Complexity Measure Evaluation and Selection

Jeff Tian
IBM PRGS
Toronto Laboratory
North York Ontario, Canada
and
Marvin V. Zelkowitz
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

April 16, 2003

Abstract

A formal model of program complexity developed earlier by the authors is used to derive evaluation criteria for program complexity measures. This is then used to determine which measures are appropriate within a particular application domain. A set of rules for determining feasible measures for a particular application domain are given, and an evaluation model for choosing among alternative feasible measures is presented. This model is used to select measures from the classification trees produced by the empirically guided software development environment of Selby and Porter, and early experiments show it to be an effective process.

Index terms: Program complexity, program measurement, measure evaluation, selection criteria, optimization

1 Introduction

A goal for software engineering is the generation of high quality software, and the role of measurement is to be able to determine when that attribute has been achieved. With program complexity, we want to predict those *external* properties of a program, such as reliability, understandability, and maintainability, through more tangible *internal* measures that capture concrete aspects of a program, such as control flow complexity (e.g., McCabe's cyclomatic number [12]), volume (e.g., statement count and Halstead's software science measures [11]), or information content (e.g., prime program complexity [1]). In this paper we consider various criteria which aid the selection of internal complexity measures to predict those external properties.

Internal measures depend only on the program that is subjected to study, while external measures may depend on other properties [8, 9]. For example, the external measure *understandability*

depends on both the program and the observer trying to understand it. Given two programs, one straight line without branches and the other structured, a programmer might more easily understand his own poorly structured one than a more structured version written by someone else [17].

While external measures have the most impact on the software development cycle, they are harder to quantify than internal ones. As a result, research strives to derive good internal measures [11, 12] and to establish correlations between internal and external measures by empirical study [2, 4, 15]. However, there is no systematic way to use these studies to aid the selection process.

One approach has been to develop formal axiomatic models of complexity [10, 13, 14, 18, 20, 22]. In our model [18] we differentiate between complexity being an inherent property between two programs and a measure being an approximation of this property. We base the selection criteria presented here on this formal model.

In this paper, we first summarize our earlier complexity model as a set of testable conditions on potential measures in order to determine feasible measures. We then provide evaluation functions for choosing among the feasible measures. Finally we apply this selection process to Selby and Porter's empirically guided software development environment [16] to show improvement on the predictive behavior of this system by selecting measures based on our criteria.

1.1 Model Overview

In our model of axiomatic program complexity qualitative comparisons are made through the use of complexity *rankings*, which define pair-wise relationships on programs. To approximate those rankings, we use *measures* to map programs into real numbers.

The axioms summarize essential common properties. They define what programs should be comparable and what universal characteristics these rankings and measures must satisfy. A classification scheme describes relevant subclasses of complexities, sorting out commonalities and differences among them.

If we view measures as points in a measure space, the axiomatic requirements prescribe the boundary conditions of the feasible region where measures are to be selected. Usually there are multiple measures falling inside the feasible region (by satisfying all the boundary conditions). To select among them, a set of scaling functions is constructed by assessing relative effectiveness of aspects of the measures to compute a dominance relation between measures. For example, if we want to maximize values of a vector of measures, a simple vector comparison using "$\leq$" can effectively determine the dominance relation, e. g., if $A \leq B$, measure A is eliminated.

Due to the multi-dimensionality of measure evaluation, there are usually multiple measures where no dominance relation holds among them. Therefore we need to select among the remaining measures using some objective function. The objective function is formulated by evaluating the relative importance of and trade-off among scaling dimensions, assigning appropriate weights to them, and computing the weighted sums for the remaining measures. The final selection is done by comparing these weighted sums.

1.2 Notational Conventions

In this paper, the following notation will be used.

- The term *program* means either a complete program or program fragment.

- A function $\mathcal{V}$ defined on a set of programs will have domain $\mathcal{D}(\mathcal{V})$. P and Q will represent programs in $\mathcal{D}(\mathcal{V})$, i.e. $(P, Q \in \mathcal{D}(\mathcal{V}))$. We will usually omit '$\in \mathcal{D}(\mathcal{V})$' as being understood.
- A program P can be represented graphically by an abstract syntax tree, $AST(P)$.
- For programs P and Q, $IN(P,Q)$ is true if P is a subprogram of Q. (i.e., $AST(P)$ is a subtree of AST(Q).)
- If $IN(P,Q)$ is true, then $dist(P,Q)$ represents the number of edges (i.e., number of nodes+1) between the root of $AST(P)$ and the root of $AST(Q)$.
- The cardinality of a set $\mathbf{S}$ is denoted as $|\mathbf{S}|$. $\mathcal{D}(\mathcal{V})$ is assumed to be countably infinite.
- $\Re$ is the set of real numbers; $\mathcal{N}$ is the set of natural numbers.
- We assume a set of properties (axioms) that valid complexity measures must possess. We call these *boundary conditions* as they define the feasible region when individual measures are viewed as points in the (multi-dimensional) measure space. We label the ith boundary condition as $\mathbf{BC}_i$. If a measure $\mathcal{V}$ satisfies $\mathbf{BC}_i$, we denote it as $\mathbf{BC}_i(\mathcal{V})$.

2 The Feasible Region for Complexity Measures

2.1 Basic Axioms

One major use of complexity information is to choose among functionally equivalent solutions to specific problems. For any functionally equivalent program set, an acceptable measure must be defined for all or none of the programs in it. Therefore:

$\mathbf{BC}_1$. An acceptable measure must compare between functionally equivalent programs:

$$\mathbf{BC}_1\text{: } (\forall P,Q)(Functionality\ of\ P = functionality\ of\ Q) \wedge P \in \mathcal{D}(\mathcal{V}) \Rightarrow Q \in \mathcal{D}(\mathcal{V})$$

Software development is an incremental process. Effective measures must compare component and composite programs. Therefore:

$\mathbf{BC}_2$. An acceptable measure must compare between component and composite programs:

$$\mathbf{BC}_2\text{: } (a) : (\forall P,Q)(IN(P,Q) \wedge P \in \mathcal{D}(\mathcal{V})\ ) \Rightarrow Q \in \mathcal{D}(\mathcal{V})$$
$$(b) : (\forall P,Q)(IN(P,Q) \wedge Q \in \mathcal{D}(\mathcal{V})\ ) \Rightarrow P \in \mathcal{D}(\mathcal{V})$$

When a complexity measure is defined over all programs, $\mathbf{BC}_1$ and $\mathbf{BC}_2$ are trivially true:

Theorem T1: *Given measure* $\mathcal{V}, (\forall P) P \in \mathcal{D}(\mathcal{V}) \Rightarrow (\mathbf{BC}_1(\mathcal{V}) \wedge \mathbf{BC}_2(\mathcal{V}))$

Monotonicity of Program Composition

In general a composite program is more complex than any of its components. However, there are exceptions when contextual information may help reduce such complexity. For example, an *if-then-else* statement may be easier to understand than either the *then* or *else* component alone. However, as programs get larger, the general trend must be followed otherwise infinitely large programs will paradoxically have low complexity. Therefore:

BC$_3$. An acceptable measure must assign sufficiently large values to sufficiently large programs:

$$\mathbf{BC}_3\text{: } (\exists k)(\forall P, Q)\ (dist(P,Q) > k) \Rightarrow \mathcal{V}(P) \leq \mathcal{V}(Q)$$

For any measure $\mathcal{V}$, there is some constant k such that non-monotonicity of two programs is limited by a distance of at most k in the abstract syntax tree (See Figure 1). For example, $\frac{1}{n}$, where n is a measure of program size (e.g., the number of lines), cannot be a complexity measure because it violates this condition.

