Program Slices as an Abstraction for Cohesion Measurement

Linda M. Ott

Michigan Technological University

James M. Bieman

Colorado State University

Abstract

The basis for measuring many attributes in the physical world, such as size and mass, is fairly obvious when compared to the measurement of software attributes. Software has a very complex structure, and this makes it difficult to define meaningful measures that actually quantify attributes of interest. Program slices provide an abstraction that can be used to define important software attributes that can serve as a basis for measurement. We have successfully used program slices to define objective, meaningful, and valid measures of cohesion. Previously, cohesion was viewed as an attribute that could not be objectively measured; cohesion assessment relied on subjective evaluations.

In this paper we review the original slice-based cohesion measures defined to measure functional cohesion in the procedural paradigm as well as the derivative work aimed at measuring cohesion in other paradigms and situations. By viewing software products at differing levels of abstraction or granularity, it is possible to define measures which are available at different points in the software life cycle and/or suitable for varying purposes.

1 Introduction

In his seminal work on slicing [25], Weiser presented several slice-based metrics. They were

- *Coverage* which is a comparison of the average number of statements in the slices of a module to the length of the module,
- *Overlap* which is an indication of how many statements in each slice are found only in that slice,

- *Clustering* which is a reflection of the layout of the statements in the code,
- *Parallelism* which is an indication of the percentage of slices with few statements in common, and
- *Tightness* which is a measure indicating the percentage of statements that are in every slice.

Although Weiser suggested the use of slices to define metrics, he did not identify actual software attributes that these metrics might meaningfully measure. He did report on one exploratory study using his metrics. Because of the exploratory nature, however, the findings were limited to observations on slicing.

Rather than focus on slicing, we can develop more generally useful measures by first identifying attributes of interest. We have studied the attribute of cohesion and its relationship with slices.

The concept of cohesion has been well known for nearly two decades [26]. For much of this time, cohesion was viewed as an attribute that could only be subjectively evaluated. Although various aids were developed for classifying modules such as the Page-Jones decision tree [22], as Fenton noted in his 1991 book on software metrics [5]

> Unlike coupling, cohesion does not appear to admit of a graph-type model. As such it is far more difficult to attempt to define anything more than the ordinal measure proposed ...

We found that slices can be used to create a model for measuring cohesion.

2 Slices and Cohesion Measurement

Longworth [17] was the first to hypothesize that some of the slice-based metrics suggested by Weiser might be used as indicators of cohesion. He demonstrated that *Coverage*, a modified definition of *Overlap*, and *Tightness* could be used to differentiate between high and low levels of cohesion.

In [21,24] Thuss improved the behavior of the metrics through the use of *metric slices*. A *metric slice* takes into account both the *uses* and *used by* data relationships [11]; that is, they are the union of Horwitz et.al.'s backward and forward slices [12]. In addition, Thuss argued that the intent of a module in the procedural paradigm is to compute a value or values that are communicated to the remainder of the program or the external environment via output variables. Thus, he argued that limiting the slicing criteria to one slice for each of the output variables would highlight the intended purpose of the modules. He also developed slice profiles as a tool to help visualize the relationships among

slices [20].

We next studied the suitability of these metrics as indicators of cohesion [18]. The concept of metric slices was further refined through the the use of data tokens (i.e., variable and constant definitions and references) rather than statements as the basic unit of which slices are composed [18]. These slices were called *metric data slices*. Using data tokens as the basis of the slices ensured that all changes of interest would cause a change in at least one slice of a module. A change of interest is any change that could affect the cohesiveness of a module. Changes of interest include adding code, deleting code, or replacing one variable instance with another variable. An analysis of a slice model of programs shows how program changes are reflected in the slice-based metrics. Since the behavior of the metrics matched our intuitive understanding of cohesion, this analysis reinforced the hypothesis that these metrics are indeed indicators of cohesion.

3 Measuring Functional Cohesion

Functional cohesion is the highest level of cohesion in the procedural paradigm. Slice abstractions provide a model for a set of metrics for measuring functional cohesion [3].

Functional cohesion is identified by examining procedure outputs. Each output "object" (output parameter, modified global variable, or file) represents one component of a procedure's functionality. Although a procedure may perform a computation that does not produce outputs, outputs of some kind are generally the externally visible manifestation of functionality. Functional cohesion is based on how closely the program parts that contribute to different outputs are bound. Using this approach, procedures with only one output exhibit maximum functional cohesion.

Each output of a procedure has a corresponding data slice. An "output" is any single value explicitly output to a file or device, an output parameter, or an assignment to a global variable. Since functional cohesion indicates the cohesion of the whole procedure, the measures use a concept similar to that of *end-slices* [14]. *Backward slices* are computed from the end of procedure [1] and *forward slices* are computed from the tops of the backward slices.

