Measuring Functional Cohesion

James M. Bieman
Colorado State University

Linda M. Ott
Michigan Technological University

Submitted for Publication

Technical Report CS–93–109

June 24, 1993

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 and composition operators.

Index terms — software metrics, cohesion, program slices, measurement theory

1 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. Module cohesion can be classified using an ordinal scale that includes *coincidental*, *logical*, *temporal*, *procedural*, *communicational*, *sequential*, and *functional* cohesion [37]. 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 [9]. Our aim is to develop quantitative measures that

---

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.

Authors addresses: J. Bieman, Department of Computer Science, Colorado State University, Fort Collins, CO 80523. Email: bieman@cs.colostate.edu, (303)491-7096, Fax: (303) 491-6639; L. Ott, Department of Computer Science, Michigan Technological University, Houghton, MI 49931, Email: linda@cs.mtu.edu, (906) 487-2187.

indicate the degree of functional cohesion, the most desirable of these functional strength cohesion categories. Note that one can also evaluate cohesion from the perspective of data abstraction [21]. Fenton describes this *abstract* or *data* cohesion as a different notion of cohesion with a different set of measurement attributes [9]. In this paper, we address functional cohesion; we defer the treatment of abstract or data cohesion to future work.

The relationship between software engineering objectives and cohesion measurement can be clarified through the *Goal/Question/Metric (GQM)* paradigm of Basili and Rombach [2]. The GQM paradigm requires that software measurements be tied to a software engineering *goal.* Our goal is to improve maintenance programmer productivity and effectiveness. This goal leads to *questions* (for example, 'how can we develop code that is easy to modify?'). The answers to such questions lead to *metrics.* Intuition suggests that "cohesive" modules are easier to understand and are thus easier to maintain. Our intuition concerning cohesion is important, since intuition is the first stage of measurement [10]. Thus, we are motivated to develop cohesion measures. In particular, we now focus on *functional cohesion* measures, with the ultimate goal of developing measures that can predict *maintainability.*

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 [9]. Physical science measurement techniques suggest the following process [1]:

1. Identify and define intuitive and well-understood attributes of cohesion. We must qualitatively understand what we want to measure. The categories of cohesion (*functional* — *coincidental*) have been identified and are fairly well understood. Functional cohesion is a key cohesion category, and thus we focus our current efforts on developing functional cohesion measures.

2. Specify precisely the documents and the attributes to be measured. We must be able to describe precisely the object that we measure, and the property of the object that the measurement is to indicate. Our efforts focus on measuring cohesion in program source code. A major objective of this paper is to define functional cohesion attributes.

3. Determine formal models or abstractions which capture these attributes. Formal models are required to unambiguously produce numerical measurement values. This is another one of our objectives.

4. Derive mappings from the models of attributes to a number system. The mappings must be consistent with the orderings implied by the model attributes. The mappings are the cohesion measures.

Our objectives include the development of (1) a good model of functional cohesion, and (2) measures that use the model to quantify functional cohesion. We also validate the measures by demonstrating that they are consistent with expected cohesion model orderings, and determining their scale properties.

The role of experimentation in software measurement research is to map structural measures back to goals. But, before we can conduct effective empirical research, we must first have sound measures [1].

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.

A program slice is the portion of program text that affects a specified program variable [33]. A variation on program slices can model and measure functional cohesion [26]. A slice profile is one heuristic tool that can help one visualize the cohesion in a procedure [23]. In this paper, we focus on the development and analysis of quantitative measures that indicate the "amount of functional cohesion" in procedures. 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.

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 [36] 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 [3]. We use a *slice abstraction* of a program based on *data slices* to model cohesion [24].

Functional cohesion itself is a complex attribute and can be described in terms of sub-attributes. Particular sub-attributes of cohesion must be visible in the cohesion model. To be measurable on an ordinal scale, an attribute 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 [22].

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 [39].