In our earlier paper, boundary condition **BC$_4$** differentiated between a complexity ranking and a measure approximating that ranking. This is a crucial property in our evaluation criteria, and we will return to this after we describe the next condition.

Discriminating Power of Measures

An acceptable measure must assign different values to different programs sufficiently often. For example, the knot count [21] for any structured program is always 0 which would make it an inappropriate measure for a program universe consisting exclusively of structured programs [20]. In what follows, we consider the mapping function $\mathcal{V}$ to regions of size δ (called a δ-region). Therefore:

BC$_5$. No δ-region around any point v can totally dominate a measure in the sense of having almost all values clustered around this region, i.e.,

$$\mathbf{BC}_5\text{: } (\forall k \in \Re)(\exists \delta > 0)(\exists S)\ |S|\ is\ infinite\ \wedge\ P \in S\ \Rightarrow\ \mathcal{V}(P) \notin [k-\delta, k+\delta]^1$$

If $p_v = prob(\mathcal{V}(P) = v)$ is the probability of having value v, we can define the probability of a given program P having a value within this region as:

$$prob(\mathcal{V}(P) \in [k-\delta, k+\delta]) \ = \sum_{k-\delta \leq v \leq k+\delta} p_v$$

This distribution, which defines the *probability mass function* (*pmf*), is described by the set of pairs $\{\langle v,\ p_v\rangle\}$. The projections of $\{\langle v,\ p_v\rangle\}$ yield $\{v\}$, the set of points on the measurement scale which have some program with that complexity, and $\{p_v\}$, the probability bag[2].

In the special case that the universe of programs $|\mathbf{U}|$ is infinite but $|\{\langle v,\ p_v\rangle\}|$ finite, the condition $\sum p_v = 1$ makes **BC$_5$** reduce to the requirement that $p_v \neq 1$, or equivalently, $p_v < 1$.

Lets return to **BC$_4$**. Since we are only concerned about feasible measures, we want all measures $\mathcal{V}$ to agree with boundary condition **BC$_4$**:

[1] Original model used following probability definition: $(\forall k \in \Re)(\exists \delta > 0) prob(\mathcal{V}(P) \in [k-\delta, k+\delta]) < 1$

[2] $\{p_v\}$ is a bag instead of a set because there might be multiple points v with the same p_v.

$\mathbf{BC}_4$. An acceptable measure agrees with its ranking.

$$\mathbf{BC}_4\text{: } (\forall P, Q)\ \mathcal{R}(P, Q) \Rightarrow \mathcal{V}(P) \leq \mathcal{V}(Q)$$

Where $\mathcal{R}(P, Q)$ defines a ranking between programs P and Q, with P no more complex than Q.[3]

This is an essential property that we wish to be true for each measure that ranks the complexity between any two programs. It clarifies when we can apply measurement theory to a candidate measure. If there is a ranking between programs P and Q, we want to assume that our candidate measure mimics this property.

Unfortunately, $\mathbf{BC}_4$ is undecidable (e.g., let $\mathcal{R}(P, Q)$ exist if Turing machine P halts in fewer steps than Turing machine Q). Therefore, we will weaken it somewhat, and state that a measure is a feasible complexity measure if it obeys the other boundary conditions. In other words:

$\mathbf{BC}_{4'}$. Feasible complexity measures satisfy our boundary conditions.

$\mathbf{BC}_{4'}$: *A complexity measure $\mathcal{V}$ is a measure which satisfies conditions* $\mathbf{BC}_1, \mathbf{BC}_2, \mathbf{BC}_3$ *and* $\mathbf{BC}_5$.

2.2 Classification of Measures

Boundary conditions $\mathbf{BC}_1$ through $\mathbf{BC}_5$ provide minimal requirements on measuring complexity. However, there are constraints imposed by external factors. For example, if a measure is needed to assess the effects of various indentation rules on program comprehension, a control flow measure such as cyclomatic number which ignores presentation characteristics is obviously a wrong choice. As a result, a classification scheme defines a boundary condition used to reject inappropriate measures.

Complexity may be required from a measure that is either: a *vertical* classification which depends on the computational model upon which the measure is defined; or a *hierarchical* classification which deals with the complexity relationship in composite-component programs pairs:

$$\text{Vertical} \begin{cases} \text{Abstract} \begin{cases} \text{Functional} \\ \text{Non Functional} \end{cases} \\ \text{Non Abstract} \end{cases} \qquad \text{Hierarchical} \begin{cases} \text{Context Free} \begin{cases} \text{Primitive} \\ \text{Non Primitive} \end{cases} \\ \text{Interactional} \end{cases}$$

Measure Classification

- *Abstract* measures depend only on abstract syntax trees of the program which contain semantic information but not features such as comments or blank spaces; *non-abstract* measures depends on both (i.e., $(\exists P, Q) AST(P) = AST(Q) \wedge \mathcal{V}(P) \neq \mathcal{V}(Q)$).

- *Functional* measures are invariant to object (e.g., variable) renaming where programs with isomorphic abstract syntax trees have the same complexity (i.e., $(\forall P, Q) AST(P) = AST(Q) \Rightarrow \mathcal{V}(P) = \mathcal{V}(Q)$). *non-functional* measure are not invariant to renaming (i.e., $(\forall P, Q) P \neq Q \wedge AST(P) = AST(Q) \Rightarrow \mathcal{V}(P) \neq \mathcal{V}(Q)$).

[3] The implication is explicitly not bi-directional in this axiom. If there exists a numerical comparison between two programs, there may not necessarily be a ranking between these programs.