Figure 1 displays an example of a data slice embedded in a program. The slice for *Sum* in Figure 1 is a sequence of data tokens

$$n_1 \cdot Sum_1 \cdot i_1 \cdot Sum_2 \cdot 0_1 \cdot i_2 \cdot 1_2 \cdot i_3 \cdot n_2 \cdot i_4 \cdot Sum_3 \cdot Sum_4 \cdot i_5$$

[1] That is from the *FinalUse* nodes as described in [12]

Sum	Prod	Statement
2	2	void SumProduct(int n, *Sum, *Prod);
		{
1	1	int i;
2		*Sum = 0;
	2	*Prod = 1;
5	5	for (i=1; i<n; i++) {
3		*Sum = *Sum + i;
	3	*Prod = *Prod * i; }
		}

Fig. 1. Data Slice profile for *SumProduct*. The number of data tokens included in the data slice for *Sum* and *Prod* is indicated in columns 1 and 2 respectively. Items included in the data slice for *Sum* are underscored.

where each T_i indicates the i'th data token for T in the procedure. The slice for *Prod* is

$$n_1 \cdot Prod_1 \cdot i_1 \cdot Prod_2 \cdot 1_1 \cdot i_2 \cdot 1_2 \cdot i_3 \cdot n_2 \cdot i_4 \cdot Prod_3 \cdot Prod_4 \cdot i_6$$

A metric slice profile of the data slices, as shown in Figure 1, gives a sense of the relationships among data items. The column for a slice variable indicates the number of data tokens in that line that are included in the slice. This profile was derived from the earlier method developed by Thuss [20,24].

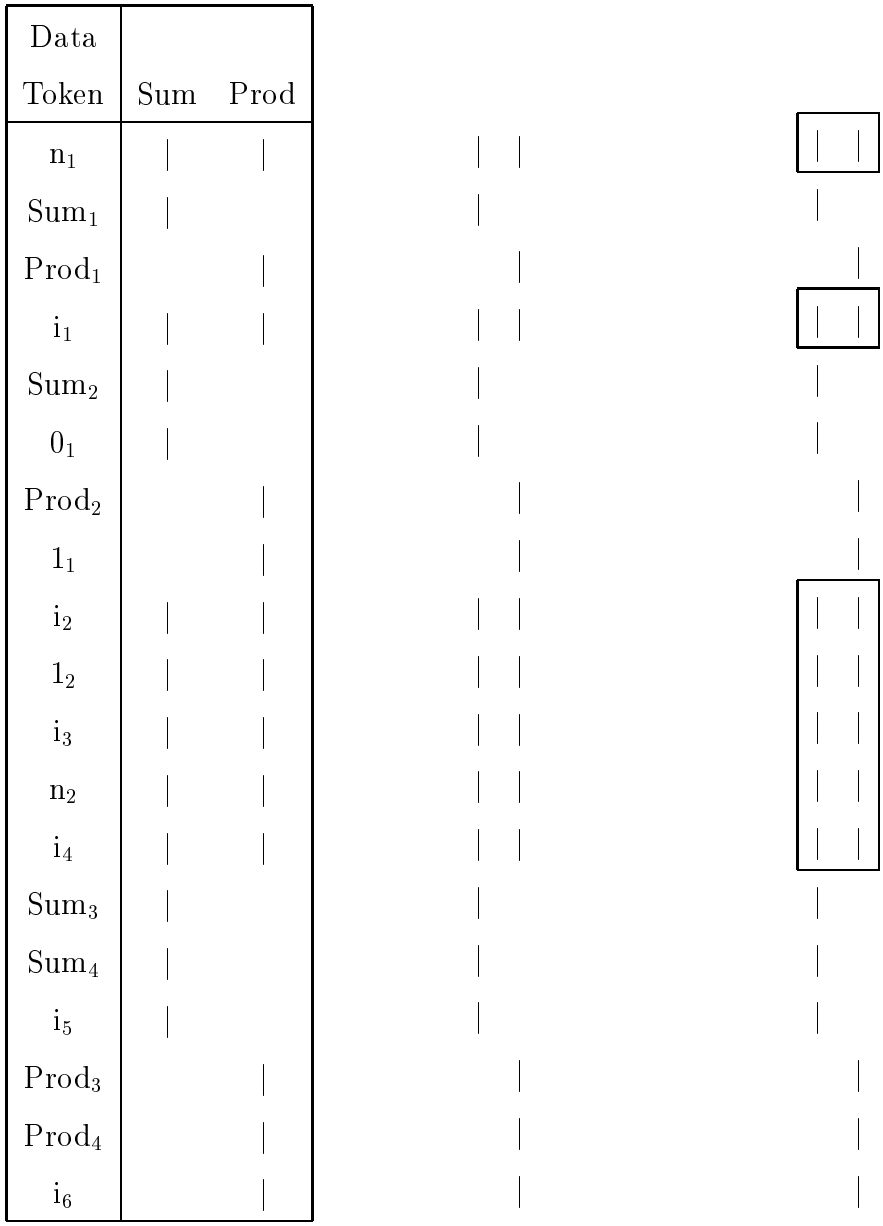
A *Slice Abstraction* models each procedure as a set of data slices and a data slice as a sequence of data tokens. The model strips away all non-data tokens from a procedure and includes only the data tokens in the abstraction.