The scale type of a measure is of critical importance. A useful measure may be of nominal, ordinal, interval, ratio, or absolute scale. Fenton succinctly describes the requirements for these scales:

> "Broadly speaking any attribute which imposes only a classification on a set of entities has nominal scale type, and any attribute which imposes only a linear ordering has ordinal scale type. If in addition to a linear ordering there is a notion of 'relative distance' between entities, then the scale type is interval. If there is also a 'zeroness' relation, i.e. some of the entities observably possess *nothing* of the attribute, then the scale type is ratio since the zeroness relation must be mapped to 0. *Absolute scales* arise out of attributes which amount to simple counts of entities." [9]

The scale of a measure determines the arithmetic operations that can be meaningfully performed on measurement values. For example, an arithmetic mean is well defined on interval, ratio, and absolute scale measures. However, an arithmetic mean of ordinal scale measurement values is **not** meaningful as an average. Rather, a median should be used for ordinal scale measures.

Our analysis focuses on the relative effect of alternative changes in order to determine measurement properties such as scale. A modification to a procedure changes the cohesion model, and such changes affect the cohesion measures. We analyze these changes to insure that the cohesion orderings of the models before and after a modification are consistent with the orderings.

The paper has the following organization. In Section 2, we define the abstractions used to model functional cohesion. In Section 3, we examine the cohesion attributes and measures, and Section 4 evaluates the scale properties of the measures. In Section 5, we provide some examples of procedures, cohesion orderings, and cohesion measures. Section 6 is a review of related work. Our conclusions are given in Section 7.

2 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. 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.

2.1 Program Slices

Slicing is a method of program reduction introduced by Weiser [33, 34, 35]. 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 [6], maintenance [11, 13, 23], and testing [15, 16, 20, 27]).

Weiser's algorithm for computing slices is based on data flow analysis. It is suggested in [25] 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 [14, 29]. 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* [14] 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 analyses.