- A measure is *context free* if a program has the same complexity regardless of where and how it is used; otherwise it is *interactional* (i.e., $(\exists P, P')IN(P,Q) \wedge IN(P',Q') \wedge P = P' \wedge Q \neq Q' \Rightarrow \mathcal{V}(P) \neq \mathcal{V}(P')$);

- A measure is *primitive* if two programs which consist of the same elements must have the same complexity regardless of the structure (i.e., $(\forall X)(IN(X,P) \Leftrightarrow IN(X,Q)) \Rightarrow \mathcal{V}(P) = \mathcal{V}(Q)$); otherwise it is *non-primitive.*

The analysis carried out before measure selection determines what type of measures are to be used. This information defines our *target* class, which is the broadest class where all the requirements are satisfied.

For example, if the target class is non-abstract, abstract measures are rejected since the assessment is impossible (e.g., the effect of use of blank lines on program comprehension). On the other hand, if an abstract measure is required, non-abstract ones are potentially valid because they depend on both the AST and presentational features. Their suitability depends on whether they contain certain information, which is beyond the use of our classification scheme.

We presented the partitions of the classification in a specific order. The second (lower) subclass is *stronger* than the first (upper) in the sense that more information is needed to compute measures in the former subclass. We use this to differentiate among the feasible measures identified by $\mathbf{BC}_{4'}$:

$\mathbf{BC}_6$**:** All measures from a class weaker than the target class are rejected.

All acceptable measures must satisfy all the boundary conditions. We use these to screen out unacceptable measures.

Example 1: Statement counts

Consider all non-null Pascal programs. Define complexity measure S_1, as $S_1(P)$ being the statement count of program P, and a related statement density measure S_2 as:

$$S_2 = 1 - \frac{1}{S_1(P)}$$

S_2 is an attempt to provide a normalized size for Pascal programs, with trivial one statement programs measuring as 0 and very long programs approaching 1. Evaluating these according to our boundary conditions yields:

- $\mathbf{BC}_1$ and $\mathbf{BC}_2$ are both satisfied by S_1 and S_2 because both measures are valid on all programs (Theorem **T1**);

- $\mathbf{BC}_3$ is satisfied because both S_1 and S_2 are strictly monotonic. When program P is a component of program Q, we know that Q has more statements than P, thus $S_1(P) < S_1(Q)$. Similarly, It is easy to show that $S_2(P) < S_2(Q)$ which satisfies $\mathbf{BC}_3$ for S_2;

- $\mathbf{BC}_5$ (and therefore $\mathbf{BC}_{4'}$) is satisfied by S_1 but not by S_2. Any region of finite length on measurement scale S_1 contains only a finite number of programs and will satisfy $\mathbf{BC}_{4'}$. However, in the case of S_2, given any $\delta > 0$, we have

$$|S_2(P) \notin [1-\delta, 1+\delta])| \; is \; finite$$

violating $\mathbf{BC}_5$, since the number of programs that have measure values between 0 and $1-\delta$, which is the number of programs of finite length (satisfying $S_1(P) < \frac{1}{\delta}$), must be finite.

The density measure S_2 is rejected due to the violation of $\mathbf{BC}_5$, while the statement count measure S_1 is potentially acceptable. Whether S_1 remains acceptable depends upon what target classification is needed.

Example 2: Using classification axiom

If we wish to investigate the relationship between programming comprehension and program documentation, we need to use a non-abstract measure (i.e., the target class is non-abstract). Alternatively, if we are interested in assessing object program size, due to the fact that comments and blank lines are ignored when compiled, abstract class is the target class.

Since the non-abstract class is stronger than the abstract class, boundary condition $\mathbf{BC}_6$ indicates that all measures from the abstract class, such as cyclomatic number which contains no information on program documentation, will be discarded. In the non-abstract class, we can further infer that:

1. A total line count (including all physical lines) might be acceptable because it is correlated with our size assessment (e.g., previous Example 1);

2. A blank line count is rejected because it is irrelevant.

On the other hand, if programming effort is to be predicted, measures from both the abstract and non-abstract classes need to be used. This is because both the executable lines as well as comments, blank lines and indentations used to document the program consumes programming effort.

3 Evaluation Scales for Complexity Measures

The boundary conditions so far provide feasible acceptance criteria. However, to compare multiple acceptable measures, we need evaluation scales. Consider two measures of program paths: one has 2 clusters of trivial (sequential code only) or non-trivial (other control structures used) programs; another measures up to N clusters (e.g., some measure of loop structure). Both may be valid according to $\mathbf{BC}_5$, but we suspect that the latter tells us more about program structure.

Multi-valued scaling has been used to evaluate software designs [6] where a triple $\langle B,\ S,\ W\rangle$ defined the basic requirement B, the scaling functions S for candidate solutions, and weight W of various attributes. We follow a similar approach to evaluate the suitability of remaining complexity measures after the elimination of non-feasible ones.

Boundary conditions $\mathbf{BC}_1$ and $\mathbf{BC}_2$ are intrinsically binary predicates, thus there is no scaling associated with them. However, the distribution associated with $\mathbf{BC}_5$ and the proportion of programs holding monotonic properties associated with $\mathbf{BC}_3$ are multi-valued quantities, naturally leading to evaluation scales.

