A Reliability Model Incorporating Software Quality Factors

S. Henry, D. Kafura, K. Mayo,
A. Yerneni and S. Wake

TR 88-45

A Reliability Model Incorporating Software Quality Factors

S. Henry, D. Kafura, K. Mayo, A. Yerneni, S. Wake

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Abstract

In this paper we describe our initial work on a long-term project to develop and validate a reliability model and a new class of software complexity metrics which are related to this model. In contrast to previous "black box" approaches, the reliability model is novel because it incorporates knowledge about the system in the form of quantitative software complexity metrics. While the initial model uses existing software metrics a parallel effort in this project is investigating new classes of metrics, **interface** and **dynamic** metrics, which are useful in their own right but are also of particular relevance to the reliability model. The initial definitions of both the model and the metrics are given along with a description of the next research milestones.

This research is supported by a grant from the Software Productivity Consortium and the Virginia Center for Innovative Technology. The authors gratefully acknowledge the support of these organizations.

I. Introduction

Models for estimating software system reliability are commonly based only on the execution behavior observed during system testing. This common "black box" approach suffers from two principle limitations. First, this approach ignores such system "quality" factors as structure and complexity even though the basic tenets of software engineering hold that these factors are prime determinants of software system reliability. Second, the execution behavior of a system is only known when the system is (at least very nearly) complete. Thus, "black box" reliability models can only deliver their assessments very late in the software life cycle - too late for major reliability problems to be corrected in a cost-effective manner.

In contrast to the "black box" approach Shooman developed a model which included structural information (execution paths)[SHOM83]. Similar to this idea we are investigating a reliability model which incorporates information about the quality factors of the system. To keep the model both objective and quantitative, and also because of our previous research work, the quality factors will be those aspects of the systems which can be measured by software metrics. Such a model might improve on current "black box" models in one or more of three ways: the reliability assessment might be more accurate, the reliability assessment might be more confident, or a partial reliability assessment might be arrived at earlier. This last possible benefit arises because, unlike the execution behavior, the quality factors begin to emerge early in the software life-cycle. Measurement of the evolving quality factors might then provide the basis for early, and increasingly more accurate, reliability estimates arrived at well in advance of the testing phase of the software life-cycle. Estimating system reliability then becomes a process which begins early, rather than late, in the life cycle and has a natural progression from the design and implementation phases to the testing phase.

In this paper we will describe our initial work on the development and validation of the reliability model which incorporates system quality factors. The organization of this paper mirrors the fact that the work itself is proceeding towards two distinct but related objectives. The first objective is the development of the new model. The preliminary model definition is presented in Section II. Section II also contains a description of a three stage validation process which we will use to explore the model's characteristics. The second objective is the development of new software quality metrics which could be used to provide additional model parameters. Beyond their use in the model, these additional software metrics are interesting and useful in their own right. The preliminary definition of the new software metrics is presented in Section III. Finally, in

Section IV, we briefly reflect on our early work and identify the next steps in our investigation of the model and the metrics.

II. Model Definition

One view of the model being developed is shown in Figure 1 below. The notation of this figure will be explain shortly. Notice that the model derives information from two sources -- the system being tested and the testing process. The information which the model incorporates from the system being tested is the complexity of the jth component , C_j , and the total number of components in the system, T. The testing process is described by the total number of test cases performed so far, N, the total number of failures (or errors) observed in component j during the N tests cases, e_j, and the total number of times that component j was executed in test case i, f_{ji}. Thus, the testing process is described by the volume of testing (N), the distribution of this testing volume over the components of the system (f_{ji}), and the distribution of component failures (e_j).The model produces two basic reliability indicators: $P_j(N,C_j)$ - the probability of failure of the jth component in the system after N test cases have been performed; and $P_S(N)$ - the probability of system failure after N test cases have been performed. By "failure" we explicitly mean the observed occurrence of an error in either the component or the system regardless of whether the component or the system ceased to operate as a result of the error's occurrence. In this sense, failure is simply the deviation from desired behavior rather than from a catastrophic event. Hereafter, the terms failure and error will be used interchangeably.