2.2 Data Slices

In [35], Weiser defined several slice based measures. Longworth [19] first studied their use as indicators of cohesion. In [28, 31], 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 [12]; 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

```
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: Data slice for *SumN* . Items included in the slice are contained within boxes.

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 (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* [17]. 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.

Figure 1 displays an example of a data slice embedded in a program. The slice for *SumN* in Figure 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 an example procedure to give a sense of the relationships among data slices. Figure 2 shows an example of a metric 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 [26, 31].

[1] That is from the *FinalUse* nodes as described in [14]

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 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.

2.3 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 Figure 1 and Figure 2 is:

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

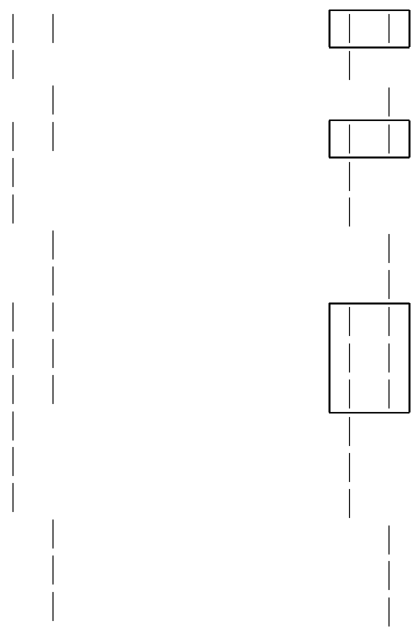
Figure 3(a) provides another view of a slice abstraction of the *SumAndProduct* procedure. The names of the data tokens are listed in the first column of Figure 3(a). For each row, a "|" in the second and third column indicates if the indicated data token is part of the data slice for the named output.

We find an uncluttered view of slice abstractions without labels useful for visualizing important attributes of functional cohesion in slice abstractions. Figure 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.

2.4 Glue, Super-glue, and Stickiness

As Figure 3(a) and Figure 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

Data Token	SumN	ProdN
N_1	$\vert$	$\vert$
$SumN_1$	$\vert$	
$ProdN_1$		$\vert$
I_1	$\vert$	$\vert$
$SumN_2$	$\vert$	
0_1	$\vert$	
$ProdN_2$		$\vert$
1_1		$\vert$
I_2	$\vert$	$\vert$
1_2	$\vert$	$\vert$
N_2	$\vert$	$\vert$
$SumN_3$	$\vert$	
$SumN_4$	$\vert$	
I_3	$\vert$	
$ProdN_3$		$\vert$
$ProdN_4$		$\vert$
I_4		$\vert$



(a) *SA(SumAndProduct)*

(b) Unlabeled View

(c) *Glue* tokens highlighted.

Figure 3: Three Views of *SA(SumAndProduct)*

	S_1	S_2	S_3
Super-glue:	$\mid$	$\mid$	$\mid$
	$\mid$		
		$\mid$	
			$\mid$
Super-glue:	$\mid$	$\mid$	$\mid$
		$\mid$	
Glue:	$\mid$	$\mid$	
Glue:		$\mid$	$\mid$
	$\mid$		
Glue:	$\mid$		$\mid$
			$\mid$

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

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 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. Figure 3(c) shows *SA(SumAndProduct)* with the glue tokens enclosed in boxes. Although there are two "|" symbols on each row of glue tokens in Figure 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 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.

Figure 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. See Section 5 for examples of procedures with three or more slices.

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 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.

3 Functional Cohesion Attributes and 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 (a probable 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 appearance 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 [37], 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 [28]. 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:

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

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

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

Another equivalent way to compute overall adhesiveness is based on the sum of the adhesiveness of individual tokens relative to the *adhesiveness space* — the sum of the potential adhesiveness assuming all of the tokens were super-glue. A is calculated as the ratio of the sum of number of slices containing each glue token in a slice abstraction to the adhesiveness space, the total potential number of slices containing each token. The adhesiveness space can be calculated as the number of data tokens in a procedure times the number of slices. For procedure p:

$$A(p) = \frac{\sum\limits_{t \in G(SA(p))} \#\ \textit{slices containing t}}{|tokens(p)| \times |SA(p)|} \tag{5}$$

In following examples we compute A using equation (5), since it 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 [28]. 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, 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.

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,

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

Adhesiveness is calculated as follows:

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

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 Figure 4 will differ since some of the glue tokens are not super-glue. This abstraction has 5 glue tokens and 2 super-glue tokens out of a total of 11 tokens. Thus *WFC(SA(Figure 4))*$= 5/11 = .455$ and *SFC(SA(Figure 4))*$= 2/11 = .182$. Since there are two tokens on three slices and three tokens on two slices, adhesiveness is calculated as follows:

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