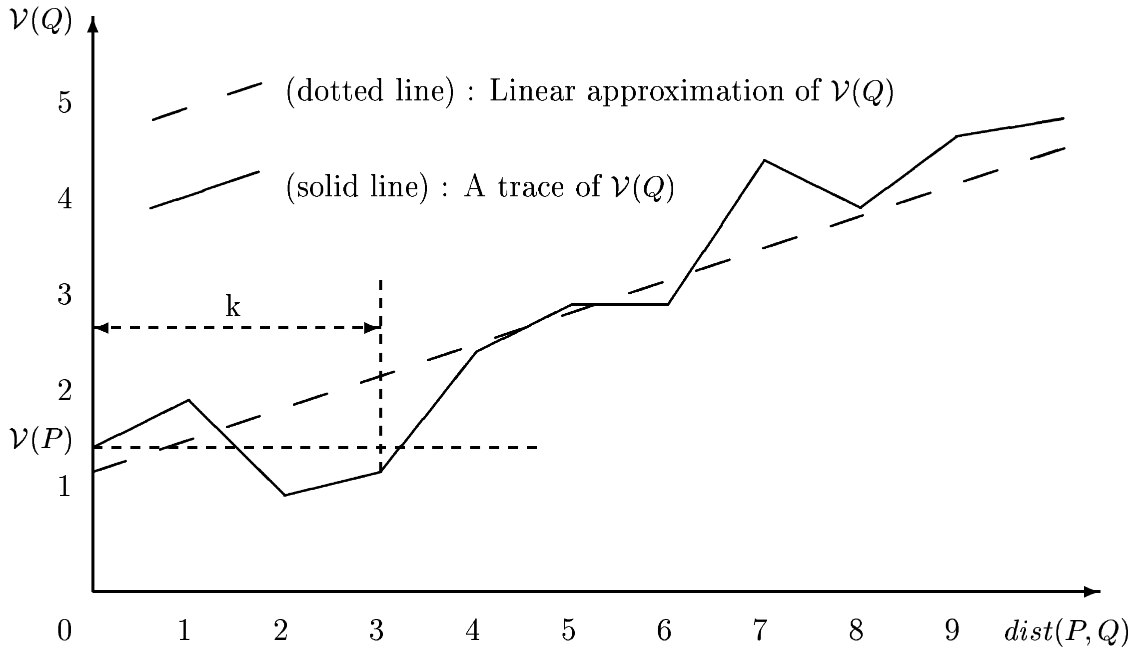

Figure 1: Monotonicity of composition

3.1 Evaluating the Monotonicity of Measures

Boundary condition $\mathbf{BC}_3$ states that no measures should continuously assign lower values to successively larger programs. This allows local deviations from monotonicity but requires a general monotonic trend as in Figure 1. If programs are produced using some incremental or cumulative development method where restructuring is usually avoided, a monotonic measure might be able to predict the size and the development cost more accurately than a non-monotonic one.

Once the monotonicity of the underlying property is known, we can compare measures in terms of their conformance. To compare quantitatively, we define our scale to capture both the *extent* and *frequency* of non-monotonic deviations to the general trend of monotonicity.

Definition: The period of monotonicity T, which captures the extent of non-monotonicity, is the minimum k ($T = \min\{k\}$) such that monotonicity property $\mathbf{BC}_3$, $(\forall P, Q)$ ($dist(P, Q) > k \Rightarrow \mathcal{V}(P) \leq \mathcal{V}(Q)$), is satisfied.

Definition: The period of strong monotonicity S is the minimum k ($S = \min\{k\}$) such that strong monotonicity, $(\forall P, Q)$ ($dist(P, Q) > k \Rightarrow \mathcal{V}(P) < \mathcal{V}(Q)$), is satisfied.

$\mathbf{BC}_3$ must to be true if k is greater than or equal to T, or:

$$(\forall k \geq T)(\forall P, Q)(dist(P, Q) > k \Rightarrow \mathcal{V}(P) \leq \mathcal{V}(Q))$$

The higher the period of monotonicity, the greater the extent of non-monotonic deviations. When $T = 0$, the measure in question is strictly monotonic.

For example, McCabe's cyclomatic number is monotonic (period 0), but not strongly monotonic, since adding statements may not change the cyclomatic number (e.g., adding sequential statements).

It has no period of strong monotonicity ($S = \infty$), because one can add sequential statement indefinitely without changing its cyclomatic complexity. On the other hand, statement count and prime program complexity [1] are both strongly monotonic of period 0.

3.2 Evaluating the Discriminating Power of Measures

Among the many measures that satisfy that condition $\mathbf{BC}_5$, some may possess a better distribution than others according to some criteria such as predictive power, value of information or simplicity. Consider two module data complexity measures: 1) one binary-valued measure with complexity 0 for modules without any variables and complexity 1 for all others; and 2) a measure which returns the unique variable count (η_2 of software science [11]). We would assume that the latter provides more information than the former. Elaborating boundary condition $\mathbf{BC}_5$ leads to a scale to evaluate how well a measure facilitates comparison by assigning appropriate complexity values to programs.

When the distribution of the underlying property that we are trying to predict or control is known, a good measure should correlate closely with this property, possessing a distribution as close to this underlying distribution as possible. We need a scale to assess the proximity between these two distributions.

In the following, we assume that the underlying property has a uniform distribution. Therefore, a measure that gives a uniform distribution with maximal distinguished points would be ideal.

We are interested in knowing that two points on the measurement scale $\mathcal{V}$ are different from each other so that comparisons can be made. How much they differ is of less interest to us, since we use such values to rank complexity between programs rather than compute their absolute value. For example, the horizontal distances in Figure 2 are not important, only their relative positions. Our evaluation scale of measure distribution should only depend on the bag of probability $\{p_v\}$ rather than the whole *pmf* defined by the set $\{\langle v,\ p_v\rangle\}$(see Section 3.1).

We need scales similar in concept to the classical definition of variance or skewness [7]. However, they cannot be directly used to capture what we want. First both are sensitive to values of complexity in addition to the probability associated with these values. To overcome this shortcoming, a transformation has to be carried out to get a point rather than value distribution (which is equivalent to an equal distance value distribution). In addition, they impose order on the original (orderless) $\{p_v\}$. For example, two measures $\mathcal{V}_1$ and $\mathcal{V}_3$ in Figure 2 have distributions: $\{\langle v_1^1, 0.6\rangle, \langle v_1^2, 0.2\rangle, \langle v_1^3, 0.2\rangle\}$ and $\{\langle v_3^1, 0.2\rangle, \langle v_3^2, 0.6\rangle, \langle v_3^3, 0.2\rangle\}$ respectively, resulting in $\mathcal{V}_1$ having larger variance than $\mathcal{V}_3$ although they have the identical bags $\{p_v\}$. But, for our purpose of comparison among programs, both are equally good.

Skewness evaluates the symmetry of a measure distribution. For example, a single cluster (by itself symmetric) is not as good as a distribution with $2n + 1$, where $n \geq 1$, symmetric peaks, although they both have skewness 0.

Uniformity Scale

We define our scale to capture: 1) the number of distinguished points on the measurement scale (e.g., 4 for $\mathcal{V}_2$ and 3 for $\mathcal{V}_1$ and $\mathcal{V}_3$ in Figure 2); and 2) the uniformity of these points. We define a general scale to evaluate the desirability of distributions:

Distribution Scale DS_1 : For any given measure $\mathcal{V}$, region length $\delta > 0$ and probability threshold

