An Axiomatic Approach of Software Functionality Measure

Yuan-Tsong Chen and Murat M. Tanik
Department of Computer Science and Engineering
Southern Methodist University
Dallas, TX 75275

Abstract

In all software life cycle models, it has been recognized that requirements specification is the most important stage [Yeh90]. Since changes made in specifications will have a ripple effect on implementation [Cook91], measuring specifications becomes desirable.

Software functionality measure is a field that remains almost untouched [Fen91, Shep88]. Current specification complexity metrics lag behind program complexity metrics in both number and validity. In this paper, three axioms for functional complexity of software are proposed. The first axiom asserts that functional complexity is a function and computable at the specification level. The second axiom states that an empty specification has zero functional complexity. The third axiom asserts that a harder specification has no less functional complexity than an easier one. With these three axioms and various definitions on the notions of functional complexity, we prove a number of theorems. This axiomatic approach sets up a framework for functionality metrics to be evaluated and helps to define new ones.

Three of the most commonly used code metrics, including statement count, Halstead’s programming effort, and McCabe’s cyclomatic number are reviewed. Also covered are specification metrics including Hellerman’s metric, Schutt’s metric, Coulter’s metric, and Albrecht’s functional point. These metrics are evaluated with respect to the three axioms. Results show that code metrics do not satisfy these axioms, while Hellerman, Schutt and Coulter metrics satisfy these axioms. Albrecht functional point, however, does not satisfy these axioms.

I. Introduction

Providing a higher level of abstraction has been one of the most important characteristics of current software development environments. A change made in the specification/design stage could possibly ripple to affect a large portion of implementations. New software life cycles are thus focused on fast feedback at the specification/design stage and automated tools and environments to exercise designs.

In studying design related phenomena, the following observations are made [Tani92a]:

(1) Human beings cannot correctly specify all levels of a system at once, at the outset of its development. As a result, specifications need continuous refinement.

(2) Rapid feedback promotes the refinements of the specification and design.

(3) Automated tools are required to provide rapid feedback and to handle tasks at which the human mind is not very adept.

Tanik and Chan [Tani91] proposed an Abstract Design Paradigm guided by the above observations. Software development can be viewed as a process of design refinements which can be modeled as a finite automata [Tani92b]. The set of specification states is composed of all the primitive specifications. The input alphabet is the set of atomic refinement activities. The transition function maps a specification state and a refinement activity to a new specification state. An initial state and a final state complete this automata.

In transition from one specification to another, it is desirable to be able to measure the attributes of specifications, with which we can better choose refinement activities. This research lays a framework for such metrics to be developed and evaluated.

Most of the well studied metrics are measures of code. However, it seems that no existing metric is accepted as good by all software engineers. Pros and cons of code metrics are argued in empirical studies [Davi88, Li87, Basi83] as well as in theoretic approaches [Weyu88, Prat84, Fen86]. Probably, it is unlikely that a single metric to serve all people for all software related complexity aspects will be discovered. Rather, a careful selection of metrics may better serve individual needs.

From the study of software life cycles [Boeh88, Tani89], it is known that earlier stages of software development, such as requirements acquisition, are much more important than later ones. This implies that specification metrics are of more importance than code metrics [Shep88]. Studies on measuring specifications are still scarce. Yet, it is believed that metrics for specifications will be the focus of many software engineers.

Research on software metrics suffers one serious problem of not having enough data demonstrating the validity of software metrics [Curt83]. Even if data are collected to justify a metric, it is often arguable how representative the
collected data are. Contradictory data can be found in justifying a metric. It remains collected data are. Contradictory data for different environments should be provided for users ever, we probably should not expect to have a single set of generally accepted empirical data and researchers to choose to verify their metrics.

Although the theoretic approach on software metrics is far less popular than the empirical approach, it is helpful in verifying and defining metrics. Prather Bat84 proposed far less popular than the empirical approach, it is helpful in three axioms for code metrics and verified them on McCabe’s Cyclomatic metric [McC76] and Halstead’s program volume [Hal77]. This work is further explored and strengthened by Fenton and Whitty [Fen86]. Weyuker [Wey98], instead, proposed a list of desirable properties for code metrics and evaluated some popular metrics with respect to these properties. Her results showed that none of these metrics satisfied all the properties. Yet, it is arguable that not all the properties are actually desirable. These works are only applicable at code level. We will take an axiomatic approach for functionality measures on specification metrics in this paper.

