A Mathematical Perspective For Software Measures Research *

Austin C. Melton †, David A. Gustafson‡, James M. Bieman§, Albert L. Baker¶

Preprint of paper that appeared in *IEE/BCS Software Engineering Journal*, 5(5):246-254, 1990.

Abstract

We identify and analyze basic principles which necessarily underlie software measures research. In the prevailing paradigm for the validation of software measures there is a fundamental assumption that the sets of measured documents are ordered, and that measures should report these orders. We describe mathematically the nature of such orders. Consideration of these orders suggests a hierarchy of software document measures, a methodology for developing new measures, and a general approach to the analytical evaluation of measures. We also point out the importance of units for any type of measurement and stress the perils of equating document structure complexity and *psychological complexity*.

Keywords: software measures, abstractions of software documents, software structure, analytical evaluations of measures

1 Introduction

This paper presents some underlying principles for software measures research. By "software measures" we mean measures which are obtainable directly from software documents. (Lines of code is a common example of a software measure.) The careful characterization of software measures research as presented in this paper generalizes earlier efforts to provide axioms for software measures, e.g., [19, 38, 49], is consistent with measurement theory, e.g., [20, 34, 42], and provides a reasonable and precise approach to the definition and analytical evaluation of software document measures.

In this paper we provide a careful analysis of three problems in software measures research:

1. In the research to date we see no clear and explicit distinction between a *psychological complexity measure* and a *document measure*. We use the term *document measure* to refer to measures that are obtainable directly from particular types of documents. According to published claims, a *psychological complexity measure* should actually quantify the elusive notions of understandability, maintainability, estimated production costs, etc.[e.g. [1]] In Section 2 we comment further on this distinction of measure types and argue that researchers should first focus their efforts on the structural complexity of documents and provide analytical evaluations of measures of structural complexity. Later, using the mathematical perspective developed in the paper, we show that it is unlikely that a single document measure can serve as an adequate psychological complexity measure.

2. Software measures researchers are often guilty of failing to determine, or at least to state, what their measures quantify — they rarely attach units to measures. Units of measures is the topic of Section

*This work was supportd by NATO through the research collaboration travel grant 0343/88

†Department of Computing and Information Sciences, Kansas State University, Manhattan, Ks 66506, ph (913) 532-6350. This research was supported in part by the National Science Foundation grant DCR-8604080 and by the Office of Naval Research grant N00014-88-K-0455.

‡Department of Computing and Information Sciences, Kansas State University, Manhattan, Ks 66506, ph (913) 532-6350

§Department of Computer Science, Colorado State University, Ft. Collins, Co 80523 ph (303) 491-7096

¶Department of Computer Science, Iowa State University, Ames, Ia 50011 ph (515) 294-2165

3. We argue that labeling measures with actual units will help avoid misuse of proposed measures, increase attention on what is actually being quantified, and support a careful definition of composite document measures. We do not claim that the mathematical perspective presented in this paper will immediately allow us to attach units to all measures. However, we do feel that well-defined and well-developed abstractions (presented later) can be useful in determining units and in helping to define composite measures.

3. In general, researchers follow a prevailing paradigm when attempting to validate that a document measure is in fact a useful complexity measure. However, the assumptions that necessarily underlie this paradigm are not well-understood and have not been stated explicitly in the published literature. In Section 4 we describe this common paradigm and its underlying assumptions.

An important assumption inherent in the validation paradigm is the existence of orders of the sets of documents. Further, document orderings are inherent for any measure of the structure of documents. In Section 5 we provide useful mathematical definitions, and in Section 6 we consider the types of orders document sets might have. We argue that these orders do not have some of the properties that are often tacitly assumed in published measures research, and we define a *structural complexity measure* as a measure which preserves a given, well-defined order.

The importance of orders on sets of software documents becomes critical because a single set of documents probably has many different orders that could be defined on it. A researcher needs to specify what order is being worked with so that the results of incomparable software measures are not related or compared.

Many published document measures are defined for particular abstractions of actual documents. For example, McCabe's cyclomatic complexity measure [37] is defined for flowgraphs, and flowgraphs can be viewed as abstractions of source programs. ¿From these considerations of the orderings of abstractions of documents come a hierarchy of types of structural complexity measures. Document measures defined on abstractions of documents and the hierarchy of such measures are presented in Section 7.

Both theoretical and practical benefits result from our mathematical perspective on software measures research. We reduce the likelihood that a document measure will be misrepresented since the property being measured is clearly specified, and the mathematical perspective provides useful analytical evaluation techniques for structural complexity measures and a general approach to defining new document measures. This mathematical perspective should encourage the careful development and evaluation of structural complexity measures.

In this paper we are not presenting a complete methodology which can be immediately used to turn software measures (research) into a mature engineering discipline. We are, however, trying to determine a foundation upon which a mature discipline can be built. Some of the ideas presented here may seem well-known to experienced software measures researchers, and this recognition is a confirmation that we are on the right track in trying to define a foundation for software measures research. Further, it is important that these underlying assumptions are clearly stated and explained so that we can build on them.

2 The Perils of Psychological Complexity

Software measures are frequently described by researchers as measures of the "understandability" of a software document (usually program code). The underlying assumptions concerning human understanding are often unstated and not fully justified. A major problem is that *software structural complexity* may not be directly related to the difficulty of "understanding" a software document. However, useful results can still be obtained from the investigation of structural properties of software documents. For example, structural information can help a programmer construct a set of test cases [39, 23]. A danger lies in equating *structural complexity* with *psychological complexity*. See [18] for a general discussion of the need for software structural measures.