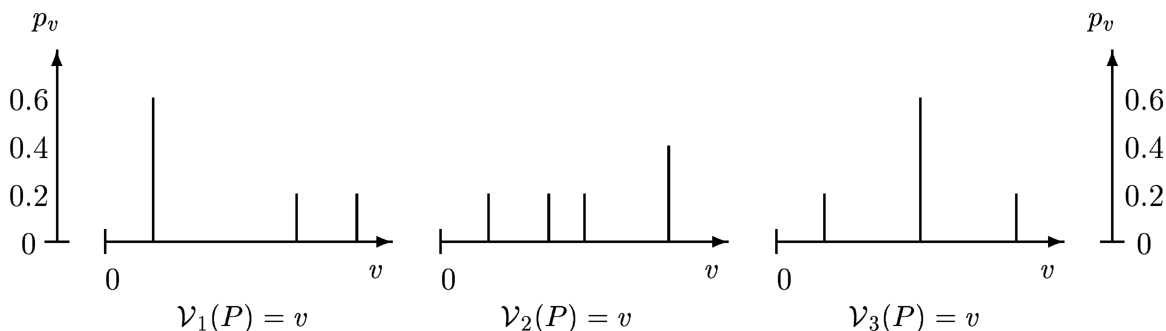

Figure 2: Distributions of complexity values

$\epsilon > 0$, the scale DS_1 is defined to be the pair $\langle n, d\rangle$ where n is the number of regions satisfying:

$$p_k \;=\; prob(k\delta \leq \mathcal{V}(P) < (k+1)\delta) \;>\; \epsilon$$

Or equivalently, n is the cardinality of probability bag [4] $\{p_k \mid p_k > \epsilon\}$. And:

$$d = \begin{cases} 0 & \textit{if } n = 0 \\ n\left(\sqrt{\frac{\sum_k (\frac{1}{n} - p_k)^2}{n}}\right) = \sqrt{\frac{\sum_k (1 - np_k)^2}{n}} & \textit{otherwise} \end{cases}$$

which is the normalized standard deviation of the probability bag $\{p_k \mid p_k > \epsilon\}$.

Notice that $\frac{1}{n}$ is the probability of getting any v if it is uniformly distributed. It is also the mean of the bag $\{p_k\}$. Thus, the standard deviation d of the bag gives us a good estimate of how different this is from a uniform distribution. Finally, it is normalized by scaling the bag $\{p_k\}$ by n (which is equivalent to setting the mean to 1).

In the case where $|\{\langle v, p_v\rangle\}|$ is finite, the above scale computation can be simplified, with $\epsilon = 0$ and δ to be the shortest distance between two successive points with non-zero probability, i.e.,

$$\delta \;=\; \min_{P,Q} |\mathcal{V}(P) - \mathcal{V}(Q)|$$

In this case, any region $[k\delta,\ (k+1)\delta)$ contains at most one point with non-zero probability, thus the number of non-zero probability points is the same as the number of non-zero probability regions. We only count the points with non-zero probability and calculate the normalized standard deviation (d) of the probability bag $\{p_k \mid p_k > \epsilon\}$ of these points, — a much reduced computational task.

To use the scale DS_1 for measure selection, holding everything else equal, the measure with maximized n **and** minimized d will be selected. The dominance relation condition between two measures $\mathcal{V}_1$ with $\langle n_1, d_1\rangle$ and $\mathcal{V}_2$ with $\langle n_2, d_2\rangle$ is defined to be: $(n_1 \geq n_2) \wedge (d_1 \leq d_2)$ but not both "=" hold. The dominated one will be eliminated.

[4] $\{p_k\}$ is a bag instead of a set because there might be multiple regions $[k\delta, (k+1)\delta)$ such that p_k is the same.

Example 3: Measure Evaluation

Consider three measure $\mathcal{V}_1$, $\mathcal{V}_2$, and $\mathcal{V}_3$ with the probability distributions shown in Figure 2. The evaluation using scale DS_1 is summarized below:

	$\mathcal{V}_1$	$\mathcal{V}_2$	$\mathcal{V}_3$
n	3	4	3
d	0.7483	0.3464	0.7483

To illustrate the computation of d, we consider that for $\mathcal{V}_2$, $n = 4$, and the bag $\{p_k\} = \{0.2, 0.2, 0.2, 0.4\}$. Therefore:

$$d = \sqrt{\frac{(1 - 4 * 0.4)^2 + 3 * (1 - 4 * 0.2)^2}{4}} = 0.3464$$

In this example although $\mathcal{V}_1$ and $\mathcal{V}_3$ have different distributions of complexities, the $\langle n, d \rangle$ pairs are identical due to the identical bags $\{p_v\}$.

Assuming everything else equal, $\mathcal{V}_2$ will be selected over $\mathcal{V}_1$ and $\mathcal{V}_3$ by scale DS_1 due to the dominance relation holding among them. This conforms with the picture, that $\mathcal{V}_2$ possesses more evenly distributed complexity values (as captured by $d(\mathcal{V}_2)$) over a larger set of distinguished points (4 vs. 3).

3.3 Selection of Multiple Measures

Some preliminary selection can be made using dominance relations. We say a measure $\mathcal{V}_1$ dominates another measure $\mathcal{V}_2$ if $\mathcal{V}_1$ is evaluated to be better than or as good as (but not all equal) $\mathcal{V}_2$ on every relevant scaling function. However, under most situations, dominance relation holds for only a very few measures. There may be conflicting results by different scales, because program complexity and complexity measures are inherently multi-dimensional. The scaling functions derived to evaluate individual aspects of complexity measures may not be adequate for the overall evaluation and the selection of complexity measures. In what follows, we describe a general evaluation and selection technique, taking into account all those individual evaluations by different scales as well as the interactions and trade-off among them.

Scaling Vector and Measure Elimination

If $\mathcal{V}$[i] is evaluated to be better than $\mathcal{V}$[j] using scale $\mathbf{S}_k$, as long as $\mathcal{V}$[i] is no worse than $\mathcal{V}$[j] under scales other than $\mathbf{S}_k$, we can conclude that $\mathcal{V}$[i] is better than $\mathcal{V}$[j]. This global dominance relation can be formally formulated, using the global scaling vector $\mathcal{G}$, as follows:

Global Scaling Vector: Let $dim(X)$ be the dimensionality of vector X. The dimensionality of the global scaling vector $\mathcal{G}$ is defined on relevant scales $\{\mathbf{S}_j\}$ as:

$$dim(\mathcal{G}(\mathcal{V})) = \sum_j dim(\mathbf{S}_j(\mathcal{V}))$$

The i-th element of $\mathcal{G}(\mathcal{V})$, $\mathcal{G}(\mathcal{V})[i]$, is defined successively as:

$$\mathcal{G}(\mathcal{V})[i] = \begin{cases} \mathbf{S}_j(\mathcal{V})[k] & \text{if maximizer} \\ -\mathbf{S}_j(\mathcal{V})[k] & \text{if minimizer} \end{cases}$$

until all individual scaling dimensions $\mathbf{S}_j(\mathcal{V})[k]$ are exhausted. $\mathbf{S}_j(\mathcal{V})[k]$, the k-th element of scaling $\mathbf{S}_j$, is a maximizer if larger values are more desirable, and a minimizer if the opposite is true.[5]

Dominance Relation: A measure $\mathcal{V}_i$ is said to dominant another measure $\mathcal{V}_j$ if

$$(\ \mathcal{G}(\mathcal{V}_i) \geq \mathcal{G}(\mathcal{V}_j)\ ) \land (\exists k)(\ \mathcal{G}(\mathcal{V}_i)[k] > \mathcal{G}(\mathcal{V}_j)[k]\ )$$

Using this dominance relation, we can eliminate all dominated measures.