The adhesiveness measure shows that the data tokens covered slightly more than one third of the slice space in the slice abstraction of Figure 4.

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.

4 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" [9]. 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 metric values. Zuse shows that software measures can be validated in terms of their scale properties [39, 40]. We combine the methods of Fenton and Zuse to analytically validate the cohesion measures in terms of intuitive notions of cohesion and scale properties. First we show that the measures are on an ordinal scale that matches our intuition concerning the cohesion attributes that are measured. Then we determine whether the metrics assume the requirements of a ratio scale.

4.1 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 [38, 39, 40]. These are the requirements of a *weak order*. From [38] 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 is equally as "cohesive" as P_2
$P_1 \star\geq P_2$	$P_1 \star> P_2$ 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. To show that a measure is on an ordinal scale we need to show that it is consistent with a set of *elementary transformations*. 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 Figure 3(b) to demonstrate necessary attributes and transformations.

4.1.1 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:

$$\begin{array}{c} | \\ | \\ \vdots \\ | \end{array}$$

A one slice procedure is entirely cohesive, and should have the highest possible *SFC*, *WFC*, and *A*. All three of our metrics 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 code is an example of such a transformation:

`y=x` $\Longrightarrow$ `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` $\Longrightarrow$ `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 an added token to a slice, when the token 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.

4.1.2 Strong Functional Cohesion Orderings

We follow the transformations above to evaluate transformations to slice abstraction a creating a':

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 better.

   (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, $SFC(a') > SFC(a)$.

4.1.3 Weak Functional Cohesion Orderings

We follow a similar approach to develop an ordering for WFC.

1. Add a slice to a creating a'.

   (a) Add functionality by adding a new output to the program. With this transformation, $WFC(a') > WFC(a)$ 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 $WFC(a') \geq WFC(a)$. 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, $WFC(a') = WFC(a)$. But 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): $WFC(a') \geq WFC(a)$ 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 already in one of the other slices and $WFC(a') > WFC(a)$. If the added token is not in any other slice then no new glue is created and $WFC(a') = WFC(a)$.

(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.

4.1.4 Adhesiveness Orderings

We see how the transitions affect the adhesiveness ratio.

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 tokens(a) $\times$ $|a|$ 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 $A(a') = \frac{g}{|tokens(a)|+n+n\cdot|a|}$. Thus, if $\frac{g}{|tokens(a)|+n+n\cdot|a|} > A(a)$, then $A(a') > A(a)$, if $\frac{g}{|tokens(a)|+n+n\cdot|a|} = A(a)$, then $A(a') = A(a)$, and $\frac{g}{|tokens(a)|+n+n\cdot|a|} < A(a)$, 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, s, of slices that the new token lies on and the total number of slices in the abstraction, $|a|$. If $A(a) > \frac{s}{|a|}$ 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: $A(a') \geq A(a)$. or $A(a') \leq A(a)$. In the normal case the data token(s) added to a slice already lie on at least one additional slice, thus increasing the *adhesiveness* of a', then $A(a') > A(a)$. It is only possible for $A(a') \leq A(a)$ when a slice is extended by rearranging code to include token(s) that were not previously in any slice.

4.1.5 Evaluation of Orderings and Cohesion Metrics

To validate that the three measures, *SFC, WFC,* and A, are on 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.

4.2 Cohesion Measures and the Ratio Scale

Zuse shows that ratio scale measures are often more meaningful than interval scales [39]. Thus we evaluate our functional cohesion measures in terms of the requirements for ratio scale measurement. To demonstrate that a measure is on the ratio scale we include a program composition operator "$\circ$" to the relational system. Thus we have a relational system $(\mathcal{P}, \star\!\geq, \circ)$.

According to Zuse, 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 the m be consistent with the intuitive ordering of the procedure imposed by the attribute being measured. The second axiom requires that m be additive. We examine the three functional cohesion measures to see if they are consistent with these axioms. One way to view the requirement that a cohesion measure be additive and is on a ratio scale is to determine the modifications to a procedure that will "double" its cohesion. We currently do not know how to "double" the cohesion in a procedure, however we can evaluate the effects of an intuitive composition operator.

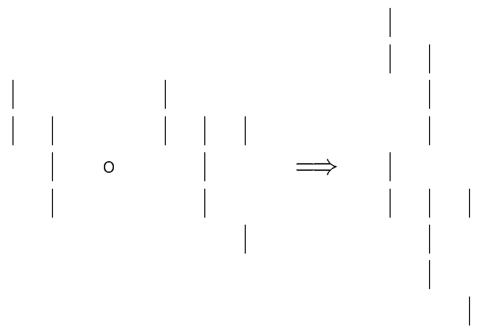