The following definition highlights that our focus is on measures obtainable directly from software documents:

Definition 2.1 Let D be a set of similar software documents and let $\Re$ be the set of real numbers. A *software document measure* m is a function from D to $\Re$. The mapping m is denoted $m : D \rightarrow \Re$.

We do not attempt to rigorously define "similar document." In the software engineering community, there is no agreement on the specific set of software documents that make up a software project. One software engineer's specification is another software engineer's detailed design, which may be another software engineer's implementation. The well-versed software engineer is capable of recognizing a large number of different document types. For example a software engineer can recognize an SADT design [43] or an abstract model formal specification [2, 43]. Thus, by "similar" documents we mean those recognizable as being of a similar type.

Our definition of a software measure is consistent with the general notion of a measure from measurement theory [23]. Also, note that a software measure is not a software "metric" because a "metric" is necessarily a function with two arguments [3].

Many published software document measures have been promoted either explicitly or implicitly as measures of psychological complexity. For example, Harrison and Magel's scope number [28] is described as quantifying the difficulty of understanding a decision in a program. However, a *psychological complexity measure* requires at least two arguments, including the document and the programmer interacting with the document [12].

The programmer component in models of psychological complexity is often slighted because human "understanding" is difficult to quantify. Halstead acknowledged the significance of a model of the programmer when he argued that his software document measure E actually quantifies the number of elementary mental discriminations [27]. Regardless of the validity of the mental discrimination argument [11, 13], Halstead models the programmer with a constant – the Stroud number [48].

The replacement of a variable with a constant may have insignificant side effects in certain mathematics and engineering applications. However, the side effects can be pronounced when programmers are treated as an invariant component in a psychological complexity measure. A number of researchers have found that the programmer is the most significant component – programmer performance can vary by orders of magnitude [5, 9, 14, 44].

The danger of assuming away the programmer component of psychological complexity can be demonstrated with a simple example. Consider the axioms developed by Prather to define a "proper measure" of program psychological complexity [38] and Property 5 of Weyuker [49]. Essentially, Prather and Weyuker assert that a procedure is at least as complex as the sum of the complexity of its parts, and a "proper measure" should not contradict this intuition. Although the notion of a "proper measure" appears sound, the intuition is not valid for measures of psychological complexity [46]. The composition of code segments can result in a larger program that has lower psychological complexity than the original segments. Most programmers will find that a fragment of a well-known algorithm is more difficult to understand that the entire implementation. For example, the code fragment in Figure 1 may make less sense to experienced programmers than the full Shell sort procedure in Figure 2. Researchers have used "schema", "clique", "chunk", "paradigms", "plans", "frames" etc. to describe the phenomenon that an entire conceptual unit is easier to understand than its components [17, 22, 41, 46, 47]

```
while j > 0 do
    if A[j] > A[j + incr] then begin
       swap(A[j], A[j + incr]);
       j := j − incr
    end
    else
       j := 0 (* break *)
```

Figure 1: Pascal Code Fragment

```
procedure Something( var A : array [1..n] of integer );
var
   i, j, incr : integer;
begin
   incr:= ndiv2;
   while incr > 0 do begin
      fori := incr + 1 to n do begin
         j :=i − incr;
         while j > 0 do
            if A[j] > A[j + incr] then begin
               swap(A[j], A[j + incr]);
               j := j − incr
            end
            else
               j := 0 (* break *)
      end ;
      incr := incr div 2
   end
end ; (*something*)
```

Figure 2: Pascal Procedure Using Code from Figure 1

We expect that strong connections exist between the structure of software documents and the psychological activities of software developers. For the present, however, we concentrate on *structural complexity issues* – the complexity that arises from a software document itself, independent of any cognitive issues. Eventually, with a good understanding of structural complexity, and with the aid of those trained to understand programmer psychology, the relation between structural complexity and psychological complexity can be investigated with some reasonable expectation of success.

3 What Are We Measuring?

An important component of any measure using interval, ratio, or absolute scales [20] is the unit of measurement [35, 36, 45]. A measure is actually more than a mapping to $\Re$ as in Definition 2.1. A measure might more properly be viewed as a function to a set of ordered *magnitudes* where a *magnitude* is a product of a real number and a unit; the unit distinguishes the magnitudes of different kinds of quantities [36]. For example, a program may have $40,000 \cdot [\textit{line of code}]$ or $4.5 \cdot [\textit{decision/module}]$ [1]. The real number values 4.5 and 40,000 cannot be directly compared unless there is a meaningful way to compare their units. For a comparison of different measures to be valid the units must be comparable.

In elementary physics and engineering courses, students are taught that the appropriateness of an answer can be determined by "calculating the units" [40]. For example, if a calculation is expected to give a result that is a measure of force, the answer should be in force units. Since $\textit{Force} = \textit{mass} \cdot \textit{acceleration}$, an answer with units such as $[kg \cdot meter/sec^2]$ is expected. As this example illustrates, many physical science measures

[1] We use the common convention of using square brackets to denote a unit of measurement

such as force are combinations of simpler measures. It is hoped that the field of software measures can develop so that software measures can be meaningfully combined.

Software structural measures are usually expressed without units. For example, the cyclomatic complexity of a flowgraph is usually expressed simply as an integer value. Yet, cyclomatic complexity is a measure with units based on the structure of a flowgraph, namely the number of *linearly independent paths* [*LI-path*] from the start node to the terminal node in a flowgraph [37]. As in physics, we should be able to combine software structural measures to form more complex and useful measures. Potentially useful measures defined in terms of cyclomatic complexity include the number of *LI-paths* through a particular node in a flowgraph, and the average number of *LI-paths* through all nodes. The units of either measure is [*LI-path*/*node*]. The meaningful combination and manipulation of software structural measures requires the combination and manipulation of the units of measurement.

