability or maintainability. Rather, we have derived ordinal measures of an important attribute of programs — functional cohesion. A well-defined measure is a prerequisite to empirical studies that relate one attribute to another.

Tools to automate the measurement of functional cohesion are more difficult to develop than tools to measure control flow structure. However, such automated measurement tools are feasible — they can make use of the kind of data flow analysis often performed by compilers. We are now developing functional cohesion measurement tools for empirical studies. One empirical study that we plan to conduct involves relating the traditional cohesion classes: *coincidental*, *logical*, *temporal*, *procedural*, *communicational*, *sequential*, and *functional* cohesion to our functional cohesion measures. In a sense, these cohesion classes are different levels of functional cohesion. We would expect a module with only coincidental cohesion to measure near zero for our three proposed measures. However, we do not know how our measures will evaluate modules that fall into the other cohesion classes. Such a study could help demonstrate whether or not the traditional cohesion classes are actually on an ordinal scale.

Appendix

Application of Transformations

We follow the transformations described in Section IV-A.1 to evaluate the orderings implied by the three functional cohesion measures. In the following discussion, we assume that slice abstraction a is modified to create a',

Strong Functional Cohesion (SFC) Orderings

1. Add a slice to a creating a'.
   (a) Adding a new output to a. (This requires adding at least one token to the procedure.) With this transformation, $SFC(a') < SFC(a)$. Adding an output always reduces *SFC* because a new functionality is added. Adding a slice can never increase the super-glue tokens, but it is likely to increase the non-super-glue if a new token is added. Our intuition about *SFC* is that fewer functionalities, in terms of output data, is always more cohesive.
   (b) Output existing functionality without adding any tokens. In this case, $SFC(a') \leq SFC(a)$. Adding a slice still cannot increase the number of super-glue tokens, while the number of non-super-glue tokens might not change.
2. Extend one or more slices in a creating a'. We have two cases here:
   Case 1: $|a| = 1$
   $SFC(a') = SFC(a)$ since a' is still a one slice abstraction.
   Case 2: $|a| > 1$
   Case 2(a): Extend a slice by adding a new data token.
   i: $SFC(a') < SFC(a)$ if the added token is

new and is added to only one slice. No new super-glue tokens are created but the total number of tokens (non-super-glue tokens) has increased.

ii: *SFC(a') > SFC(a)* if the added token is new and is added to all of the slices. One new super-glue token is created.

Case 2(b): *SFC(a') = SFC(a)* if the added token is not new but is not in **all** of the other slices in a then no new super-glue or non-super-glue is created.

Case 2(c): *SFC(a') > SFC(a)* if the added token is not new and is in all of the other slices in a. This transformation turns a non-super-glue token into super-glue.

To summarize, when an incremental change increases the number of super-glue tokens in a procedure with more than one slice, *SFC(a') > SFC(a)*.

Weak Functional Cohesion (WFC) Orderings

1. Add a slice to a creating a'.
   (a) Add functionality by adding a new output to the program. Here, *WFC(a') > WFC(a)* if and only if the net effect is to "glue" previously non-cohesive parts creating a higher percentage of glue tokens. If $g = G(a') - G(a)$, the set of new glue tokens created by the added functionality, and $t =$ *tokens(a') − tokens(a)*, the set of added tokens, then *WFC(a') > WFC(a)* if and only if $\frac{|g|}{|t|} >$ *WFC(a)*. The potential for increasing weak functional cohesion depends on the amount of glue in the original slice abstraction, a. If there is a significant number of non-glue tokens in a, then there is a lot of potential to increase the weak functional cohesion in a by adding a slice.
   (b) Output existing functionality without adding new data tokens, then *WFC(a') ≥ WFC(a)*. We are creating a new slice, and some tokens that lie on one slice in a may lie on the new slice in a' as well. New glue tokens can be created in this manner, but the total number of tokens does not change. It is possible that all of the tokens on the new slice do not lie on any other slices. In this case, *WFC(a') = WFC(a)*. This can only happen if there are values produced that are never referenced by any of the slices for all of the output tokens in a.
2. Extend one or more slices in a creating a'. Again, we have two cases here:
   Case 1: $|a| = 1$
   WFC(a') = WFC(a) since a' is still a one slice abstraction.
   Case 2: $|a| > 1$
   Case 2(a): Add a new token. If it extends only one slice, then there is no new glue added and *WFC(a') < WFC(a)*. If new glue is added, then *WFC(a') > WFC(a)*.
   Case 2(b): *WFC(a') ≥ WFC(a)* when the added token is not new but is not in **all** of the other slices in a. New glue is created if the token added to the slice is in just one of the other slices and *WFC(a') > WFC(a)*. If the added token is already a glue token, then no new glue is created and *WFC(a') = WFC(a)*.
   Case 2(c): *WFC(a') = WFC(a)* when the added token is not new and is in all of the other slices. The added token is already a glue token and thus the *WFC* value does not change.