4.2.1 Composition Operators for Slice Abstractions

Adding more code to a program cannot increase its cohesion, rather adding code will tend to decrease cohesion. Thus, we expect that the $m(P_1 \circ P_2) \leq m(P_1)$, where m is one of the cohesion measures. Thus we reverse the direction of the inequality of the relation and use $\leq$ to compare the real number values produced by the measures instead of the relation $\star\geq$.

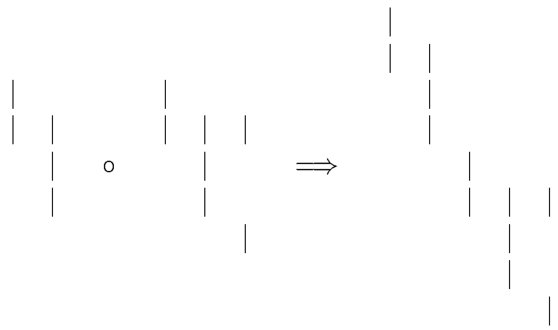
The binary composition operator is a critical component to Zuse and Bollman's method of demonstrating whether a measure is on a ratio scale. One demonstrates the scale properties relative to the binary operation. Zuse and Bollman developed the technique to evaluate control flow complexity measures [39], where binary composition is fairly easy to define for flow graphs. The difficulty of defining a composition operator for slice abstractions concerns the mechanism for combining the slices together.

We define two mechanisms for combining two slice abstractions into one:

Option 1: Tie slices together. When two slice abstractions A_1 and A_2 are "tied" $A_1 \circ A_2$, the outputs of A_1 are connected to the inputs of A_2. Each slice in A_1 is tied to a maximum of one slice in A_2, and each slice in A_2 is tied to a maximum of one slice in A_1. This is an optimistic composition operator, from the perspective of functional cohesion. It assumes that the output in the first program represented by A_1 is used by the second program represented by A_2. Thus, $|A_1 \circ A_2| = \max(|A_1|, |A_2|)$ The following is an example of this composition operator:



Option 2: Arbitrary composition: When two slices A_1 and A_2 are "arbitrarily composed", the slices in A_1 are not connected to the slices in A_2. This is a pessimistic mechanism for composition — it minimizes the functional cohesion in the resulting composed program. We compose the same two slice abstractions using the arbitrary operator:



The "width" (number of slices) of the resulting slice abstraction grows using this composition operator. With this operator, $|A_1 \circ A_2| = |A_1| + |A_2|$

4.2.2 Cohesion Measures and Composition

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), the requirement that $m(P_1 \circ P_2) = m(P_1) + m(P_2)$ is not satisfied. The measures are not additive, and are, thus, not on a ratio scale. However, the behavior of the measures under composition provides insight.

One property that is required (but not sufficient) for a ratio scale measure is that the measure be consistent with the axiom of *weak monotonicity* [38]:

$$P_1 \star\geq P_2 \Rightarrow P_1 \circ P_3 \star\geq P_2 \circ P_3$$

We use the axiom of weak monotonicity as a mechanism for comparing the three cohesion measures.

The axiom of monotonicity <u>does not hold</u> when a one-slice abstraction is composed with an abstraction that has two or more slices. Monotonicity does not hold because under either composition operator, the glue and super-glue tokens (all tokens in a one-slice abstraction are both glue and super-glue) become non-glue tokens. As a result, the cohesion of a one-slice abstraction does not contribute to the cohesion of the composition.

We examine the behavior of the cohesion measures to see whether the axiom of weak monotonicity holds when the composition operator is applied to abstractions either of an equal number of slices, or an unequal number of slices (assuming that neither of the unequal abstractions contains only one slice).

Composition Under Option 1 and Monotonicity

First we look at the case where the abstractions that are composed have an equal number of slices. In this case, the axiom of weak monotonicity holds for all three measures.

Consider the case where the abstractions a_1 and a_2 that are composed contain an unequal number of slices, and neither abstraction contains only one slice. In this case, *SFC* does not satisfy the axiom of monotonicity. All of the super-glue in the abstraction with fewer slices is not super-glue in the composition. The *WFC* and A measures exhibit the same behavior as when the composed abstractions contain the same number of slices, and the the axiom holds.

Composition Under Option 2 and Monotonicity

When arbitrarily composing two abstractions, $a_1 \circ a_2$, the outputs of a_1 are <u>not</u> connected to the inputs of a_2. As a result, the *SFC*($a_1 \circ a_2$)= 0 for any a_1 and a_2. There is no super-glue in such a composition. As a result, the axiom of monotonicity does not hold for *SFC*.

The computation of *WFC* is the same under Option 2 as under Option 1, thus *WFC*($a_1 \circ_{\text{Option 1}} a_2$)= *WFC*($a_1 \circ_{\text{Option 2}} a_2$) — a glue token is a glue token whether or not a program segment is tied to another program segment. Thus, the axiom of monotonicity holds for *WFC*. The axiom also holds for the computation of A, although $A(a_1 \circ_{\text{Option 1}} a_2) > A(a_1 \circ_{\text{Option 2}} a_2)$.

4.3 Scale Properties of the Measures

All three measures are on an ordinal scale to the extent that users accept the orderings of Section 4.1 as valid. We have evaluated the three measures in terms of two options for composition, Option 1 — tying slices together or Option 2 — arbitrary composition. None of the measures are on a ratio scale since normalization for size prevents the measures from being additive.