When units are omitted, the meaning of a measurement is lost, i.e., without units of measurement, the specific property that is being measured is unknown. And if a unit of measure cannot be specified for a particular measure, the measure is especially suspect.

Consider the Halstead effort measure E [27]. E is described as a measure with [*mental discriminations*] as its unit of measure. The Halstead volume measure V is also described as a measure with the same units – [*mental discrimination*]. The measure $V = N \log_2 n$ can be assigned units of [*bit*]. (N is the total number of program tokens and n is the size of the vocabulary.) V is the minimum number of bits needed for a binary encoding of a program. Since both E and V are advertised as having units of [*mental discrimination*] and V is **actually** a software document measure in [*bit*] units, we expect E to also have [*bit*] measurement units. The expression for E used in practice [10] along with a dimensional analysis [36] is

$$E = \frac{V}{\hat{L}} = \frac{\eta_1 N_2}{2\eta_2} V \cdot \left[\frac{\textit{unique-operators} \cdot \textit{total-operands}}{\textit{unique-operands}} \cdot \textit{bit} \right]$$

Even with the assumption that *unique-operands* and *total-operands* are the same unit, the units of E cannot be reduced to *bit*. Thus, it is clear that E produces real numbers of a different kind of quantity than V.

The omission of units of measure can hide the "structuralness" of a software measure that is promoted as a psychological complexity measure.

4 Validation Paradigm and Document Orderings

A common approach is used by researchers to demonstrate that a software document measure is useful.[2] Using this common validation paradigm, a document measure is applied to a sample set of software documents (usually programs). Independently, a *quantified criterion* [14] which is not a document measure is applied to the sample set of programs. A *quantified criterion* is a "measure" that is intuitively related to desirable property of a software document. However, the magnitude of a *quantified criterion* is determined using some information that is not part of the software document. Example criterion that have been used by researchers include rankings of documents by experts [25], time or cost to produce or debug [1, 6, 7, 15, 26] number of errors corrected during development [6, 21, 24], the number of changes made during development [30], etc. The criterion may be quantified through experiments or case studies. The purpose of the validation is to determine how well the software document measure can predict the criterion.

We can take a more abstract view of the validation paradigm. Let D be a set of similar software documents, let m be a software document measure defined on D, and let c be a quantifiable criterion that is an intuitive measure of the psychological complexity of the software documents in D. Note again that c will use information not contained in D to quantify psychological complexity. The goal of a validation is to establish the degree to which m can predict c. A validation process takes the following form: choose a finite and small subset D' of D. D' is the document sample actually used in the validation. Obtain a ranking of the documents in D' using c and the other information required by c such as expert opinion, number of bugs, etc. The measure m is validated to the extent to which it preserves the order on D' obtained independently of m by c.

[2] In this discussion we are not concerned with the problems that result from neglecting or oversimplifying the human component of psychological complexity.

Regardless of any deficiencies with this prevailing validation paradigm [33], there are underlying assumptions that are pertinent and significant for software document measures research. These assumptions have not been stated previously. The first assumption is in fact a necessary condition for measurement [34].

Assumption 4.1 There is an order on the abstractions of the documents in D; the order is based on the relative complexity.

In validation studies, the "correct order" is determined by the criterion that is intended to reflect psychological complexity.

Assumption 4.2 A "valid measure" m preserves or carries the "correct order" on D into $\Re$. That is, if two documents d_1 and d_2 are related by the "correct order", then the images $m(d_1)$ and $m(d_2)$ are related in the same relative order in $\Re$.

These previously unstated assumptions concerning document orders provide a new perspective on the nature of software measures. We make use of the notion of document orders to develop mathematical properties of software document measures.

Our attention is focused on software document measures and not psychological complexity measures. However, even with software document measures we necessarily assume the existence of document orders. And a software document measure should preserve a meaningful order. A software document measure intended to quantify some aspect of the structural complexity of documents must be consistent with some intuitively sound order of the documents. ("Aspect" and "structural" are more fully developed in Section 6.) We need some formal definitions before we can more fully analyze the implications of these fundamental document order assumptions.

5 Relevant Mathematical Definitions

We review these first few basic mathematical concepts as a convenience to the reader.

Let S be a set. A *relation* R on S is a subset of the Cartesian product of S with itself. Thus, $R \subseteq S \times S$.

Definition 5.1 Given a set S and a relation R on S, R is

1. *reflexive* if $(x, x) \epsilon R$ for every $x \epsilon S$,
2. *antisymmetric* if whenever $(x, y) \epsilon R$ and $(y, x) \epsilon R$ then $x = y$,
3. *transitive* if whenever $(x, y) \epsilon R$ and $(y, z) \epsilon R$ then $(x, z) \epsilon R$.

Consider a directed graph representation of a relation R on a set S where each node represents an element of S and there is an edge from x to y if and only if $(x, y) \epsilon R$. We can use the correspondence between a relation R and the associated directed graph G to say:

1. R is reflexive if and only if for each node x in G there is an edge directed from x to itself;
2. R is antisymmetric if and only if G has no cycles of length two; and
3. R is transitive if and only if whenever there is a path from node x to node y there is also an edge from node x to node y.
4. If R is antisymmetric and transitive then G has no cycles of length greater than one.