Evaluation by an Objective Function

It is unlikely that a single measure dominates all others. To select among those measures, we need to assess the relative importance and trade-off of each scale. Generally speaking, under different environments with different goals, different scaling functions have differing importance. An objective function needs to be derived to capture the relative importance of each scale and to be used to make the final selection.

The result of this assessment is that a weight of importance is assigned to each $\mathcal{G}(\mathcal{V})[i]$, which forms the weight vector $\mathcal{W}$. Notice that unlike $\mathcal{G}$, $\mathcal{W}$ is independent of measures, i.e. $\mathcal{W}$ is the same for any given measure $\mathcal{V}$:

$$(\forall i, j, k)\ \mathcal{W}(\mathcal{V}[i])[k] = \mathcal{W}(\mathcal{V}[j])[k] = \mathcal{W}[k]$$

At the present time, determination of the weights $\mathcal{W}$ is still a subjective activity. However, techniques, such as the equilibrium probability in [6], can be used to help the user determine subjectively which scaling function is most important in that particular application.

The objective function to be used for the final selection is simply the dot product of $\mathcal{G}(\mathcal{V})$ and $\mathcal{W}$. The selection problem reduces to the optimization problem among all $\mathcal{V}[i]$ that satisfy all boundary conditions:

$$\max_i \left(f_i = \sum_j \mathcal{G}(\mathcal{V}_i)[j] * \mathcal{W}[j] \right)$$

Since all measures not in the feasible region were eliminated in the previous stage, the final selection is a simple unconstrained optimization problem. A maximization algorithm can be employed to find the solution (or solutions if multiple measures have the same objective function value).

Measure elimination using dominance relations can be combined (or merged) into this final step of selection. If $\mathcal{V}_i$ dominates $\mathcal{V}_j$ then $f_i > f_j$, and as a result of maximization, $\mathcal{V}_j$ will be eliminated.

[5] Other possibilities can be handled in similar fashion. For example, if the target is T and it is equally undesirable to over-shot as under-shot target, then the symmetric distance of $|T - \mathbf{S}_j(\mathcal{V})[k]|$ can be used in definition for $\mathcal{G}(\mathcal{V})[i]$.

4 Measure Selection for Classification Trees

The selection process uses the following general framework of software measurement including goal and environment analysis, measure selection, refinement and feedback:

1. *Analyze the environment and goal to identify, classify and establish relations of relevant entities, attributes and properties.*

   This analysis focuses on external usage of the measures (the goal) and the external constraints (the environments). They restrict the use of certain measures, and discriminate among others.

2. *Make the selection of candidate measures.*

   This process can be further subdivided into three steps, corresponding to the previous sections in this paper. To summarize, we need to:

   (a) Reject all measures falling outside the feasible region using boundary conditions.

   (b) Evaluate specific aspects of measures using evaluation scales.

   (c) Aggregate individual evaluation scales and make the final selection.

3. *Fine tune the measures in consideration of aspects not considered in step 2.*

   The best available measure selected may not match the goal and environment perfectly. In such cases, fine tuning is performed to derived a better measure based on the one selected.

4. *Use the measures selected and feedback all relevant information.*

   Under use, problems might be found, modifications and adjustments made, and lessons learned. These may reflect weakness of certain axioms, improper classification, inadequate training, or improper use etc. All these provide valuable information, to perfect the axioms and classification, to improve the selection process or to refine the fine tuning.

This framework resembles Basili's GQM paradigm [3]. Step 1 roughly correspond to the Goals and Questions formulation in GQM, while Step 2 corresponds to the selection of Metrics in GQM. Steps 3 and 4 provide the important feedback needed to improve the process.

4.1 Classification trees

The following is an example of measure selection [19] for Classification Tree Analysis (CTA) based upon an earlier study by Selby and Porter [16] where complexity measures are used to construct classification trees of modules to identify critical (in terms of cost and faults) components in a software development environment. Our study consisted of the following:

Environment: Sixteen software systems, ranging from 3000 to 112,000 lines of FORTRAN source code, were selected from NASA ground support software for unmanned spacecraft control [5]. Each required between 5 and 140 person-months to develop over a period of 5-25 months by 4-23 persons. Each project contained from 43 to 531 modules, totalling over 4700 modules. These modules are the objects on which the analysis in the study is based.

Data: There are 74 attributes, each quantified by a specific measure, for each module divided into three broad categories: fault, effort, and style (or complexity), summarized in Tables 1, 2, and 3. In each of the tables, we deliberately separated measures into two groups: 1) derived measures of averages in the form of x *per* y (e.g., faults per 1000 source lines), and 2) other measures.

Goal: The goal is to identify critical components, specifically to identify components with high cost and many faults.

Process: There are 6 parameters to be selected (for more detail, see [16]):

1. *Goal* (the attribute to be estimated): effort or faults;
2. *Availability*: whether an attribute is available early in development cycle;
3. *Evaluation Function Heuristic*: selecting attributes (measures) to generate classification tree;
4. *Tree Termination Criteria*: the least proportion of positive or negative instance to make a prediction (thus to terminate a tree);
5. *Number of Projects in the Training Set*: the data from the most recent n projects ($1 \leq n \leq 15$, the training set) are used to estimate the result on the next project (the singleton testing set);
6. *Ordinal Grouping of Attributes*: either quartile or octile.

The identification of those critical components (parameter 1) are through the classification trees for modules based on some historical data (parameter 5). At each interior node of the tree, a measure (or an attribute) is used to separate modules into various sets according to measure values (parameter 6). A high risk module is one with cost or faults at the upper most quartile or octile. If more than a certain portion (parameter 4) of modules in a specific set are high risk (or low risk), the set is said to be high risk (or low risk). Otherwise another attribute is used to further separate among these modules.

Performance Measures: When using the classification tree on the test data, a number of performance measures can be obtained depending on the match between predicted high/low risk modules and actual high/low risk modules, as follows:

- *Coverage:* Percent of modules where positive or negative predictions are made;
- *Accuracy:* Percent of modules where prediction and actual data agree;
- *Completeness:* Percent of actual high risk modules correctly predicted;
- *Consistency:* Percent of predicted high risk modules who are actually high risk.