The slice abstraction for the *SumProduct* procedure of Figure 1 is

$$\begin{aligned} &SA(SumProduct) = \\ &\quad \{ n_1 \cdot Sum_1 \cdot i_1 \cdot Sum_2 \cdot 0_1 \cdot i_2 \cdot 1_2 \cdot i_3 \cdot n_2 \cdot i_4 \cdot Sum_3 \cdot Sum_4 \cdot i_5, \\ &\quad n_1 \cdot Prod_1 \cdot i_1 \cdot Prod_2 \cdot 1_1 \cdot i_2 \cdot 1_2 \cdot i_3 \cdot n_2 \cdot i_4 \cdot Prod_3 \cdot Prod_4 \cdot i_6 \ \} \end{aligned}$$

Figure 2(a) provides another view of a slice abstraction of the *SumProduct* procedure. The names of the data tokens are listed in the first column of Figure 2(a). A "|" in the second and third column indicates if the indicated data token is part of the data slice for the named output.

Slice abstractions without labels are useful for visualizing important attributes of functional cohesion. Figure 2(b) is an unlabeled view of the slice abstrac-



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

Fig. 2. Three Views of *SA(SumProduct)*

tion of the *SumProduct* 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.

As Figure 2(a) and Figure 2(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, i_3, n_2, and i_4 are in

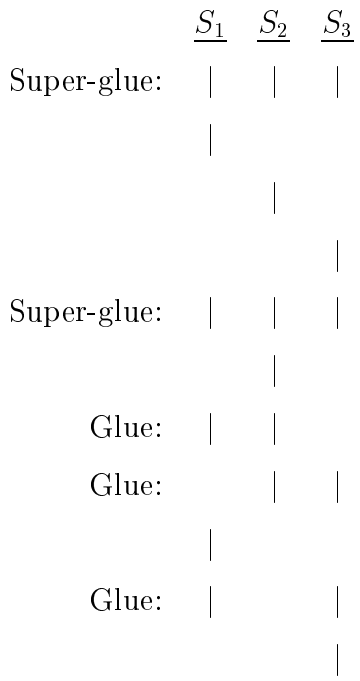


Fig. 3. A 3-slice *SA* with *glue* and *super-glue*.

the data slice for *Sum* and the data slice for *Prod*. Such tokens, common to more than one data slice in a slice abstraction, are the connections between the slices. These tokens are the "glue" that bind the slices. Thus, the *glue* in a slice abstraction of a procedure P, *G(SA(P))*, is 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 2(c) shows *SA(SumProduct)* with the glue tokens enclosed in boxes. Although there are two "|" symbols on each row of glue tokens in Figure 2(c), there is actually only one token for each row.

Note that all super-glue tokens are also glue tokens. Thus, when there are only one or two data slices in an abstraction, the set of super-glue tokens is the same as the set of glue tokens.

Figure 3 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 *adhesiveness* of a glue token. The notion of token adhesiveness can characterize the adhesiveness property of an entire procedure or slice abstraction.

Metrics based on the relative number of glue and super-glue tokens can easily be defined in terms of slice abstractions. According to Yourdon and Constantine [26], a procedure with functional cohesion is one in which all parts are cohesive. This view is consistent with the use of the super-glue tokens as the basis for defining cohesion attributes and measures. Thus, *strong functional cohesion (SFC)* is the ratio of super-glue tokens to the total number of data tokens in a procedure p:

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

SFC is similar to the *Tightness* measure studied by Ott and Thuss [21].

The glue tokens in a slice abstraction also represent a form of cohesion. Such functional cohesion is a "weaker" type of cohesion than indicated by the super-glue tokens. *Weak functional cohesion (WFC)* is the ratio of glue tokens to the total number of tokens in a 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. *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. The adhesiveness, α, of token t in procedure p is

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

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

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

Adhesiveness indicates the relative strength of the glue in a procedure. *Adhesiveness* is most closely related to the *coverage* measure studied by Ott and Thuss [21].

The *SumProduct* program of Figure 1 has two outputs, thus it has two slices and all glue tokens are also super-glue tokens. It has a total of 19 data to-

kens with 7 glue and super-glue tokens. As a result, $SFC(SumProduct) = WFC(SumProduct) = A(SumProduct) = .37$.