The foregoing properties of relations are used to describe orderings:

Definition 5.2 Given a set S and a relation R on S:

1. R is a *pre-order* and (S, R) is a *pre-ordered set* if R is reflexive and transitive.
2. R is a *partial order* and (S, R) is a *partially ordered set* if R is reflexive, antisymmetric, and transitive.
3. R is a *linear* or *total order* and (S, R) a *linearly* or *totally ordered set* if R is a partial order and if for each pair of elements x and y in S either $(x, y) \in R$ or $(y, x) \in R$.

Clearly each linearly ordered set is a partially ordered set, and each partially ordered set is a pre-ordered set.

We use the common notational conventions of S for (S, R); $\leq_S$ or $\leq$ for the relation R; and $x \leq y$ for $(x, y) \in \leq$. Consider the following examples.

Example 5.3 Let D be a collection of programs. We can define a relation $\leq_L$ on D so that, given $d_1 \epsilon D$ and $d_2 \epsilon D$, $d_1 \leq_L d_2$ if and only if d_1 does not contain more lines of code than d_2. Obviously, $\leq_L$ is reflexive — a program cannot have more lines of code than itself. The relation is also clearly transitive. Thus $(D, \leq_L)$ is a pre-ordered set. However, if two different programs in D contain the same number of lines of code, then $\leq_L$ is not antisymmetric and $(D, \leq_L)$ is **not** a partially ordered set.

Example 5.4 Let T be a set, and let S be the power set of T (S is the set of all subsets of T). We can define a relation $\leq$ on S so that, given $X \epsilon S$ and $Y \epsilon S$, $X \leq Y$ if and only if $X \subseteq Y$. $(S, \leq)$ is a partially ordered set. However, $(S, \leq)$ is not a linearly ordered set if T has at least two elements.

Example 5.5 Let $\Re$ represent the set of real numbers, and let $\leq_\Re$ be the usual order on $\Re$. $(\Re, \leq_\Re)$ is a linearly ordered set.

The first example shows that software document sets can have pre-orders which are neither partial nor linear. Our intuition is that partial orders underly such pre-orders of document sets, i.e., we can find a partially ordered structure to document sets. This intuition is formally stated in the following proposition.

Proposition 5.6 Let $(S, \leq_S)$ be a pre-ordered set. There exists a subset $\tilde{S} \subseteq S$ and a partial order $\leq_{\tilde{S}}$ such that for any $s_1 \epsilon \tilde{S}$ and $s_2 \epsilon \tilde{S}$, $s_1 \leq_{\tilde{S}} s_2$ if and only if $s_1 \leq_S s_2$.

We provide the intuition supporting the foregoing proposition in terms of the directed graph associated with $(S, \leq_S)$. If the ordering $\leq_S$ is not partial, it has cycles of length greater than one edge. In fact, there may be a number of different sets of nodes where each set is such that we can get from any node in the set to any other node in the set. Sets of nodes with this property are called *strongly connected components*. We can construct a subset of S by selecting only one element from each strongly connected component. In other words, each element in S which is not in a strongly connected component of the graph is included in $\tilde{S}$, and for each strongly connected component in S exactly one element is included in $\tilde{S}$. We define $\leq_{\tilde{S}}$ by restricting $\leq_S$ to $\tilde{S}$.

We call $\tilde{S}$ a *derived partially ordered set* and $\leq_{\tilde{S}}$ a *derived partial order*; more precisely, $(\tilde{S}, \leq_{\tilde{S}})$ is *derived* from $(S, \leq_S)$. Although there may be several partially ordered sets derived from the same pre-ordered set, each derived partially ordered set is in some sense equivalent. Any two partially ordered sets derived from the same pre-ordered set can differ only in that a different element may be included from each strongly connected component. ¿From the perspective that the "selected element" is a representative of the strongly connected component, all partially ordered sets that are derived from the same pre-ordered set are equivalent.

The need to work with functions that preserve specific orderings motivates the following definition.

Definition 5.7 Let $(P, \leq_P)$ and $(Q, \leq_Q)$ be pre-ordered sets and let f be a function from P to Q. The function f is said to be order-preserving if whenever $x \leq_P y$ then $f(x) \leq_Q f(y)$.

We denote a function f from P to Q with $f : P \rightarrow Q$. To show that a function operates on pre-ordered sets, we can use $f : (P, \leq_P) \rightarrow (Q, \leq_Q)$.

6 An Analysis of Document Orders

Recall from Section 4 that when we define software document measures we necessarily assume that the set of documents being measured is ordered. Of course, a measure will in fact define a particular order. But we raise the possibility that the aspects of program structure that may be of interest may be represented by orders not defined directly by measures. In other words, if **the** order of a document set with respect to some aspect of structural complexity is defined by a measure, then the measure is in fact the aspect we are seeking.

Currently we do not have such measures. Invariably after publication of a new measure a number of critical letters and articles appear. Criticisms often take one of two forms:

1. the critic provides examples of distinct documents for which the new measure yields the same value and the critic argues that the example programs are not "the same", or

2. the critic provides examples of distinct documents for which the new measure yields distinct values and the critic argues either that they should be the same or that the order implied by the measure is inappropriate.

Both forms of criticism have been used to motivate definitions of new measures. Both forms are based on observations about preferred or assumed orders on the sets of documents.

What follows is an analysis of possible orderings on the set of abstractions of documents based on the mathematical notions of order developed in Section 5:

Pre-order: At an intuitive level, if there is not at least a pre-ordering of the document set, then there is no sense of structure at all. We tacitly assume there exists an ordering $\leq_R$ such that if $x \leq_R y$ and $y \leq_R z$, then $x \leq_R z$. This is a critical point since we are really equating the notion of document structure with transitivity of an underlying order. If there is no transitive ordering, there is no structure to the documents in the document set.