Adhesiveness (A) Orderings

1. Add a slice to a creating a'.
   (a) Add functionality by adding a new output. If we add only non-glue tokens, then $A(a') < A(a)$. We have increased $|\text{tokens(a)}| \cdot |a|$ without adding any glue tokens.
   If we add both glue and non-glue tokens, then we can determine the increase or decrease of adhesiveness in terms of the number of new glue tokens, g, created by the added functionality, the number of new tokens added, n, the number of tokens, $|tokens(a)|$, and number of slices, $|a|$, in the original slice abstraction, a. Using algebraic transformations, we find that if $g/(|tokens(a)| + n + n \cdot |a|) > A(a)$, then $A(a') > A(a)$, if $g/(|tokens(a)| + n + n \cdot |a|) = A(a)$, then $A(a') = A(a)$, if $g/(|tokens(a)| + n + n \cdot |a|) < A(a)$, then $A(a') < A(a)$.
   (b) Add more glue, but no tokens to the procedure. Then, clearly $A(a') > A(a)$ since we increase the numerator but the denominator is unchanged.
2. Extend a slice:
   Case 1: $|a| = 1$
   There is no change, $A(a) = A(a')$, since *Adhesiveness* $= 1$ for any one-slice abstraction.
   Case 2: $|a| > 1$
   Case 2(a): Extend a slice by adding a token:
   i. Add a superglue token: $A(a') > A(a)$
   ii. Add a glue (but not super-glue) token: The relationship between $A(a')$ and $A(a)$ depends on the ratio of the number of slices, s, that the new token lies on and the total number of slices in the abstraction, $|a|$. If $A(a) > s/|a|$ then $A(a) > A(a')$, otherwise $A(a) \leq A(a')$.
   iii. Add a non-glue token: $A(a') < A(a)$
   Case 2(b) and 2(c): Extend a slice without adding a token to the abstraction; the token(s) used to extend the slice are already in the procedure: $A(a') \geq A(a)$. In the normal case the data token(s) added to a slice already lie on at least one additional slice, thus when they are added to the extended slice, the *adhesiveness* of a' increases, and $A(a') > A(a)$. It is only possible for $A(a') = A(a)$ when a slice is extended by rearranging code to include token(s) that were not previously in any slice.

Acknowledgements

We are grateful for travel support provided by the Michigan Technological University Faculty Development Committee. We are also grateful for the support provided by the NASA Langley Research Center, the Colorado Advanced Software Institute, CTA Inc., and Storage Technology Inc. We thank Colorado State for providing the resources for Prof. Ott during her sabbatical year when this collaborative research effort began.

We especially thank Norman Fenton, Horst Zuse, Kurt Olender, Scott Gordon, Teresa Hale, Byung-Kyoo Kang, Sakari Karstu, Janne Leminen, Marie Vans, Jeff Walls, Litao Wu, Hwei Yin and Josephine Xia Zhao who reviewed earlier versions of this manuscript. Their comments greatly improved the paper. We also received valuable insights from the graduate students in our seminars in software measurement at Michigan Technological University and Colorado State University.

We thank the anonymous referees for their careful reviews. Their comments greatly improved both the content and the presentation. We acknowledge the contribution of the referee who suggested the proposed empirical study described in the conclusions. Finally, we thank the Associate Editor, Bev Littlewood, for his timely management of the review process for this paper and for his suggestions concerning revisions.