4.2 Application of Model to CTA

Suppose we are trying to decide the cost by estimating the total development effort based on 1) the historical data between program complexity and effort, and 2) the new software system, whose complexity can be easily measured. Historical data consist of one project with 43 modules (the

Attributes	Derived attributes
errors	
faults	
fault isolation effort	
fault correction effort	errors per 1000 executable statements
fault implementation effort	errors per 1000 source lines
multiple module faults	fault correction effort per 1000 executable statements
modifications	fault correction effort per 1000 source lines
changes	faults per 1000 executable statements
change isolation effort	faults per 1000 source lines
change correction effort	
change implementation effort	
multiple module changes	

Table 1: Fault and Change Attributes

Attributes	Derived attributes
	total development effort per 1000 executable statements
	total development effort per 1000 source lines
	percent code effort of total development effort
total effort	percent design effort of total development effort
overhead effort	design effort per code effort
design effort	design effort per comment
code effort	design effort per function call
test effort	design effort per module call
	design effort per function plus module call
	design effort per input-output statement
	design effort per input-output parameter

Table 2: Development Effort Attributes

Meets BC_3	Fails BC_3
	assignment statements per 1000 executable statements
	input-output statement per comment
assignment statements	input-output parameters per comment
input-output statements	input-output statements per 1000 executable statements
input-output parameters	input-output statements per input-output parameter
source lines	input-output statements per 1000 source lines
comments	function calls per comment
source lines minus comments	function calls per input-output statement
executable statements	function calls per function plus module call
function calls	function calls per input-output parameter
module calls	function calls per module call
function plus module calls	module calls per comment
cyclomatic complexity	module calls per input-output parameter
operators	module calls per function plus module call
operands	module calls per input-output statement
total operators	function plus module calls per 1000 source lines
total operands	function plus module calls per input-output statement
decisions statements	function plus module calls per input-output parameter
format statements	function plus module calls per 1000 executable statements
origin	function plus module calls per comment
	cyclomatic complexity per 1000 source lines
	cyclomatic complexity per 1000 executable statements

Table 3: Design and Implementation Style Attributes

training set) while the current project consist of 176 modules (the test set). Based on previous experience with CTA, 1) the quartile grouping of attributes is to be used for practicality of tree branching factors; and 2) tolerance level is set to be 25%.

External attributes were not used as we were trying to evaluate internal properties. Based on this analysis, only the complexity (or style, as is called in [16]) measures, rather than various effort or fault measures, were selected to construct the classification tree (Figure 3). This reduced the candidate measures from 74 to 40.

All candidate measures satisfy boundary conditions $\mathbf{BC}_1$ (comparing functionally equivalent programs), $\mathbf{BC}_2$ (comparing component-composite pairs), and $\mathbf{BC}_5$ (no single cluster). However, many of the measures do not satisfy $\mathbf{BC}_3$, the general monotonicity axiom. These measures are measures of averages such as *assignment statements per 1000 executable statements*, which may be correlated with average effort per 1000 lines or so, but not with the total development effort. Therefore these measures were eliminated. This reduced the candidate measures from 40 to 18, with the candidate measure set $\mathbf{S}$ being the left half of Table 3.

To see that $\mathbf{BC}_6$ (i.e., eliminating weaker class measures) is satisfied, we notice that both syntactical structures as well as features such as comments consume programming effort. Both abstract and non-abstract measures provide some information, and correlate to total programming effort. Therefore measures from both these classes are potentially acceptable. On the other hand, interaction between component-composite program pairs are not a major concern, because the programs are studied at the module level only.

Given 18 remaining measures that meet $\mathbf{BC}_1$ through $\mathbf{BC}_6$, we next use our scaling and objective function to determine which of them best predicts total effort. The 18 measures are all strictly monotonic.

To use the discriminating scale DS_1, we have to adjust it according to the environment. The underlying distribution, as we assumed, is a four region distribution (grouped into four quartiles) determined by historical data. We combine this scaling activity into the aggregated evaluation over all four regions. A quartile of modules is positively identified if more than 75% of the modules (tolerance level: 25%) have the upper most quartile of effort. The negative sets can be similarly identified.

Let $m_i(\mathcal{V})$ $(i = 1, 2, 3, 4)$ be the number of modules in quartile i using measure $\mathcal{V}$; $p_i(\mathcal{V})$ be the proportion of modules in $m_i(\mathcal{V})$ belonging or to the upper most quartile of effort; and $n_i(\mathcal{V})$ be the rest in $m_i(\mathcal{V})$. As a result, a quartile is positively identified if $p_i(\mathcal{V}) \geq 0.75$, and negatively identified if $n_i(\mathcal{V}) \geq 0.75$.

To formulate the objective function for the aggregated selection, we need to evaluation the contribution of each quartile. We can weigh them by the number of modules falling into the quartile. Therefore, we formulate our selection criteria as:

$$\max_{\mathcal{V} \in \mathbf{S}} \left\{ \sum_{p_i(\mathcal{V}) \geq 0.75 \vee n_i(\mathcal{V}) \geq 0.75}^{4} \{ m_i(\mathcal{V}) * p_i(\mathcal{V}) + m_i(\mathcal{V}) * n_i(\mathcal{V}) \} \right\}$$

This selection criterion maximizes the number of modules in positively or negatively identified quartiles. For each of the quartile neither positively nor negatively identified, another measure is selected using the same criterion. The process continues until all modules are identified or all measures are exhausted.