II. Functional Complexity of Software Specifications

Definition 1: A specification is a precise and independent description of the desired program behavior [List86].

Remark: An empty specification is null and specifies no useful work.

We will denote specifications in capital letters like \(X, Y\) and \(Z\). We will refer to the functional complexity of a specification as a measure of the functionality that it has. The functional complexity of specification \(X\) is denoted by \(C(X)\). Rather than restrict ourselves to a specific metric, our approach is to be applied to any metrics. We will associate a specification \(X\) with a function \(f_X\) which characterizes \(X\) and maps the input domain of \(X\) onto the output domain. Furthermore, each specification is viewed as a set of subspecifications which are decomposed by system developers.

There is a fundamental problem that we need to face in all phases of the software development life cycle. That is "Is the software product (specification, design or program) what we really want?" This problem can be viewed as the functional equivalence problem.

Definition 2: Functional Equivalence

Two functions \(f\) and \(g\) are said to be equivalent if and only if for any input \(w\), \(f(w) = g(w)\).

Theorem 1: Functional equivalence problem is undecidable [Dav83].

Remark: The difficulty of the functional equivalence problem has been the inherent barrier of many software engineering activities such as testing and requirements acquisition.

It is quite often that we need to solve a problem by transforming it to another one, by using existing implementation or by decomposing it into subproblems. This notion is called reduction as defined below.

Definition 3: Reduction

A specification \(X\) is said to be reducible to \(Y\), denoted as \(X \Rightarrow Y\), if for any input \(w\) of \(X\), \(f_X(w) = h_2(f_Y(h_1(w)))\), for some polynomial time computable functions \(h_1\) and \(h_2\) in a set \(H\) of possible transformation functions, whose functional complexities are zero with respect to a given metric. \(h_1\) and \(h_2\) are called transformation functions.

For example, let \(X\) and \(Y\) be the problems of finding the maximum and minimum of a list of numbers, respectively. Suppose we are using Coulter’s metric, which will be discussed in the next section. To show that \(X \Rightarrow Y\), we can define transformation functions \(h_1(n) = -n\) and \(h_2(n) = -n\). Note that the functional complexities of \(h_1\) and \(h_2\) are zero with respect to the Coulter metric. Let \(w = (z_1, ..., z_k)\) be a list of \(k\) numbers. We have \(h_2(f_Y(h_1(w))) = h_2(f_Y(-z_1, ..., -z_k)) = h_2(\min(-z_1, ..., -z_k)) = \max(z_1, ..., z_k) = f_X(z_1, ..., z_k) = f_X(w)\).

By choosing transformation functions \(h_1\) and \(h_2\) to be identity functions (mapping every input \(a\) to \(a\)), reduction becomes functional equivalence. Therefore, reduction is also undecidable.

Definition 4: Subspecification

Let \(X\) and \(Y\) be two specifications. \(X\) is said to be a subspecification of \(Y\) and \(Y\) is said to be a super-specification of \(X\) if and only if \(X \Rightarrow Y\).

Two specifications \(X\) and \(Y\) are said to be different, if neither \(X \Rightarrow Y\) nor \(Y \Rightarrow X\) holds. Two different specifications are said to be disjoint if there is no non-empty subspecification common to them.

Definition 5: Primitive Specification

\(S\) is a primitive specification with respect to a set \(T\) of existing primitive specifications if there do not exist specifications \(P_1, ..., P_j\) in \(T\) such that \(S = (P_1, ..., P_j)\), where \(j > 0\) is an integer.

Remark: Primitive specifications are those that are not further decomposed. They are the basic units that developers feel comfortable to deal with.

Following immediately from the definition, we have theorem 2 below.

Theorem 2: Two primitive specifications are either equivalent or disjoint.

A large system will most likely be implemented with
Decomposing a system into subsystems is an important activity in software development.