References

[1] A. Baker, J. Bieman, D. Gustafson, N. Fenton, A. Melton, and R. Whitty, "A philosophy for software measurement," *J. Syst. & Software*, vol. 12, no. 3, pp. 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," *J. Syst. & Software*, vol. 8, no. 1, pp. 13–37, Jan. 1988.

[3] J. Bieman and L. Ott, "Measuring Functional Cohesion (Extended Version)," Tech. Rep. CS–93–109, Computer Science Dept., Colorado State Univ. Tech. Rep. CS–93–1, Computer Science Dept., Michigan Technological Univ., 1993.

[4] J. Bieman and J. Schultz, "Estimating the number of test cases required to satisfy the all-du-paths testing criterion," in *Proc. Software Testing, Analysis Verification Symp. (TAV3–SIGSOFT89)*, pp. 179–186, Dec. 1989.

[5] J. Bieman and J. Schultz, "An empirical evaluation (and specification) of the all-du-paths testing criterion," *IEE Software Eng. J.*, vol. 7, no. 1, pp. 43–51, Jan. 1992.

[6] P. Bollmann-Sdorra and H. Zuse, "Prediction models and software complexity measures from a measurement theoretic view," in *Proc. 3rd Int. Software Quality Conf. (3ISQC)*, 1993.

[7] J.-D. Choi, B. Miller, and P. Netzer, "Techniques for debugging parallel programs," Tech. Rep. 786, Univ. Wisconsin-Madison, 1988.

[8] T. J. Emerson, "Program testing, path coverage, and the cohesion metric," in *Proc. Computer Software and Applications Conf. (COMPSAC-84)*, pp. 421–431, 1984.

[9] T. J. Emerson, "A discriminant metric for module cohesion," in *Proc. 7th Int. Conf. Software Eng. (ICSE-7)*, pp. 294–303, 1984.

[10] N. Fenton, *Software Metrics - A Rigorous Approach.* London: Chapman and Hall, 1991.

[11] N. Fenton, "Software measurement: A necessary scientific basis," *IEEE Trans. Software Eng.*, vol. 20, no. 3, pp. 199–206, 1994.

[12] L. Finkelstein, "A review of the fundamental concepts of measurement," *Measurement*, vol. 2, no. 1, pp. 25–34, 1984.

[13] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Trans. Software Eng.*, vol. 17, no. 8, pp. 751–761, 1991.

[14] D. Gustafson, J. Tan, and P. Weaver, "Software measure specification," in *Proc. First ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 163–168, 1993.

[15] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *ACM Trans. Programming Languages and Systems*, vol. 11, no. 3, pp. 345–386, 1989.

[16] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Programming Languages and Systems*, vol 12, no. 1, pp. 35–46, 1990.

[17] B. Korel and J. W. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[18] B. Korel and J. W. Laski, "Stad – a system for testing and debugging: User perspective," in *Proc. 2nd Workshop Software Testing, Verification and Analysis (TAV2)*, 1988.

[19] Arun Lakhotia, "Insights into relationships between end-slices," Tech. Rep. CACS TR-91-5-3, Univ. Southwestern Louisiana, Sept. 1991.

[20] Arun Lakhotia, "Rule-based approach to computing module cohesion," in *Proc. 15th Int. Conf. Software Eng. (ICSE-15)*, pp. 35–44, 1993.

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

[22] H. D. Longworth, L. M. Ottenstein [Ott], and M. R. Smith, "The relationship between program complexity and slice complexity during debugging tasks," in *Proc. Computer Software and Applications Conf. (COMPSAC86)*, pp. 383–389, 1986.

[23] A. Macro and J. Buxton, *The Craft of Software Engineering*, Addison Wesley, 1987.

[24] A. Melton, D. Gustafson, J. Bieman, and A. Baker, "A mathematical perspective for software measures research," *IEE Software Engineering Journal*, vol. 5, no. 5, pp. 246–254, 1990.

[25] L. M. Ott, "Using slice profiles and metrics during software maintenance," in *Proc. 10th Annual Software Reliability Symp.*, pp. 16–23, 1992.

[26] L. M. Ott and J. M. Bieman, "Effects of software changes on module cohesion," *Proc. Conf. on Software Maintenance*, Nov. 1992.