The development of the model is based on a number of assumptions, the first of which we introduce here and the remainder of which will be stated as needed. The first assumption is:

> **Assumption 1:** the probability of discovering an error in a component is inversely proportional to the amount of testing to which the component has been subjected without failure.

This assumption expresses the belief that the reliability of a component increases as the component executes without failure on an increasing number of test cases.

Before trying to cast this assumption into a mathematical framework we must decide exactly how to measure the "amount of testing" which a component has undergone. Recall that one of the measured quantities from the testing process is the number of times component j was executed in test case i (f_{ji}). One measure of the amount of testing is the total number of times that

the component was executed over all test cases. This measure is denoted F_j. Another measure, denoted n_j, is the number of test cases in which the component was executed at least once.

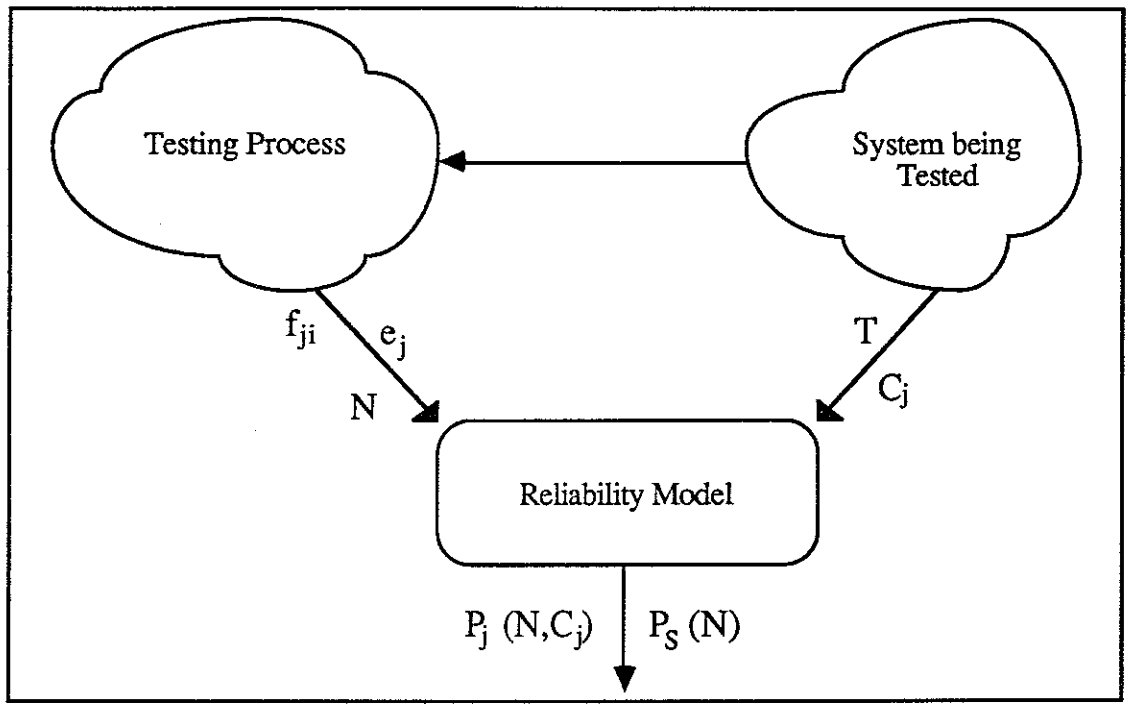

Figure 1: Major Elements of the Model

Figure 2 shows three different testing scenarios for component j. In Figure 2(a) the component is executed in only three different test cases while in 2(b) it is executed in 5 different test cases. In both 2(a) and 2(b), $F_j=\Sigma f_{ji}=10$. In choosing between 2(a) and 2(b), and all other things being equal, we would give preference to the testing scenario in 2(b) under the assumption that the environment in which the component operates is more varied between different test cases than it is within the same test case. This assumption will be explicitly stated later along with other assumptions about the testing process. In the extreme, a component is more thoroughly tested if it is executed once in each of ten 10 different test cases than if it is executed 10 times in the same test case. In 2(b) and 2(c), $n_j=5$. However, we would again prefer the scenario in 2(b) because the total number of times the component is executed in 2(b) is twice ($F_j=10$) that of the scenario in 2(c) ($F_j=5$).