**Definition 6: Decomposition**

Decomposition is the process of breaking a specification $S$ into a list of $m>0$ primitive specifications $(S_1,...,S_m)$ such that $S \equiv (S_1,...,S_m)$ and $\max(C(S_1),...,C(S_m)) \leq C(S)$.

**Remark:** In this definition, we have restricted the specifications to be at most as complicated as the undecomposed one. Note that the integration of the decomposed subsystems can also be a specification.

Following the definitions of reduction and decomposition, we have the following theorem.

**Theorem 3:** Decomposition is undecidable.

We can view requirements acquisition as the process of describing users' intentions in a precise way. This can also be viewed as to test the equivalence of two functions, one corresponding to the users' intention and the other to the written specification. This is also undecidable. System decomposition and requirements acquisition are therefore unlikely to be fully automated. Human beings have to play an important role in these activities.

In the rest of the paper, we will assume there is an oracle to answer the questions of functional equivalence and reduction. In practice, this oracle may be an expert or a testing strategy that deals with these undecidable problems which are unavoidable in software development process.

We can view a specification $S$ as a list of decomposed primitive specifications $(S_1,...,S_k)$ or as a set of distinct primitive specifications $S^D = \{S_1,...,S_l\}$ with $S_i$ occurring $q(S_i) \geq 1$ times, which perform all the functionality of $S$.

For an example of how to use $S^D$ to represent the decomposed primitive specifications of $S$, let's choose $S$ to be the problem of sorting a list of $n$ numbers into ascending order. We can decompose $S$ into $(S_1, S_2, S_3)$, where $S_1$ is the problem of merging two sorted lists into one sorted list and $S_2$ is sorting the first $n/2$ numbers into ascending order and $S_3$ is sorting the remaining $n/2$ numbers into ascending order. It is clear that $S \equiv (S_1, S_2, S_3)$. Since $S_1$ and $S_2$ are the same problem, we can thus define $S^D$ to be $\{Z_1, Z_2\}$, where $Z_1 = S_1$ and $Z_2 = S_2$ with $q(Z_1) = 1$ and $q(Z_2) = 2$.

We believe that the following axioms of functional complexity are essential for any sound functional complexity metric. A metric that satisfies the three axioms will be called a **proper measure**.

**Axioms of Functional Complexity:**

(A1) The functional complexity of a specification $X$ denoted as $C(X)$ is a function and is computable at the specification level.

**Remark:** This axiom asserts that the functional complexity is a function and it is possible to compute at the specification level without knowing the details of implementations. We believe it is essential since a useful measure of functional complexity should be made before actual coding is done.

(A2) $C(\phi) = 0$.

**Remark:** A2 states that the functional complexity of an empty specification is zero.

(A3) For specifications $X$ and $Y$, if $X \equiv Y$, then $C(X) \leq C(Y)$. The relation $\equiv$ is transitive.

**Remark:** A3 asserts that "harder" problems have no less complexity than "easier" ones.

**Theorem 4:** Suppose $X$ is decomposed into $(X_1,...,X_k)$.

Then $\sum_{i=1}^{k} C(X_i) \geq C(X)$.

**Remark:** This theorem implies that the functional complexity increases when we migrate from one abstraction level down to a lower one. It also tells us that we can never solve a problem with some implementation possessing less functionality.

**Theorem 5:** Let $C_1, C_2,..., C_r$ be proper functional complexity measures, where $r>0$ is an integer. Then any positive weighted linear combination of these measures is a proper measure.

**Remark:** This theorem is useful when various functional complexity metrics are to be combined to get a more general metric.

**Definition 7:** Ideal Functional Complexity

The ideal functional complexity of a specification $S$, denoted as $C(S)$, is defined as the functional complexity computed at the specification level where no subspecification is given.

**Definition 8:** Decomposed Functional Complexity

The decomposed functional complexity of a specification $S$ is the functional complexity computed on a set of specifications decomposed from $S$. Specifically, let $S$ be decomposed into a list of primitive specifications $(S_1,...,S_k)$. The decomposed functional complexity of $S$ is defined as

$$C(S) = \sum_{i=1}^{k} C(S_i).$$

**Definition 9:** Distinct Functional Complexity

The distinct functional complexity of a specification $S$ is defined as $C(S^D) = \sum_{S \in S^D} C(S)$.