We know, from our preceding discussion of orders defined by an arbitrary measure m that there is a pre-order defined by m. On the intuitive level, there may be a problem with pre-orders that are not partial orders. They may not do a very good job of differentiating distinct documents. Since a pre-order does not have the antisymmetry property, there may be "regions" of the relation [3] in which each document is related to every other document in the region, i.e., the strongly connected components from Section 5. If there are a small number of large regions of this type, then the order does not "say much" about document structure. In such a case, there are only a few groups of documents that are meaningfully distinguished by the order. Some researchers claim that McCabe's cyclomatic complexity suffers from exactly this deficiency [29].

partial order: If the underlying order of the document set is a partial order and not a linear order, then the potential problem associated with the simple pre-order is eliminated. The antisymmetry property of a partial order precludes the regions of distinct documents that all fit "in the same place" in the preorder. However there is another problem associated with partial orders, since, in general, not every pair of documents is comparable under a partial order. (Since every partial order is also a pre-order, this is also a problem for orders of document sets that are just pre-orders.)

This is a fundamental problem for software document measures research. Since a measure m maps the entire set of similar documents to $\Re$, all abstractions of documents are comparable under the measure. But if the underlying order for the document set is a partial order and not a linear order, then the measure has the effect of creating comparabilities. In other words, when we try to define a measure to preserve a partial order, we loose information about which documents are not comparable in the order.

Consider the following informal example. Assume that we construct Pascal procedure bodies by the sequencing and nesting of statements. Given two procedure bodies P_1 and P_2, we have $P_1 \leq P_2$ if and only if starting with P_1 we can derive P_2 through a series of sequencing and nesting operations. (Thus any procedure body consisting of exactly two statements in sequence can be derived from a procedure consisting of either one of the statements.) Clearly there are many procedure bodies which are not comparable under this order. A procedure body consisting of a sequence of two while statements is not comparable with a procedure body that consists of two nested while statements. But any measure which purports to quantify "nesting complexity" will necessarily compare the sequenced and nested while statements.

There are, of course, two possibilities in the the simple informal example: either the measure compares things that are not comparable or we have not identified an appropriate ordering. It is the opinion of the authors that resolving this comparability issue for software document measures will not be easy.

linear order: This would seem to be the ideal. Let $\leq_L$ be the linear order for a particular aspect of structural complexity of documents. Every two items are comparable, so, given a measure m, $m(d_1) \leq_{\Re}$

[3] Consider a directed graph representation of the order R as described in Section 5.

$m(d_2)$ is meaningful for every document d_1 and d_2. However, linear orders may be too discriminating to be of practical value. We could not have $x =_L y$ even when the abstractions of x and y are only minutely different. Assume our similar documents set consists of programs in a given programming language and an abstraction which includes the variable names. A program P_1 and a program P_2 which differ only by the spelling of a single variable name could not be equal with respect to L, i.e., we could not have $P_1 =_L P_2$. Thus a linear order would have the flavor of an exhaustive enumeration of all the documents.

To emphasize the significance of the pre-order assumption for the notion of the structure of documents, we provide the following definition:

Definition 6.1 Let D be a document set with pre-order $\leq_D$ and let $\leq_{\Re}$ be the usual notion of less than or equal to for $\Re$. A *structural complexity measure* is an order preserving function $m : (D, \leq_D) \longrightarrow (\Re, \leq_{\Re})$.

This definition actually suggests a method for software measures research. The definition emphasizes the potential relationship between a measure and an underlying order on the document set. It is simply a formalization of the Assumptions 4.1 and 4.2 given in Section 4. We note again that any measure will itself define a pre-order on the document set. However, the emphasis of the definition is of practical interest. If for a given measure m we can show that m preserves a given order R, where R is not defined directly by m, then we have a demonstrable analytical property of m. The value of the property is of course related to the extent to which m and R are different (so that "not defined directly by m" is not just cosmetically concealed). The value of the property is also related to the extent to which R describes a structural property that has some practical value. In [3] we develop an order for document sets and show that McCabe's cyclomatic complexity preserves the defined order.

7 A Measures Research Methodology

An important point in the preceding Section is that the notion of document structure relies on the existence of pre-orders for the document sets. We also refer to "aspect" of document structure, reflecting the pervasive belief that software document structure is probably multi-faceted. This implies there may well be a number of pre-orders of the document sets that are not everywhere consistent, i.e., two documents may be related in one order but not in another or two documents may be related "oppositely" ($d \leq_1 \tilde{d}$ and $\tilde{d} \leq_2 d$) in different orders. The existence of varied aspects of the structure of programs is apparent. Program size, e.g., as measured by Halstead's volume, is not the same as program control flow complexity, e.g., as measured by McCabe's cyclomatic complexity.

We often define an "aspect" of software documents when we use a particular *abstraction* of the software document. For example, cyclomatic complexity is actually defined for a flowgraph abstraction of the corresponding program. This adds a new component to defining a structural complexity measure: first define the *abstraction* of the document, and then define a measure on the particular abstraction.

Given particular abstractions of software documents, we can raise questions about orderings of the abstractions analogous to those just considered for documents. But first note that any order on the abstraction will define a pre-order on the original document set, as long as there is a well-defined mapping from the original documents to the abstractions.