The *WFC* and *SFC* of the 3-slice abstraction in Figure 3 will differ since some of the glue tokens are not super-glue. Out of a total of 11 tokens, this abstraction has 5 glues tokens of which 2 are super-glue. Thus *WFC(SA(Figure 3))*= $5/11 = .45$ and *SFC(SA(Figure 3))*= $2/11 = .18$. Since there are two tokens on three slices and three tokens on two slices, *adhesiveness* is calculated as follows:

$$A(SA(Figure\ 3)) = \frac{2\times3+3\times2}{11\times3} = .36$$

Adhesiveness indicates that the data tokens covered slightly more than one third of the slice space in the slice abstraction of Figure 3.

An implementation of these measures for C programs is available [7]. A study by Karstu provided experimental evidence that cohesion as measured with slice-based cohesion measures may be inversely related to the number of revisions that a module undergoes during maintenance. This is some of the first experimental evidence for the long held belief that modules with high cohesion require less maintenance [13].

4 Design-level Functional Cohesion

Cohesion may also be measured using only procedure interface information. Bieman and Kang derive design-level cohesion measures using an approach very similar to the one described in Section 3 to measure code-level functional cohesion [2].

They define design-level functional cohesion (DFC) measures only in terms of the dependencies between module input and output components. Input components include all input parameters and referenced global variables. Output components include all output parameters, modified global variables, and values returned by a function. An input or output component has a dependence relation with a particular output component if its value affects the output. A design document can, and should, specify dependence relations. We can also derive dependence relations from code since all components that lie on an output's data slice have a dependence relation with that output.

DFC measures are defined in terms of:

- The total number of input and output components, T, and the number of output components, O.

- The number of *non-isolated* components, N. A component is *non-isolated* if it has a dependence relation with more than one output, or if the module has only one output. Non-isolated components are analogous to the glue tokens used in the code-level functional cohesion measures.
- The number of *essential* components, E. A component is *essential* if it has dependence relationships with all outputs of a module. Essential components are analogous to super-glue tokens.
- The *connectedness* of component i, C_i, This is the relative number of outputs with which a component has a dependence relation. The connectedness is analogous to the token adhesiveness used to compute code-level module adhesiveness. The connectedness of the i'th component is

$$C_i = \begin{cases} \frac{D_i - 1}{O - 1} \text{ if } O > 1 \\ 1 \qquad \text{otherwise} \end{cases}$$

  where the i'th component has dependence relations with D_i outputs.

The three DFC measures are analogous to the three code-level functional cohesion measures:

(1) *Loose cohesion,* $LC = N/T$, is analogous to code-level *weak functional cohesion (WFC).*
(2) *Tight cohesion,* $TC = E/T$, is analogous to code-level *strong functional cohesion (SFC).*
(3) *Module cohesiveness,* $MC = \frac{\sum_{i=1}^{T} C_i}{T}$, is analogous to code-level *adhesiveness (A).*

Like the code-level functional cohesion measures, a module with only one output has $LC = TC = MC = 1$. Also, a module with two outputs has $LC = TC = MC$, since all non-isolated components are also essential tokens — they have dependence relations with all two outputs. The two output program *SumProduct* of Fig. 1 has three input/output components: *n, Sum, Prod.* Only n has a dependence relation with more than one output. Thus, $T = 3$, $O = 2$, $N = E = 1$, and $LC = TC = MC = .33$.

The DFC measures are very similar to the code-level functional cohesion measures. Take a code-level slice abstraction as defined in Section 3 and delete all tokens that do not represent inputs and outputs. Such a *design-level slice abstraction (DSA)* for the *SumProduct* procedure is

$$DSA(SumProduct) = \{n \cdot Sum, n \cdot Prod\}$$

The code-level functional cohesion measures can be applied directly to a DSA. Then, $LC = WFC$, $TC = SFC$, and $MC = A$.

The DFC measures have also been implemented and the implementations are available over the world wide web [7]. In [2], Bieman and Kang provide a detailed comparison between the code-level and design level measures. Analytical results show that the DFC measures should behave in a similar manner to the code-level functional cohesion measures. Initial empirical results indicate that design level cohesion values can effectively predict the functional cohesion of an implementation.

5 Applying Slice-based Measures to Formal Specifications

Leminen applied the concept of slicing to study the cohesion of Z formal specification schemas [16]. He used Schema slicing to obtain a slice abstraction model of an operation schema. This model consists of a set of *schema data slices* obtained for each output of the operation. A *schema data slice* for a given output is a sequence of data tokens in the schema predicate that are related to that output. A *schema data slice* is based on the bindings of values to the schema variables in order to satisfy the schema predicate. Leminen identified a technique for obtaining schema slices using logical and precondition dependencies among the primitive clauses of a schema predicate.