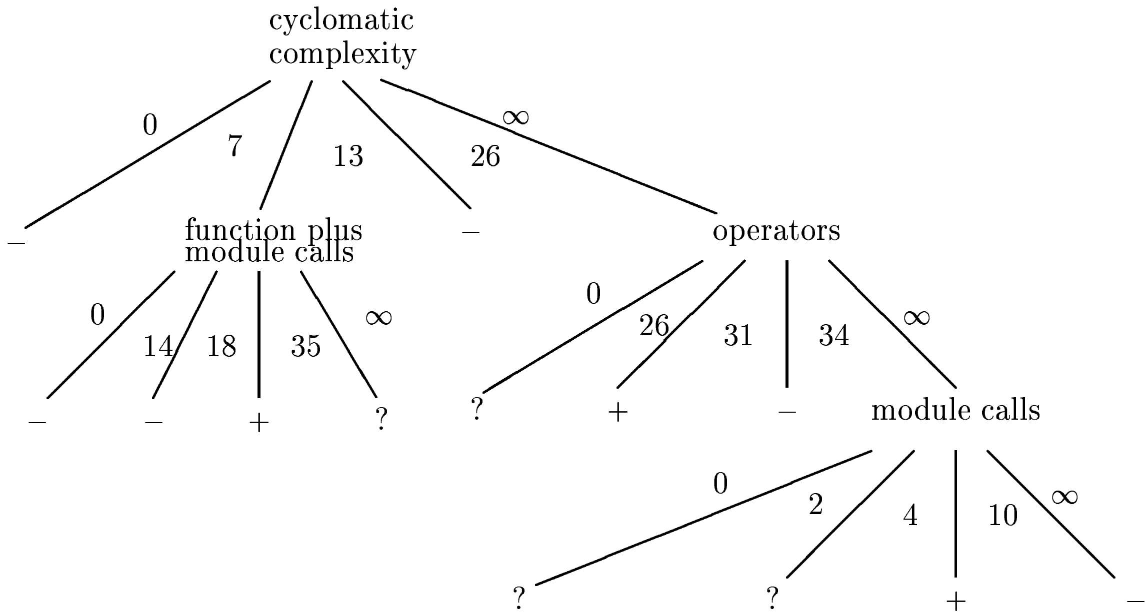

Figure 3: Classification Tree

		Actual		
		+	–	total
Predicted	+	7	17	24
	–	4	143	147
	total	11	160	171

a. modified

		Actual		
		+	–	total
Predicted	+	7	32	39
	–	4	129	133
	total	11	161	172

b. original

Table 4: Prediction Result

4.3 Results of CTA Application

Figure 3 gives the classification tree constructed from training data. Each non-terminal node has an associated measure on four exiting arcs. Each arc has a pair of numbers (lower bound to its left and upper bound to its right), representing the range of measured complexity of modules. A terminal node is marked by either a "+", a "–", or a "?", representing that the modules in the node is positively, negatively, or not identified respectively. This allows each module to be successively classified into finer and finer classes, and ending up in positive, negative or unknown classes at the terminal nodes of the tree.

Using the tree on the test data with 176 modules, various predictions can be made (Table 4.a). To compare the relative effectiveness of the selection process, we also generate the classification tree using the original CTA tools, and summarize the result in Table 4.b. To ensure meaningful comparison, only the 18 complexity measures are used. Assuming our specific goal and environment, the original selection method will select from 40 different complexity measures.

Table 5 gives the comparison of modified and original classification trees according to each performance measure, as well as compares the result with random guessing. Prediction made by both the modified and original CTAs are better than random guessing. All the performance

performance measure	modified	original	random
not identified	5	4	
correctly identified	150	136	
incorrectly identified	21	36	
coverage	97.16%	97.72%	100%
accuracy	87.71%	79.06%	75%
completeness	63.63%	63.63%	25%
consistency	29.16%	17.94%	25%

Table 5: CTA Performance Comparison

measures except coverage using the modified classification tree are an improvement to the original CTA, and coverage is quite comparable. In an earlier paper [19], additional examples are given demonstrating the value of this process.

5 Conclusion and Future Research

In this paper, we developed a technique for general complexity measure selection, based on our earlier formal model of complexity. The general selection problem is formulated as a constrained optimization problem, with clearly defined steps for determining the feasible region, choosing among feasible measures, and aggregating these individual evaluations for the final selection.

While the viability and the effectiveness of the technique was demonstrated by various examples throughout the paper, a more conclusive evaluation needs to be done on a set of real-life software measurement projects. The initial result of this, which is summarized in Section 4.1, looks very promising, and we expect similar results using multiple sets of test data.

By basing measure selection on a formal model of program complexity, we are able to intelligently choose from among numerous candidate measures, as our CTA study demonstrates. Giving this field a scientific basis is an important prerequisite for applying these techniques in industrial applications.

References

[1] W. G. Bail and M. V. Zelkowitz, "Program complexity using hierarchical abstract computers," *Journal of Computer Languages*, Vol. 13, No. 3/4, pp 109–123, 1988.

[2] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Comm. of the ACM*, 27(1):42–52, Jan. 1984.

[3] V. R. Basili and H. D. Rombach, "The TAME project: Towards improvement-oriented software environments," *IEEE Trans. Soft. Eng.*, Vol. 14, No. 6, pp 758–773, June 1988.

[4] B. W. Boehm, *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[5] D. N. Card, F. E. McGarry, J. Page, , S. Eslinger, and V. R. Basili, "The software engineering laboratory," Technical Report SEL-81-104, Software Eng. Lab., NASA/GSFC, Greeenbelt, MD, 1981.

[6] S. Cárdenas-García and M. V. Zelkowitz, "A management tool for evaluation of software designs," *IEEE Trans. Soft. Eng.*, Vol. 17, No 9, pp 961–971, Sept. 1991.

[7] C. F. Dietrich, *Uncertainty, Calibration and Probability: The Statistics of Scientific and Industrial Measurement.* John Wiley, New York, NY, 1973.

[8] N. E. Fenton, *Software Netrics: A Rigorous Approach,* Chapman and Hall, 1991.

[9] N. Fenton and A. Melton, "Deriving structurally based software measures," *J. of Systems and Software*, Vol 12, No. 3, pp 177–187, 1990.

[10] N. E. Fenton and R. W. Whitty, "Axiomatic approach to software metrication through program decomposition," *The Computer J.*, Vol 29, No. 4, pp 330–339, 1986.

[11] M. H. Halstead, *Elements of Software Science.* Elsevier, new York, 1977.

[12] T. J. McCabe, "A complexity measure," *IEEE Trans. Soft. Eng.*, Vol. 2, No. 6, pp 308–320, 1976.

[13] A. Melton, D. A. Gustafson, J. M. Bieman, and A. L. Baker, "A mathematical perspective for software measure research," *Software Engineering J.* Vol 5. No. 3, pp 246–254, Sept. 1990.

[14] R. E. Prather, "An axiomatic theory of software complexity measure," *The Computer J.* Vol 27, No. 4, pp 340–346, 1984.

[15] H. D. Rombach, V. R. Basili and R. W. Selby (Ed.), *Experimental Software Engineering Issues: Critical Assessment and Future Directions.* Lecture Notes in Computer Science 706, Springer-Verlag, Berlin, 1993.

[16] R. W. Selby and A. A. Porter, "Learning from examples: Generation and evaluation of decision trees for software resource analysis," *IEEE Trans. Soft. Eng.*, Vol. 14, No. 12, pp 1743–1757, Dec. 1988.

[17] B. Shneiderman, "Exploratory experiments in programmer behavior," *Int'l J. of Computer and Information Science*, Vol. 5, No. 2, 1976.

[18] J. Tian and M. V. Zelkowitz, "A formal model of program complexity and its application," *J. of Systems and Software* Vol. 17, No. 3, pp 253-266, 1992.

[19] J. Tian, A. Porter and M. V. Zelkowitz, "An improved classification tree analysis of high cost modules based upon an axiomatic definition of complexity," IEEE 3rd International Symp. on Software Reliability Engineering, Research Triangle Park, NC, pp 164-172, October, 1992.

[20] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Soft. Eng.*, Vol. 14, No. 9, pp 1357–1365, September, 1988.

[21] M. R. Woodwards, M. A. Hennell, and D. Hedley, "A measure of control flow complexity in program text," *IEEE Trans. Soft. Eng.*, Vol. 5, No. 1, pp 45–50, January, 1979.

[22] H. Zuse, *Software complexity*, Walter de Gruyter, Berlin, 1991.