For example, let the abstraction of programs be simply the sequence of token types, where token types are those that might be returned by a lexical analyzer to a parser. We can define a pre-order on the sequences of token types — a token sequence S_1 is "less than" a token sequence S_2, $S_1 \leq_t S_2$, if and only if S_1 has fewer tokens. This order on token sequences defines a preorder on the set of programs, where a program P_1 with corresponding token sequence S_1 is "less than" program P_2 with corresponding token sequence S_2 if and only if $S_1 \leq_t S_2$. The "less than" order of the set of programs is a preorder since many programs will map to the same sequence of token types.

The consideration of different types of orders on the sets of abstractions leads to a hierarchy of structural complexity measures. Recall that structural complexity measures need only be order preserving functions from a set of documents to the real numbers.

We define a subset of the set of structural complexity measures by requiring that the measure preserve a partial order on a particular abstraction:

Definition 7.1 Let D be a set of similar documents and A be a set of abstractions. Also let *abs*: $D \longrightarrow A$ and $\leq_A$ be a partial order on A. Finally, let $m^\star$ be an order preserving mapping, $m^\star : (A, \leq_A) \longrightarrow (\Re, \leq_\Re)$. Then $m(D) = m^\star(abs(D))$ is a *well-founded structural complexity measure.*

We can define a subset of well-founded structural complexity measures by restricting the nature of the orders on abstractions. In particular, partial orders defined by specific transformations on abstractions can be used in analyzing the properties of structural complexity measures. For example, consider the flowgraph as an abstraction of source programs and the following two transformations on flowgraphs:

Definition 7.2 Given a flowgraph $G = (N, E, s, t)$, then a flowgraph $G' = (N \cup \{x\}, E', s, t)$ is obtained from G by a *node addition transformation (T_1)* if:

1. $x \notin N$,
2. $\exists (y, z)[(y, z) \in E \wedge E' = (E - \{(y, z)\}) \cup \{(y, x), (x, z)\}]$, and
3. the only differences between G and G' are given in 1. and 2.

This first transformation allows a new flowgraph to be generated by adding a node along an existing edge of a given flowgraph.

Definition 7.3 Given a flowgraph $G = (N, E, s, t)$, then a flowgraph $G' = (N, E \cup \{e\}, s, t)$ is obtained from G by an *edge addition transformation (T_2)* if $e \notin E$ and $e = (x, y)$, where $x \in N$ and $y \in N$.

This second transformation allows a new flowgraph to be generated by adding a new edge between existing nodes in a given flowgraph.

Now define $\leq_F$ for flowgraphs G_1 and G_2 so that $G_1 \leq_F G_2$ iff G_2 can be obtained from G_1 by a finite sequence of applications of T_1 and T_2. The set of transformations and the associated partial order provide one perspective on the structural complexity of flowgraphs. In fact, the authors have shown that McCabe's cyclomatic complexity measure preserves this partial order $\leq_F$ [3].

The partial order $\leq_F$ formalizes a notion of *containment* for flowgraphs. A flowgraph G is contained in a flowgraph G' if and only if $G \leq_F G'$. This notion of containment can be generalized to arbitrary document sets and any finite set of transformations. The transformations must define a partial order on each document set and the set of transformations must be constructively complete. The transformations are constructively complete if they are sufficient to construct the entire document set from a given finite set of "initial" documents. The trivial flowgraph with just a start node, a terminal node, and one edge from the start node to the terminal node is the only initial document needed to construct all flowgraphs using Definition 7.2 and Definition 7.3.

Definition 7.4 Let D be a set of similar software documents, A be a set of abstractions where *abs:* $D \longrightarrow A$, *and let:*

1. T be a finite set of relations that map A, and possibly one or more components of the document type, to A,
2. P be a partial order defined on A using T, and
3. T be constructively complete.

Then P is a *containment-based partial order.*

Of course, what really matters here is what measures say about the document set D. It is just that we will most often define the containment-based partial order on some abstraction of the document set. The node in transformation T_1 and the edge in transformation T_2 are examples of "components of the abstraction type". One might want to be more precise about defining "components of the abstraction type". Such a development would necessarily be based on the careful definition of the representation of the abstraction type. We choose not to limit the representation of software abstractions of software documents, or the documents themselves, to sequences (or strings) of characters or some such primitive representation. Thus we allow for the possibility of abstractions and documents that are two dimensional, e.g., data flow diagrams or flowgraphs.

This notion of a containment-based partial order can be used in a general definition of a subset of well-founded structural complexity measures:

Definition 7.5 Let D be a set of similar documents and A be a set of abstractions. Also let *abs*: $D \longrightarrow A$ and $\leq_A$ be a containment-based partial order on A. Finally, let $m^\star$ be an order preserving mapping, $m^\star : (A, \leq_A) \longrightarrow (\Re, \leq_\Re)$. Then $m(D) = m^\star(abs(D))$ is a *containment-based structural complexity measure.*

Perhaps every well-founded structural complexity measure is a containment-based structural complexity measure. One might be able to argue that for each partial order there is an appropriate set of transformations that "define" the partial order. However, the issue here is the extent to which the transformations describe the structure of the abstraction. In [31] Howatt and Baker develop a containment based partial order which can be used to analytically evaluate proposed measures of control flow nesting complexity in program flowgraphs.

The definitions of well-founded structural complexity measure and a containment-based structural complexity measure suggest a methodology for software document measures research. First, in defining a document measure one needs to specify precisely the documents or abstraction of documents for which the measure is defined. If the measure is for an abstraction of documents, it is essential that the mapping from documents to the abstraction be carefully defined. One might then precisely define the "aspect" of document structure to be measured by defining a partial order on the documents, or abstractions of documents. An appropriate measure of this aspect of structure should then be an order preserving mapping from the documents, or abstractions, to the real numbers.