He argued that cohesiveness in the specification domain and cohesiveness in the program domain are intuitively similar attributes. In both domains, cohesiveness can be characterized by the amount of relatedness between the functionalities of a module as manifested by its outputs. Measures analogous to *strong functional cohesion*, *weak functional cohesion*, and *adhesiveness* were developed and shown to match the intuitive cohesiveness of operation schemas.

6 Slice-based Cohesion Measures for Object-Oriented Software

As in procedural software, we can apply slicing notions to the measurement of attributes of object-oriented software. Again, cohesion is the attribute that seems most applicable to a slicing analysis.

Cohesion as used in the procedural paradigm can be applied directly to methods in the object-oriented world. Functional cohesion is an appropriate measure for procedural code where basic software units are procedures and functions. In object-oriented software, the basic design units are classes, which are collections of instance variables and methods. Functional cohesion cannot be applied directly to classes. Fenton suggests that what we are interested in here is a different form of cohesion, namely *data cohesion*, rather than *functional cohesion* [5].

We examine two slice-based approaches for measuring object-oriented cohesion [19]. One approach, by Ott and Gupta [9], is a direct extension of our work on functional cohesion. The other approach, by Bieman and Kang [1], evolved from Chidamber and Kemerer's measure of the lack of cohesion ($LCOM$) between methods [4]. In both approaches, the class is the basic unit and the instance variables are the "glue" that connects the methods in a class. The difference in the approaches is in the granularity of the analysis. Ott and Gupta slice method bodies while Bieman and Kang treat a method as an atomic unit.

6.1 Data Cohesion

In [9], Ott and Gupta apply the approach used to define functional cohesion measures to measure the cohesion of classes in an object-oriented paradigm. Again, they assume that cohesion is an indication that the elements of a module belong together, however, they refer to the elements of a class as the basic unit in object-oriented software. An object is represented by its behavior as reflected through its class methods and its state information as maintained in the class instance variables. The intent of a class is to model an object rather than computing a value as in the procedural paradigm. Hence, slices are obtained for each class method and then concatenated together to form a slice model. The measures for cohesion are based on the number of data tokens that appear in more than one slice and thus "glue" the module together.

Figure 4 is an example of a data slice profile for class *stack*. The figure indicates the number of data tokens included in the slice from each statement. The slice profile for a class is the concatenation of the slice profiles for all methods in the class.

A *class slice abstraction* of a class *C*, *CSA(C)*, is the set of concatenated slices, one for each instance variable, formed by concatenating the data tokens obtained from the method data slices for that instance variable.

The super-glue tokens for the class, denoted as, *SG(CSA(C))*, is the union of the super-glue tokens of each of the methods of the class. Similarly, the set of glue tokens for the class, denoted as, *G(CSA(C))*, is the union of the glue tokens of each of the member methods of the class. *tokens(C)* is a set of all data tokens of a class *C*. The following measures parallel the functional cohesion measures.

(1) *Strong data cohesion (SDC)* is a measure based on the number of data tokens included in all the data slices for a class, i.e. a count of the number

of super-glue tokens in the class C.

$$SDC(C) = \frac{|SG(CSA(C))|}{|tokens(C)|} \tag{5}$$

(2) *Weak data cohesion (WDC)* measures the amount of data cohesion in a class based on the glue tokens.

$$WDC(C) = \frac{|G(CSA(C))|}{|tokens(C)|} \tag{6}$$

(3) *Data adhesiveness (DA)* is a more precise measure of the binding or relatedness among the data slices. *Data adhesiveness* for a class C is defined as the ratio of the sum of the number of slices containing each glue token to the product of the number of data tokens in the class and the number of data slices. Thus,

$$DA(C) = \frac{\sum_{d \epsilon G(CSA(C))} \#\ slices\ containing\ d}{|tokens(C)| \times |CSA(C)|} \tag{7}$$

As an example, we apply these cohesion measures to the class *stack* in Figure 4. The *CSA(stack)* has three slices with 19 data tokens, 12 glue tokens, and 5 super-glue. Hence,

$$SDC(CSA(stack)) = \frac{5}{19} = .26$$

$$WDC(CSA(stack)) = \frac{12}{19} = .63$$

$$DA(CSA(stack)) = \frac{7 \cdot 2 + 5 \cdot 3}{19 \cdot 3} = .51$$

6.2 Class Cohesion

Bieman and Kang treat the method and instance variable class components as the key class units that may or may not be connected [1]. Thus, if a slice includes a method, it includes the method as an indivisible unit. A method and an instance variable are related by the way that an instance variable is used by the method. Two methods are related (connected) through instance variable(s) if both methods use the instance variable(s). Using this orientation, class cohesion can be measured by the relative connectivity (through instance variables) of the methods.