From the definitions above, the decomposed functional complexity of a primitive specification is the same as its ideal functional complexity. We can also observe that dis-
distinct functional complexity of a system is no greater than its decomposed functional complexity.

Theorem 6: \( C(S) \geq C(S') \) and \( C(S) \geq C(S'') \).

Remark: This theorem shows the relationship among the three proposed functional complexities. For a given problem, the ideal functional complexity is the minimum amount of complexity that we need to conquer in order to solve it. The decomposed functional complexity may exceed ideal functional complexity due to overhead incurred in decomposed subspecifications. Distinct functional complexity reflects the programming effort we need to spend to implement the specification.

Definition 10: Decomposed Intersection
Let \( X = \{X_1,\ldots,X_m\} \) and \( Y = \{Y_1,\ldots,Y_n\} \) be two specifications. Also let \( q(X_i) \) be the number of occurrences of \( X_i \) for \( 1 \leq i \leq m \) and \( q(Y_j) \) be the number occurrences \( Y_j \) for \( 1 \leq j \leq n \). The decomposed intersection \( X \cap Y \) is defined to be \( (Z_1,\ldots,Z_l) \), where \( Z_k = X_i = Y_j \) for any \( k, 1 \leq k \leq l \) and some \( i, j, 1 \leq i \leq m \) and \( 1 \leq j \leq n \) with \( q(Z_k) = min(q(X_i), q(Y_j)) \).

Remark: The notion of decomposed intersection is to capture the common functionality of two specifications in their decomposed forms. This intuitive definition, however, suffers in that there exists two equivalent specifications \( X \) and \( Y \) with different decompositions such that \( X \cap Y \neq X \).

Definition 11: Ideal Intersection
Let \( X = \{X_1,\ldots,X_m\} \) and \( Y = \{Y_1,\ldots,Y_n\} \) be two specifications. Also let \( q(X_i) \) be the number of occurrences of \( X_i \) for \( 1 \leq i \leq m \) and \( q(Y_j) \) be the number occurrences \( Y_j \) \ for \( 1 \leq j \leq n \). The ideal intersection \( X \cap Y \) is defined to be the common subspecification of \( X \) and \( Y \) with largest functional complexity, i.e., max \( C(Z) \).

Remark: Ideal intersection defines the common functionality between two specifications. It is termed ideal in that it represents the largest common functionality possible for two specifications. Note that for two primitive specifications, their ideal intersection is the same as their decomposed intersection.

Theorem 7: For distinct primitive specifications \( X \) and \( Y \), \( C(X \cap Y) = C(X \oplus Y) = 0 \).

Remark: Distinct primitive specifications are those to be implemented individually without reusing any other specifications.

Theorem 8: For two specifications \( X \) and \( Y \), \( C(X \oplus Y) = \sum_{i=1}^{l} q(Z_k) C(Z_k) \leq min(C(X), C(Y)) \), where \( Z_k = X_i = Y_j \) for some \( i, j, 1 \leq i \leq m \) and \( 1 \leq j \leq n \) with \( q(k) = min(q(X_i), q(Y_j)) \).

Remark: The functional complexity of the decomposed intersection of two specifications is less than or equal to the minimum of their decomposed functional complexities.

Theorem 9: For specifications \( X \) and \( Y \), assuming that \( C(X) = C(X) \) and \( C(Y) = C(Y) \), \( C(X \cap Y) \leq C(X \cap Y) \leq min(C(X), C(Y)) \).

Remark: Note that without the restriction \( C(X) = C(X) \) and \( C(Y) = C(Y) \), this theorem will not hold. It is possible that overhead is introduced and \( C(X \oplus Y) > min(C(X), C(Y)) \).