Note that even if one does not define a measure using a particular ordering, measures can be analytically evaluated using various known orders for the documents. The authors are finding value in the methodology both for defining measures and for analyzing existing measures.

As a summary, consider the following method for developing a new structural complexity measure:

1. Specify a set of software documents on which the subsequent steps will be based. While this step seems obvious, it is not always done. Occasionally, restrictions on the set of documents limit the usefulness of measures.

2. Specify abstractions for the selected set of documents. It is important that the set of appropriate abstractions be well-defined. As a simple example, it is important for particular tools and measures that each node in the flowgraph lie on a path from the start node to the terminal node. In such cases the additional requirement should be stated explicitly.

3. Define a mapping from the document set to the set of abstractions. The basic ideas in a tool or measure may be sound and original, but it is equally important that others be able to analyze, and possibly implement, the ideas. When the ideas are based on a particular abstraction of documents, replication of results and general use of the measure or tool depends on having a well-defined mapping to the abstraction.

4. Define an order on the set of abstractions. This is probably the most critical step in the process. The order on the set of abstractions will, in some sense, define the aspect of structure which we are considering. The defined order can also then be compared with other possible orders to see how the current "aspect of structure" is related to other defined aspects. The formal nature of the order will reveal possible pitfalls of structural complexity measures that preserve the order, as described in Section 6.

5. Define an order preserving function from the abstraction to the real numbers. This will, by definition, give us a well-founded structural complexity measure. The order on the abstraction will often suggest the units to be associated with the measure. Using this approach, we should have a good idea of what the measure is actually quantifying and how we might analytically evaluate the measure.

8 Conclusions

The mathematical perspective developed in this paper contributes to the goal of providing a theoretical basis for software document measures research. Earlier efforts to provide this basis have centered on the development of a set of axioms which software document measures ought to satisfy, e.g., [38]. The theoretical

basis developed in this paper provides a more generally applicable hierarchy of properties that a software document measure might satisfy. Our hierarchy includes:

1. software document measure,
2. structural complexity measure,
3. well-founded structural complexity measure, and
4. containment-based structural complexity measure.

Many of the axioms developed in the earlier research efforts (e.g., [19, 38, 49]) are subsumed by the hierarchy given above, as long as one interprets these earlier sets of axioms as applying to structural complexity and not to psychological complexity.

Our mathematical perspective also makes explicit inherent assumptions about the document sets measured — these document sets are ordered. We also stress the importance of abstractions of document sets in developing and evaluating software structural complexity measures. A particular abstraction might be used to show that a given software document measure is a well-founded structural complexity measure. The nature of the abstraction might well provide insight into what is actually being measured (or to demonstrate equivalence of two seemingly different measures).

We also suggest that using abstractions of documents and orders on these abstractions will suggest the units to be associated with a given measure. This should additionally support consideration of various combinations of units as is common in the physical sciences.

The authors are finding that the new mathematical perspective does chart new and productive ground for software document measures research. We are currently making a distinction between abstractions of documents and the measures defined using these abstractions. For example, we are developing data flow abstractions for both imperative [8, 16] and functional programming languages. The data flow equivalents of scope [28], finite execution path [4], and structuredness [32] appear promising for developing testing strategies as well as for measuring the structure of programs. We are continuing to develop containment partial orders which can be used as the basis for analytical evaluation tools of measures of document structural complexity. We also are hopeful that applying the perspective to different types of documents may yield useful structural complexity measures for formal specification and detailed design documents.

In summary, the mathematical perspective on software measures research presented in this paper provides some basic assumptions inherent in the software measures research to date, clarifies what we do and do not measure, and suggests numerous approaches to the analytical evaluation of proposed structural complexity measures.

References

[1] A. J. Albrecht and J. E. Gaffney. Software function, source lines of code, and development effort prediction: a software science validation. *IEEE Trans. Software Engineering*, SE-9(6):639–648, November 1983.

[2] A. Baker, J. Bieman, and P. Clites. Implications for formal specifications – results of specifying a software engineering tool. *Proc. of the IEEE Computer Society's Eleventh Annual International Computer Software & Applications Conference (COMPSAC87)*, 131–140, October 1987, Tokyo, Japan, Computer Society Press.

[3] A. Baker, J. Bieman, D. Gustafson, and A. Melton. Modeling and measuring the software development process. *Proc. of the Twentieth Hawaii International Conference on Systems Sciences*, II:23–30, January 1987, Western Periodicals, North Hollywood, CA.

[4] Albert L. Baker, James W. Howatt, and James M. Bieman. Criteria for finite sets of paths that characterize control flow. *Proc. of the Nineteenth Hawaii International Conference on System Sciences*, IIA:158–163, January 1986, Western Periodicals, North Hollywood, CA.

[5] V. R. Basili and D. H. Hutchens. An empirical study of a syntactic complexity family. *IEEE Trans. Software Engineering*, SE-9(6):664–72, 1983.

[6] V. R. Basili and R. W. Selby. Metric analysis and data validation across fortran projects. *IEEE Trans. Software Engineering*, SE-9(6):652, November 1983.

[7] C. A. Behrens. Measuring the productivity of computer systems development activities with function points. *IEEE Trans. Software Engineering*, SE-9(6):648–652, November 1983.

[8] 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.

[9] R. E. Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal Man-Machine Studies*, 9:737–751, 1977.