Individual methods in a single class can be connected via two mechanisms:

(1) *MIV relations* involve communication between methods through shared

array	top	size	Class Stack
			`class Stack {int *array, top, size;`
			`public:`
			`Stack (int s) {`
2		2	`size = s:`
2		2	`array = new int[size];`
	2		`top = 0;}`
			`int Isempty() {`
	2		`return top==0};`
			`int Size() {`
		1	`return size};`
			`int Vtop() {`
3	3		`return array[top-1];}`
			`void Push (int item) {`
2	2	2	`if (top==size)`
			`printf("Empty stack.\n");`
			`else`
3	3	3	`array[top++]=item;}`
			`int Pop() {`
	1		`if(Isempty())`
			`printf("Full stack.\n");`
			`else`
	1		`--top;}`
			`};`

Fig. 4. Data slice profile for class *stack*.

instance variables. An *MIV relation* is created when two or more class methods read or write to the same class instance variable.

(2) *Call relations* involve the sending of messages directly (or indirectly) from one method to another. Instance variables used by the server may also be used indirectly by the client when one method invokes another through message passing.

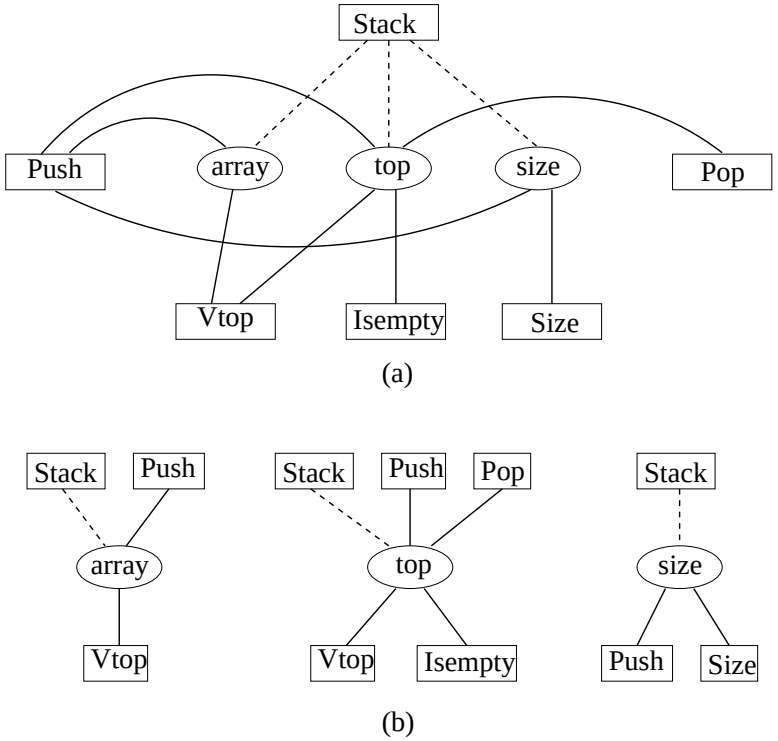


Fig. 5. MIV relations for class *Stack*

A call relation can be reflected by the MIV relation; two methods with a call-relation are also connected through the instance variables used by both methods. One method uses the instance variable(s) directly and the other uses the instance variable(s) indirectly through the call relation. There is no MIV relation when a server method neither writes nor reads instance variables.

Figure 5(a) shows the MIV relations among class components of *Stack* in Figure 4. A link between a rectangle and an oval indicates that the method corresponding to the rectangle uses the instance variable corresponding to the oval. Figure 5(b) shows the connections for each instance variable. Here, the instance variable *top* is used by the methods *Stack, Push, Pop, Vtop,* and *Isempty*. All of the methods that use the variable *top* are connected through the variable *top*. These methods should be defined in one class or in classes with an inheritance relationship in order to access the instance variable.

Constructors and destructor methods create connections between methods even if the methods do not have any other relationships. Thus, the model and measures do not include constructor and destructor functions. Dashed lines represent links between the constructor Stack and instance variables in Figure 5.

Cohesion of a class indicates the degree of connectivity of the visible methods in the class. Instance variables are not usually visible to the clients of a class, the state of an object is provided through class methods. Instance variables

are involved in MIV relations among visible methods. Invisible methods are also involved indirectly when they are called by visible ones. Therefore, class cohesion is modeled as the MIV relations among all visible methods (not including constructor or destructor functions) in the class.

Actual measures of class cohesion are based on the direct and indirect connections of method pairs. Let $NP(C)$ be the total number of pairs of visible methods in a class C. NP is the maximum possible number of direct or indirect connections in a class. If there are N methods in a class C, $NP(C)$ is $N * (N-1)/2$. Let $NDC(C)$ be the number of direct connections and $NIC(C)$ be the number of indirect connections in class C. Two measures are *Tight class cohesion* and *Loose class cohesion*:

(1) *Tight class cohesion (TCC)*, *TCC(C) = NDC(C) / NP(C)*, is the relative number of directly connected methods.
(2) *Loose class cohesion (LCC)*, *LCC(C) = (NDC(C) + NIC(C)) / NP(C)*, is the relative number of directly or indirectly connected methods.