[27] K. J. Ottenstein and L. M. Ottenstein [Ott], "The program dependence graph in a software development environment," in *Proc. ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, 1984. See also SIGPLAN Notices, vol. 19, no. 5, pp. 177–184.

[28] L. M. Ott and J. J. Thuss, "The relationship between slices and module cohesion," in *Proc. 11th Int. Conf. on Software Eng.*, pp. 198–204, 1989.

[29] L. M. Ott and J. J. Thuss, "Using slice profiles and metrics as tools in the production of reliable software," Tech. Rep. CS-92-8, Dept. Computer Science, Michigan Technological Univ., April 1992. Also published as Tech. Rep. CS-92-115 Dept. Computer Science, Colorado State Univ.

[30] L. M. Ott and J. J. Thuss, "Slice based metrics for estimating cohesion," *Proc. IEEE-CS Int. Software Metrics Symp.*, pp. 71–81, 1993.

[31] T. Reps and W. Yang, "The semantics of program slicing and program integration," in *Proc. Colloquium on Current Issues in Programming Languages*, pp. 360–374, 1989. Lecture Notes in Computer Science, vol. 352, Springer-Verlag, New York, NY.

[32] R. Selby and V. Basili, "Analyzing Error-Prone System Coupling and Cohesion," Tech. Rep. UMIACS-TR-88-46, Computer Science, University of Maryland, June 1988.

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

[34] D. Troy and S. Zweben, "Measuring the quality of structured designs," *J. Systems and Software*, vol. 2, pp. 113–120, 1981.

[35] M. D. Weiser, "Program slicing," in *Proc. 5th Int. Conf. on Software Eng.*, pp. 439–449, 1981.

[36] M. D. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

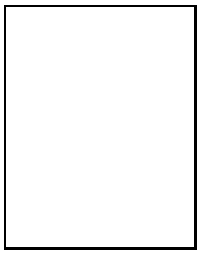
[37] M. D. Weiser, "Program slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[38] L. Wilson and L. Leelasena, "The QUALMS program documentation," Tech. Rep. Alvey Project SE/69, SBP/102, South Bank Polytechnic, London, 1988.

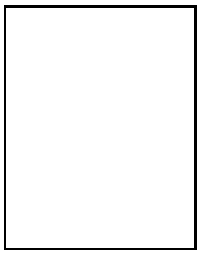
[39] E. Yourdon and L. Constantine, *Structured Design.* Englewood Cliffs, NJ: Prentice-Hall, 1979.

[40] H. Zuse and P. Bollmann, "Software metrics: Using measurement theory to describe the properties and scales of software

complexity metrics," *ACM SIGPLAN Notices*, vol. 24, no. 8, pp. 23–33, Aug. 1989.
[41] H. Zuse, *Software Complexity Measures and Methods*. Berlin: W. de Gruyter, 1991.
[42] H. Zuse, "Support of validation of software measures by measurement theory," Invited Presentation at the 15th Int. Conf. Software Eng. (ICSE-15) and the First IEEE-CS Int. Software Metrics Symp., Baltimore, MD, May 1993.



James M. Bieman is an Associate Professor in the Computer Science Department at Colorado State University. Dr. Bieman's research is focused on automated software testing and software measurement. He is evaluating the use of executable specifications and software probes as practical software testing oracles. He is developing methods to quantify software reuse, functional cohesion, and attributes of object oriented software. He is a senior member of the IEEE, a member of the ACM, and is currently the Chair of the IEEE-CS TCSE Subcommittee on Quantitative Methods and Chair of the Steering Committee for the IEEE-CS International Symposium on Software Metrics.



Linda M. Ott is an Associate Professor of Computer Science at Michigan Technological University. Her research is focused on measuring functional cohesion of software and software specifications and exploring the existence of relationships between functional cohesion and other software quality and software process attributes. Dr. Ott is also interested in functional and data cohesion of object-oriented software. She is a member of the IEEE Computer Society, ACM, and is currently editor of Q-Methods Report, the newsletter of the IEEE-CS TCSE Subcommittee on Quantitative Methods, and is Vice-chair of that subcommittee.