[10] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models.* Benjamin/Cummings, Menlo Park, California, 1986.

[11] N. S. Coulter. Software science and cognitive psychology. *IEEE Trans. Software Engineering*, SE-9(2):166–171, 1983.

[12] B. Curtis. Conceptual issues in software metrics. *Proc. of the Nineteenth Hawaii International Conference on Systems Sciences*, 2:154–157, January 1986, Western Periodicals, North Hollywood, CA.

[13] B. Curtis. In search of software complexity. *Proc. IEEE Workshop on Quantitative Software Models*, 95–106, October 1979, IEEE Press, New York, NY.

[14] B. Curtis. Measurement and experimentation in software engineering. *Proc. IEEE*, 68(9):1144–1157, September 1980.

[15] B. Curtis, S. B. Sheppard, and P. Milliman. Third time charm: stronger prediction of programmer performance by software complexity metrics. *Proc. 4th International Conference on Software Engineering*, 356–360, 1979, IEEE Press, New York, NY.

[16] N. Debnath and J. Bieman. A representation and analysis of interprocedural structure. *Proc. of the Twentieth Hawaii International Conference on Systems Sciences*, II:92–100, January 1987, Western Periodicals, North Hollywood, CA.

[17] D. E. Egan and B. J. Schwartz. Chunking in recall of symbolic drawings. *Memory and Cognition*, 7(2):149–158, March 1979.

[18] N. E. Fenton and A. A. Kaposi. Metrics and software structure. *Journal of Information and Software Technology*, 29(6):301–320, July/August 1987.

[19] N. E. Fenton and R. W. Whitty. Axiomatic approach to software metrication through program decompostion. *The Computer Journal*, 29(4):329–339, 1986.

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

[21] A. B. Fitzsimmons. Relating the presence of software errors to the theory of software science. *Proc. 11th Hawaii International Conference on Systems Sciences*, 40–46, January 1978.

[22] R. W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8):455–460, August 1979.

[23] P. G. Frankl, S. N. Weiss, and E. J. Weyuker. ASSET: a system to select and evaluate tests. *Proc. IEEE Conference on Software Tools*, 72–79, April 1985, IEEE Press, New York, NY.

[24] Y. Funami and M. H. Halstead. A software physics analysis of Akiyama's debugging data. *Proceedings of the Symposium on Computer Software Engineering*, 133–138, 1976, Polytecnic Press, Brooklyn, NY.

[25] R. D. Gordon. Measuring improvements in program clarity. *IEEE Trans. Software Engineering*, SE-5(2):79–90, 1979.

[26] R. D. Gordon and M. H. Halstead. An experiment comparing fortran programming times with the software physics hypothesis. *AFIPS Conference Proc.*, 45:935–937, 1976.

[27] M. H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[28] W. A. Harrison and K. I. Magel. A complexity measure based on nesting level. *ACM Sigplan Notices*, 16(3):63–74, 1981.

[29] W. A. Harrison, K. I. Magel, R. Kluczny, and A. DeKock. Applying software complexity metrics to program maintenance. *Computer*, 15(9):65–79, 1982.

[30] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. Software Engineering*, SE-7(5):510–518, 1981.

[31] J. Howatt and A. Baker. On the rigorous definition and analysis of program complexity measures. *The Journal of Systems and Software*, to appear.

[32] J. W. Howatt and A. L. Baker. *A New Perspective on Measuring Control Flow Complexity.* Technical Report TR 85-1, Dept. Computer Science, Iowa State University, Ames, Iowa, 1985.

[33] J. K. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray and M. A. Adler. Software Complexity Measurement. *Communications of the ACM*, 29(11):1044-1050, November 1986.

[34] D. H. Krantz, R. D. Luce, P. Suppes and A. Tvesky. *Foundations of measurement*, Academic Press, 1971.

[35] H. E. Kyburg. *Theory and measurement.* Cambridge University Press, Cambridge, 1984.

[36] B. S. Massey. *Measures in Science and Engineering: Their Expression, Relation and Interpretation.* Ellis Horwood, West Sussex, England, 1986.

[37] T. J. McCabe. A complexity measure. *IEEE Trans. Software Engineering*, SE-2(4):308–320, 1976.

[38] R. E. Prather. An axiomatic theory of software complexity measure. *The Computer Journal*, 27(4):340–347, 1984.

[39] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Engineering*, SE-11(4):367–375, 1985.

[40] W. C. Reynolds. *Thermodynamics.* McGraw-Hill, New York, 1968.

[41] C. Rich and H. E. Shrobe. Initial report on a lisp programmers apprentice. *IEEE Trans. Software Engineering*, SE-4(6):456–467, November 1978.

[42] F. S. Roberts. *Measurement Theory.* Addison Wesley, 1979.

[43] D. Ross. Applications and extensions of SADT. *Computer*, 18(4):25–35, April 1985.

[44] H. Sackman, W. J. Erickson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, 11(1):3–11, January 1968.

[45] L. A. Sena. *Units of Physical Quantities and their Dimensions.* MIR Publishers, Moscow, 1972.

[46] B. A. Sheil. The psychological study of programming. *ACM Computing Surveys*, 13(1):101–120, 1981.

[47] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Engineering*, SE-10(5):595–609, September 1984.

[48] J. M. Stroud. The fine structure of psychological time. *Annals of New York Academy of Sciences*, 138(2):623–631, 1967.

[49] E. J. Weyuker. Evaluating Software Complexity Measures. *IEEE Trans. Software Engineering*, SE-14(9):1357-1365, 1988.