None of the measures satisfy the axiom of weak monotonicity when a one-slice abstraction is included in the composition. Even after eliminating one-slice abstractions, *SFC* does not satisfy the axiom under either composition option. However, if we limit the abstractions to those with two or more slices, both *WFC* and A satisfy the axiom of weak monotonicity under both composition options.

The adhesiveness measure, A, matches our intuition that arbitrary composition results in a less cohesive program than composition when slices are tied together . *WFC* does not distinguish between programs composed of the alternative operations. Thus, *Adhesiveness* appears to have the most desirable properties of the three measures.

5 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 Figure 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 2.2, the slices in the examples are the union of the backward and forward slices of 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. Figure 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 after 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 Figure 6. *Decode2* was created by adding a variable *count* to the original procedure *Decode*. The added variable *count* is incremented when *Decode2* is called. It is a global variable that may indicate the number of times that *Decode2* is called. The *SFC* measure for *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* has dropped to .42 and A 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 demonstrates how the measures can behave when functionality is combined. Procedure *Lookup* in Figure 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 Figure 7, the three cohesion measures give relatively high values for this procedure, $WFC = 1.0$, $A = .90$, and $SFC = .70$. Most of the data tokens affect or are affected by the three outputs.

In Figure 8, we combine procedure *Lookup* with procedure *Decode* from Figure 5 to create procedure *Lookup2*. The procedures are combined in a manner to simulate the tie slices option for composition described in Section 4. The cohesion measurement values for this procedure are lower than for the procedure in Figure 8, $WFC = .83$, $A = .69$, and $SFC = .43$. Procedure *Decode* is less cohesive than procedure *Lookup*. *WFC* and A fall between their values for the two original procedures, while *SFC* has a value that is below the value of either procedure. *SFC* tends to drop dramatically, when non-cohesive components are added.

Procedures *Lookup* and *Decode* are again combined in Figure 9 creating procedure *Lookup3*. This time, we combine the procedures in a manner to simulate the arbitrary composition option from Section 4. For this combined procedure, the cohesion measurements are $WFC = .83$, $A =$

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;

$$WFC = \frac{8}{15} = .53$$

$$A = \frac{8 * 2}{15 * 2} = .53$$

$$SFC = \frac{8}{15} = .53$$

Figure 5: A slice profile and cohesion measurements for a simple procedure

value	small	count	
1	1		procedure Decode(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;

$$WFC = \frac{8}{19} = .42$$

$$A = \frac{8 * 2}{19 * 3} = .25$$

$$SFC = \frac{0}{19} = 0.0$$

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

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;

$$WFC = \frac{27}{27} = 1.0$$

$$A = \frac{8*2+19*3}{27*3} = .90$$

$$SFC = \frac{19}{27} = .70$$

Figure 7: A table lookup procedure.

.43, and $SFC = 0.0$. *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.

6 Related Work

Our current efforts are based on earlier work using slice based measures as indicators of cohesion [19, 31, 26, 28]. Longworth [19] and Thuss [31, 26] examined the potential of measures proposed by Weiser [33] 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 [26]. 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 [7, 8], Lakhotia [18], Troy and Zweben [32], and Selby and Basili [30].

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;

$$WFC = \frac{33}{40} = .83$$

$$A = \frac{16 * 2 + 17 * 3}{40 * 3} = .69$$

$$SFC = \frac{17}{40} = 0.43$$

Figure 8: A table lookup procedure “tied” with a decode procedure.

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;

$$WFC = \frac{35}{42} = .83$$

$$A = \frac{15 * 2 + 20 * 3}{42 * 5} = .43$$

$$SFC = \frac{0}{42} = 0.0$$

Figure 9: A table lookup procedure "arbitrarily composed" with a decode procedure.

6.1 Emerson's work

Emerson bases his cohesion measure on a control flow graph representation of a module [7, 8]. 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 [8]. 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 [33]. Weiser only used "backwards slices" while Emerson's subflowgraph is clearly related to both forwards <u>and</u> backwards slicing.

The measure defined by Emerson is somewhat analogous to the *coverage* measure defined in [26]. (*coverage* is the average of the lengths of each slice to the module length.) Emerson's measure is the ratio of the average of the size of the reference sets (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 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 [22]. In contrast, our slice based cohesion measures are based on intuitively sound abstractions that are designed to isolate functional cohesion attributes from other factors.