Definition 12: Decomposed Union
Let \( X = \{X_1,\ldots,X_m\} \) and \( Y = \{Y_1,\ldots,Y_n\} \) be two specifications. Also let \( q(X_i) \) be the number of occurrences of \( X_i \) for \( 1 \leq i \leq m \) and \( q(Y_j) \) be the number occurrences \( Y_j \) for \( 1 \leq j \leq n \). The decomposed union \( X \cup Y \) is defined to be \( (Z_1,\ldots,Z_l) \), where \( Z_k = X_i \) or \( Z_k = Y_j \), for any \( k, 1 \leq k \leq l \) and \( 1 \leq j \leq n \) with \( q(k) = max(q(X_i), q(Y_j)) \).

Remark: The notion of decomposed union is to capture the combined functionality of two specifications in their decomposed forms. This intuitive definition, however, suffers in that there exists two equivalent specifications \( X \) and \( Y \) with different decompositions such that \( C(X \oplus Y) = C(X) \).

Definition 13: Ideal Union
Let \( X = \{X_1,\ldots,X_m\} \) and \( Y = \{Y_1,\ldots,Y_n\} \) be two specifications. Also let \( q(X_i) \) be the number of occurrences of \( X_i \) for \( 1 \leq i \leq m \) and \( q(Y_j) \) be the number occurrences \( Y_j \) for \( 1 \leq j \leq n \). The ideal union \( X \cup Y \) is defined to be the super specification of \( X \) and \( Y \) with least functional complexity, that is, min \( C(Z) \).

Remark: Ideal union defines the combined functionality of two specifications. Note that for two primitive specifications, their ideal union is the same as their decomposed union.

Theorem 10: For distinct primitive specifications \( X \) and \( Y \), \( C(X \cup Y) = C(X \oplus Y) = C(X) + C(Y) \).

Remark: The combined complexity of two distinct primitive specifications are the sum of their complexities.

Theorem 11: For two specifications \( X \) and \( Y \), \( C(X \oplus Y) = \sum_{i=1}^{l} q(Z_k) C(Z_k) \geq max(C(X), C(Y)) \), where \( Z_k = X_i = Y_j \) for some \( i, j, 1 \leq i \leq m \) and \( 1 \leq j \leq n \) with \( q(k) = max(q(X_i), q(Y_j)) \).

Remark: Decomposed union of two specifications is at least as complex as either of them.

Theorem 12: For specifications \( X \) and \( Y \), assuming that \( C(X) = C(X) \) and \( C(Y) = C(Y) \), \( max(C(X), C(Y)) \leq C(X \oplus Y) \leq C(X) + C(Y) \).
III. Software Metrics

Metrics are quantitative measures of certain attributes of software artifacts. They are used to measure products, development process, environments, and problem domains. Ever since Halstead [Hals77] proposed software science, software metrics have been criticized as expensive and useless data collection. Yet, now in industry, they are used as a yardstick in management. As pointed out by Mills and Dyson [Mills90] and Henry and Selig [Henry90], the trend of research and practice of this field has transcended the measuring of existing products to predicting the cost of developing a new application. Specification metrics serve this purpose.

A large number of software metrics have been proposed to measure various aspects of software systems. We will survey a few frequently used metrics and evaluate them with respect to the axioms addressed in the previous section.

Code Metrics

Code metrics are measurements of attributes of programs. These are the most intensively studied metrics. Although these metrics are generally easy to compute, they can only be computed at code level.

Statement Count

The most obvious and intuitive measure of software complexity is probably the number of statements. Its simplicity has made it the most widely used software complexity measure. The calculation of number of statements is trivial after the definition of a statement of a programming language is settled.

McCabe’s Cyclomatic Number

McCabe [McC87] defines a complexity measure based on graph theory. In a program flow graph, let \( n \) be the number of nodes, \( e \) be the number of edges, and \( p \) be the number of connected components. The McCabe’s metric is defined to be \( V = e - n + 2p \).

Halstead’s Programming Effort

Software science was introduced by Halstead [Hals77]. Following the notation of Halstead, let \( N_2 \) be the number of distinct operators, \( N_1 \) be the number of operators, and \( N_2 \) be the number of operands. The program volume \( V \) is defined to be \( (N_1 + N_2) \log_2 (n_1 + n_2) \). \( V \) is the minimum possible volume for a given algorithm. Define programming effort \( E = V^2/V^p \). Due to the difficulty in computing \( E \), an approximate formula for \( E \) is given as \( \frac{n_1 N_2 (N_1 + N_2) \log_2 (n_1 + n_2)}{2n_2} \). This will be the effort measure that we use in this paper.