The value of LCC is always greater than or equal to the value of the corresponding TCC. For the *Stack* example of Figure 4, the class cohesion measures are:

$$TCC(Stack) = 7/10 = 0.7$$
$$LCC(Stack) = 10/10 = 1$$

The TCC measure indicates that 70% of the visible methods in class *Stack* are directly related. The LCC measure shows that all visible methods of class *Stack* are related directly or indirectly.

TCC and LCC indicate the degree of connectivity between visible methods in a class. These visible methods are those defined within the class or inherited by the class. However, class cohesion measures for visible methods defined only within the class are also useful because the measures are not affected by the cohesion of a superclass.

Local class cohesion measures are defined by using the local (non-inherited) methods in a class. The instance variables used and methods called by the visible methods for local class cohesion may include inherited variables. The local class cohesion measures for class *Stack* are equal to the class cohesion measures since class *Stack* does not use inheritance.

Shumway [23] demonstrates that TCC and LCC are consistent with an empirical relation system defined by intuitive notions of class cohesion. He shows that these measures do, in large part, satisfy the representation condition of measurement. However, they do not completely reflect the ability to split a

class without breaking method connections. In a limited study, Bieman and Kang found that class cohesion is inversely related to reuse; classes exhibiting high class cohesion tended to be reused through inheritance less often [1].

7 Slicing Granularity

We use slices to help quantify software attributes. The attributes may be viewed at various levels of abstraction. At the code-level intra-procedure slices are needed to see connections between code components inside procedure bodies. At the design-level all details about the connections between components within procedures are not of interest. Slicing can be used at a level of abstraction that satisfies the measurement goals. A key *slicing granularity* decision is to determine what kinds of components are sliced and what kinds of components are indivisible and cannot be sliced.

The functional cohesion measures described in Section 3 quantify the cohesion within procedure and function bodies. The goal is to see how the internal components (i.e, statements, expressions, and data tokens) are connected. Thus, procedure and function bodies are sliced into divisible segments for analysis.

Procedure and function bodies need not be sliced to study a design. An analysis of the procedure and function interfaces and the connections between interface components should be adequate. If we take the perspective of a client of a procedure or function, we are only interested in the connections between the externally visible inputs and outputs in a module interface. This is the perspective of the design-level functional cohesion measures of Section 4. They do not make use of procedure/function body slicing.

The correct slicing granularity for class-level measures is not obvious for object-oriented software. Both approaches described in Section 6 assume that data connections are the glue that bind class components together. Thus, both use the perspective of slicing. However, the two approaches use different slicing granularities. The class cohesion measures of Section 6.2 treat a method as a single unit. The measures do not distinguish between (1) a single reference to an instance variable in only one statement in a method and (2) many references to an instance variable within most or all statements within a method.

In contrast, the data cohesion measures of Section 6.1 consider the degree that all method statements may affect an instance variable. For the strongest cohesion, all (or most) data tokens in all methods must affect the value of all instance variables. Thus, these data cohesion measures indicates a very "strict" view of cohesion. High cohesion classes will need to have very tightly

coupled methods. Because of this strict view, the data cohesion measures will show lower cohesion values than the class cohesion measures.

There are many cohesion measures based on alternative slicing strategies. There is no one best measure. The correct measure and slicing strategy to use depends on the goals of measurement. A measure must provide information that is relevant to the measurer. Measurement users must have clear objectives in order to wisely choose a measurement technique.

8 Future Measures Based on Inter-Class Slicing

Work on slice-based measurement has focused primarily on the measurement of intra-procedural or intra-class attributes such as cohesion. Inter-module attributes can also be viewed from the perspective of slicing. To measure inter-module attributes, we need to use models based on inter-procedural or inter-class structure.

Inter-module design attributes are likely to be more important than intra-module attributes. Intuition suggests that the connections or dependencies between components are what makes software difficult to design, test, modify and reuse. The intricacies of these connections in object-oriented code are especially complex. Object-oriented connections may be classified as aggregations, inter-class or intra-class method invocations, sub- or super-class connections, non-hierarchical links, links through shared instance variables, etc.

These inter-component connections are manifestations of coupling attributes. We are working now to understand the notion of coupling as applied to object-oriented software. Sets of object-oriented design patterns used by practitioners suggest desirable coupling [8] and may form the basis for defining measures. Before deriving measures, we need to clearly define the coupling attributes that we want to measure.

Any class-coupling measure definitions will be based on abstractions of object-oriented structure. Harrold and Rothermel define a set of abstractions for analyzing object-oriented software [10]. These abstractions include graphs to represent inheritance, class and interclass call graphs, class control flow graphs, and class dependence graphs. They also define the notion of framed classes to represent classes as analyzable components. OMT, Booch, or UML diagrams might be the right level of abstraction for defining coupling measures [6]. We can make use of these or similar abstractions to define relevant coupling attributes and their measures.