6.2 Lakhotia's work

Lakhotia developed a method for computing cohesion based on an analysis of the variable dependence graphs of a module [18]. 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 [37]. 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 one output module exhibits functional cohesion. 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.

6.3 Other work related to cohesion

Troy and Zweben examined the quality of structured designs using, in part, some design cohesion indicators [32]. 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 [30]. 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.

7 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 Yourdan and Constantine [37]. *Weak functional cohesion (WFC)* is based on the relative number of glue tokens in a procedure and includes some notion of Yourdan 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 also show that the measures are not on a ratio scale because they are not additive. As a result, one can use ordinal scale computations but not ratio scale computations when analyzing the measurement values. Thus, analyses requiring a median value are meaningful, but a statistical analysis that requires a mean is not valid.

We analyze the monotonicity of the measures when procedures are combined using two options for composition. One of these options should maximize functional cohesion and the other should minimize it. We show that the *SFC* measure does not satisfy the axiom of weak monotonicity, while both *WFC* and *A* do when combining procedures with two or more slices. However *WFC* is not able to distinguish between procedures combined via the two alternative options for composition, while *A* can. Thus, *Adhesiveness* appears to be the most sensitive and potentially most useful of the proposed measures.

Acknowledgements

We are grateful for the support of Michigan Technological University for providing travel support for this work. 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, Sakari Karstu, Litao Wu, and Hwei Yin who reviewed earlier versions of this manuscript. Their comments greatly improved the paper. We also received valuable insights from the graduate students in Linda Ott's seminar in software metrics at Michigan Technological University.

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] V.R. Basili and H.D. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. Software Engineering*, SE-14(6):758–773, June 1988.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] T. J. Emerson. A discriminant metric for module cohesion. *Proc. 7th Int. Conf. on Software Engineering (ICSE-7)*, pages 294–303, 1984.

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

[10] L. Finkelstein. A review of the fundamental concepts of measurement. *Measurement*, 2(1):25–34, 1984.

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

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

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

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

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

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

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

[18] Arun Lakhotia. Rule-based approach to computing module cohesion. In *Proc. 15th Int. Conf. on Software Engineering (ICSE-15)*, pages 35–44, 1993.

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

[20] 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.

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

[22] 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.

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

[24] Linda M. Ott and James M. Bieman. Effects of software changes on module cohesion. *Proc. Conf. on Software Maintenance*, November 1992.

[25] 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.

[26] 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.

[27] 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.

[28] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. *Proc. IEEE-CS Int. Software Metrics Symp.*, pages 71–81, 1993.

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

[30] R. Selby and V. Basili. Analyzing Error-Prone System Coupling and Cohesion. Technical Report UMIACS-TR-88-46, Computer Science, University of Maryland, June 1988.

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

[32] D. Troy and S. Zweben. Measuring the Quality of Structured Designs. *Journal of Systems and Software*, 2:113–120, 1981.

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

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

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

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

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

[38] H. Zuse and P. Bollmann. Software metrics: using measurement theory to describe the properties and scales of software complexity metrics. *ACM SIGPLAN Notices*, 24(8):23–33, August 1989.

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

[40] H. Zuse. Support of validation of software measures by measurement theory. Invited Presentation at the 15th Int. Conf. on Software Engineering (ICSE-15) and the First IEEE-CS Int. Software Metrics Symp., Baltimore, MD, May 1993.