Specification Metrics

Specification metrics are measurements computable at the specification level. There are a number of specification metrics derived from information theory. They are based on information theory developed by C.E. Shannon in 1948 [Shan48, Hamm84]. Originally, this theory deals with fundamental aspects of communication systems, while later on, applications have been placed in several domains. Using this theory to derive software complexity metrics sounds promising.

Consider an experiment with a finite number \( n \) of distinct outcomes with probabilities \( p_1, p_2, ..., p_n \). Before performing the experiment there is uncertainty about which outcome will occur. This can be regarded as the information that will be gained by conducting the experiment. Shannon [Shan48] gave three postulates for quantitative information measure. The first is: if the outcomes are of the same probability, then the information measure is a strictly increasing function of the number of possible outcomes. The second is: the amount of information provided by the result of an experiment is independent of the way in which the experiment is performed. The third one is for the continuity requirement. He showed that the only information measure that satisfies the three postulates is \( H(p_1,...,p_n) = -k \sum_{i=1}^{n} p_i \log p_i \), where \( k \) is a positive constant. It is quite often to take the base of the logarithm to be 2 and the constant \( k \) to be 1. The unit of \( H \) is called bit. The function \( H \) is further shown by Shannon to be the entropy function of statistical mechanics. This theory has been applied in several disciplines.

Hellerman Metric

Viewing a program as a process, \( f: X \rightarrow Y \), Hellerman [Hell72] defined a measure of computation work by partitioning the domain \( X \) into \( n \) domain classes \( X_i \) each of which contains all the points in the inverse image of some point in the range \( Y \). Let the number of elements of a set \( X \) be denoted as \( |X| \). The work of \( f \) is given as \( W = |X| H(p_1,...,p_n) \), where \( p_i = |X_i|/|X| \) and \( H \) is the entropy function in information theory.

This definition came from the idea that a program always has a set of inputs, a set of outputs and assignment of one unique output to an input. This basically is to regard a program as a table lookup process. Given an input in \( X_i \), we check the corresponding \( Y_i \) in a table. Since the probability of an input to be in \( X_i \) is \( |X_i|/|X| \), its information content is
-log(|X|/|I|), which is \( \log(|X|/|I|) \).

**Schutt Metric**

The metric proposed by Schutt [Schu77] is similar to the work function of Hellerman. It is defined as

\[
H = \frac{|X|}{|I|} \log\left(\frac{|X|}{|I|}\right),
\]

which is Hellerman metric divided by \( |I| \).

**Coulter Metric**

In the work [Coul87] by Coulter et. al., each program is associated with a finite set of memory locations \( M \), a finite set of location values \( V \), a problem space \( PS \), which is the set of state descriptions permitted by a problem specification. A state description is a predicate which states whether a location \( m \) takes a value \( v \). Obviously, \( |PS| = 2^{|M|} \). Thus we have, the complexity \( H(PS) = |M| \cdot |V| \).

**Albrecht Functional Point**

Albrecht proposed the following metric for measuring complexity of specifications [Albr83]. To compute functional point, we have to first compute the unadjusted function count \( UFC \), which is defined as \( UFC = \sum (\text{Number of items of variety } i) \cdot (\text{weight}i) \), where \( n \) is the number of different varieties of items. There are five item types defined by Albrecht. They are external inputs, external outputs, inquiries, external files and internal files. Each item type is given a subjective complexity rating of simple, average, or complex. Combining with subjective technical complexity factor \( TCF \), functional point \( FP = UFC \cdot TCF \).

**IV. Examining Software Metrics With Respect To Axiomatic Framework**

**Examining Code Metrics**

Statement count is a function that maps a program to an integer, which is the number of statements. However, it is not computable at specification level. It does not satisfy axiom A1. Axiom A2 is obviously satisfied since a null specification requires no statement to implement.