Inter-class and inter-method slicing is likely to be useful in defining and im-

plementing object-oriented cohesion measures. We can make use of object-oriented slicing methods such as those defined by Larsen and Harrold [15].

Before inter-component slicing can be applied to industrial strength object-oriented software, we need methods to correctly slice software with exception handling and concurrency. Another problem is caused by delayed binding. The type or class of an identifier might be implemented by a set of classes that are either subclasses of the named class or implementations of an interface. Overly large slices may result.

9 Conclusions

Long recognized as a key attribute of software designs [26], the concept of module cohesion has resisted objective definition and measurement. In the past module cohesion could only be assessed through a subjective evaluation by experts. Cohesion evaluations were rarely conducted due to their high cost.

We find that program slices can form the basis for models that capture the essence of various notions of software cohesion. Because program slices can be mapped to graph abstractions, they can form the basis for objective measurement and formal analysis. Because program slices can be viewed as program text, they provide the intuition needed to validate prospective measures.

Program slices should also prove useful in modeling notions of coupling. While intra-module slices model notions of cohesion, inter-module slices can model coupling.

Depending on measurement goals, slicing granularity can be adjusted to match desired levels of abstraction. To measure the functional cohesion of a procedure, slices consist of a sequence of procedure body data tokens. To measure the design-level functional cohesion of a procedure, slices consist of a sequence of input and output components. When measuring the cohesion of classes, method bodies may be sliced, or they may be treated as indivisible units.

Program slicing provides a flexible tool for defining software measures. The many available slicing options provides flexibility and are a strength of the method. One can select appropriate slicing mechanisms to build appropriate models of specific program attributes. Thus, a slice-based abstraction can match the desired level of abstraction and granularity to capture attributes of interest. Then measures based on the abstractions can effectively quantify desired attributes.

10 Acknowledgment

We thank the University of Maryland at College Park and Reliable Software Technologies Corp. in Sterling, Virginia for their support for Prof. Bieman during his sabbatical year when this article was prepared.

References

[1] J. Bieman and B-K Kang. Cohesion and reuse in an object-oriented system. In *Proc. ACM Symposium on Software Reusability (SSR'95)*, pages 259–262, April 1995.

[2] J. Bieman and B-K Kang. Measuring design-level cohesion. *IEEE Trans. Software Engineering*, 24(2):111–124, February 1998.

[3] J. Bieman and L. Ott. Measuring functional cohesion. *IEEE Trans. Software Engineering*, 20(8):644–657, August 1994.

[4] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

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

[6] M. Fowler. *UML Distilled Applying the Standard Object Modeling Language.* Addison-Wesley, Reading, MA, 1997.

[7] Funco. URL `http://www.cs.colostate.edu/~bieman/funco.html`.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[9] B. Gupta. A critique of cohesion measures in the object-oriented paradigm. Master's thesis, Department of Computer Science, Michigan Technological University, 1997.

[10] M. Harrold and G. Rothermel. A coherent family of analyzable graph representations for object-oriented programs. Technical Report OSU-CISRC-11/96-TR60, Ohio State Univ., 1996.

[11] M. Hecht. *Flow Analysis of Computer Programs.* North-Holland, 1977.

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

[13] S. Karstu. An examination of the behavior of the slice based cohesion measures. Master's thesis, Department of Computer Science, Michigan Technological University, 1994.

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

[15] L. Larsen and M. Harrold. Slicing object-oriented software. *Proc. 18th Int. Conf. Software Engineering (ICSE-18)*, March 1996.

[16] J. Leminen. Slicing and slice based measures for the assessment of functional cohesion of z operation schemas. Master's thesis, Department of Computer Science, Michigan Technological University, 1994.

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

[18] L. Ott and J. Bieman. Effects of software changes on module cohesion. In *Proc. Conference on Software Maintenance*, 1992.

[19] L. Ott, J. Bieman, B-K Kang, and B. Mehra. Developing measures of class cohesion for object-oriented software. In *Proc. 7th Annual Oregon Workshop on Software Metrics*, 1995.

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

[21] L. Ott and J. Thuss. Slice based metrics for measuring cohesion. In *Proc. IEEE-CS International Symposium on Software Metrics*, 1993.

[22] M. Page-Jones. *The Practical Guide to Structured Systems Design.* Yourdon Press, New York, 1980.

[23] M. Shumway. Measuring class cohesion in Java. Master's thesis, Department of Computer Science, Colorado State University, 1997. Available as Computer Science Technical Report CS-87-113 URL `http://www.cs.colostate.edu/~ftppub/TechReports/1997/tr97-113.ps.Z`.

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

[25] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

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