An example suffices to show the violation of axiom A3. Let \( X \) be the problem of sorting \( n \) integers and suppose it is implemented using AVL-tree sorting algorithm [Ade92, Horo90]. Let \( Y \) be the problem of sorting \( n \) integers implemented by selection sort [Horo90]. It is obvious that \( X \) and \( Y \) are functionally equivalent and thus \( X \approx Y \). The number of statements of \( X \) is more than that of \( Y \) without doubt. Thus axiom A3 is violated.

For similar reasons, McCabe’s Cyclomatic Number and Halstead’s Programming Effort only satisfy axiom A2 and violate A1 and A3.

**Examining Hellerman’s Information-Theoretic Metric**

Hellerman’s metric obviously satisfies axiom A1 in that when a specification is complete, we can always partition the input domain into classes and it is straightforward to compute \( W \). Axiom A2 is also met since a null specification has no input domain.

Let \( X \) and \( Y \) be two specifications with their associated functions \( f_X \) and \( f_Y \), respectively. Suppose that \( X \approx Y \). Partitioning the domain \( D_X \) into \( n \) domain classes \( D_{X1}, \ldots, D_{Xn}, D_{Y} \), each of which contains all the points in the domain \( D_X \) mapping to some point in range \( R_Y \). For \( w_1 \) and \( w_2 \) not in the same domain class, we know that \( f_X(w_1) \neq f_X(w_2) \). Since \( X \approx Y \), \( f_X(w) = h_2(f_Y(h_1(w))) \). The only functions whose complexity is zero with respect to Hellerman’s metric are those which have only one domain class or the function associated with a null specification. \( h_1 \) and \( h_2 \) can only be null specification (which means no transformations are used) if \( H(X) \neq 0 \). If \( H(X) = 0 \), then the two transformations can be either one domain function or null specification. In both cases, \( H(Y) \geq H(X) \) since \( |D_Y| \geq |D_X| \). Thus we conclude that Hellerman’s metric is a proper metric.

**Examining Schutt’s Information-Theoretic Metric**

Schutt’s metric is equivalent to Hellerman’s metric divided by the number of the elements of its domain. It follows from the same argument as Hellerman’s metric that Schutt’s metric is proper.

**Examining Coulter’s Information-Theoretic Metric**

Coulter’s metric is a function mapping from the memory spaces required by a specification to a non-negative integer. It is computable once the specification is clearly specified. Axiom A1 is satisfied.

A null specification requires no memory space. Axiom A2 is also satisfied.

For Axiom A3, we have to look at specification with zero complexity with respect to Coulter’s metric. Obviously, if a specification has non-zero memory locations, then its Coulter measure will be non-zero. Hence, only the null specification has zero complexity. Consider \( X \approx Y \). The two transformation functions can only be null specifications so that their complexities are zero. This means no transformations are used; \( Y \) is used to solve the input of \( X \) directly. Since we shall be able to represent any input and output of \( X, Y \) should have no less memory locations and location values than \( X \). Therefore, \( H(Y) \geq H(X) \) and Coulter’s metric is a proper metric.

**Examining Albrecht’s Function Point**

Albrecht’s functional point satisfies axiom A1 from its definition. It also satisfies axiom A2 clearly. However, an
easily constructed example as that which follows can show that it violates axiom A3. Suppose specification X has two inputs and one output. Let specification Y be the same specification as X except that the two inputs are combined together as one input (a record). It is obvious that $X \equiv Y$ but $C(X) \neq C(Y)$, which violates axiom A3. We therefore conclude that Albrecht's function point is not a proper measure.

V. Conclusion

In this paper we have proposed three axioms for specification metrics. By examining popular metrics with respect to the axioms, we found that code metrics do not satisfy these axioms. Albrecht's functional point for specifications fails to satisfy axiom A3. Hellerman's, Schutts' and Coulter's information-theoretic metrics satisfy these axioms.

With the formalism of various notions of specifications, we are able to define the intersection of two specifications, from which reuse measures can be derived. Union of two specifications can be used to find the combination of two specifications.

Future work will be to extend this approach on software reuse and to examine these axioms on more metrics. Combining different metrics to get a more general measure in specification will also be attempted.

References