Based on the example in Figure 2 we have defined a measure of testing thoroughness which balances between F_j and n_j. This measure is defined as:

$$F'_j = \frac{n_j \times F_j}{N+1} \tag{1}$$

Figure 2: Examples of Test Case Distribution

Applied to the testing situations in Figure 2, this measure yields values of 3.75, 7.14, and 3.12 for 2(a),(b) and (c), respectively. Using this measure, and temporarily ignoring component failures, we can now cast assumption 1 into a mathematical formula as:

$$\frac{\partial P_j(N,C_j)}{\partial N} \alpha - F'_j \tag{2}$$

This formula states that the rate of change of the component's probability of failure is inversely proportional to the current of testing as measured by F'_j. Substituting for the definition of the F'_j and introducing a constant of proportionality (K_1) we obtain:

$$\frac{\partial P_j(N,C_j)}{\partial N} = -K_1 \left(\frac{n_j \times F_j}{N+1} \right) \tag{3}$$

Integrating (3) with respect to N yields:

$$P_j(N,C_j) = -K_1 \times n_j \times F_j \times \log(N+1) + Y(C_j) \tag{4}$$

The $Y(C_j)$ term in equation (4) is a function of the complexity of the component. To proceed further, we make the next modeling assumption:

Assumption 2: the probability of discovering an error in a component is directly proportional to the complexity of the component.

Applying this assumption to equation (4) and introducing another constant of proportionality, K_2, we obtain:

$$P_j(N,C_j) = -K_1 \times n_j \times F_j \times \log(N+1) + K_2 \times C_j \tag{5}$$

This equation expresses in mathematical terms that a component's probability of failure decreases with increasing testing. It also expresses that more complex components require more testing than less complex ones in order to achieve the same reliability.

Equation (5), however, did not model how the probability of failure changes when an error occurs in a component. This is added to the model by introducing the change in the probability of failure, ΔP_j, on encountering an error. Assuming an error is detected on the N+1 case the equation modification is:

$$P_j(N+1,C_j) = P_j(N,C_j) + \Delta P_j . \tag{6}$$

There are a number of assumptions which underly the formalization of ΔP_j. These assumptions define the characteristics of the testing process and the error correcting process. The assumptions are:

Assumption 3: All errors which occur are observed.

Assumption 4: Each test case results in at most one error.

Assumption 5: Test cases are independent of each other.

Assumption 6: Each test case is directed at testing a different aspect of the system.

Assumption 7: Every error is corrected before the next test case is executed.

Assumption 8: Correcting an error does not introduce new components and does not change the complexities of existing components.

The first four of these assumption describe the testing process. Assumption 3 states that the testing process is thorough in the sense that errors cannot occur without being discovered. According to Assumption 4 a test case is considered to end when an error is detected. By Assumption 5, the ordering of the test cases is immaterial. Finally, Assumption 6 requires that test cases do not repeatedly test only a restricted part of the system. Together Assumptions 5 and 6 imply that reliability can only be properly assessed by a test suite which is sufficiently large and sufficiently varied in composition.

Assumptions 7 and 8 describe the error correcting process. Assumption 7 dismisses the case when the same error is repeatedly discovered. Assumption 8 may or may not be true in practice. One can certainly envision a situation where an error correction substantially changes the complexity of one or more components. However, we are willing to tolerate this assumption, at least for the moment, in order to simplify the model definition and validation. This assumption can be removed at a later time.

Returning now to equation (6), there are three possible ways to model the change in probability. These three ways are:

- modify the *probability* of failure
- modify the *rate of change* of the probability of failure
- modify *both* the probability of failure and the rate of change

Each of these three methods is based on a different intuitive notion of how the error discovery process behaves. The first alternative is consistent with the observation that more errors will be discovered in those components which have already been found to contain errors. Thus, when an error is discovered in a component its probability of failure should be increased because more errors are likely to be found in this component by additional testing. The absence of subsequent errors allows the component to quickly recover its previous probability because the rate of change of the probability is not changed. The second alternative is consistent with the philosophy that the testing and error correcting activities acting together continue to improve the reliability of components. Thus, the discovery, and subsequent correction, of an error only slows the rate at which the component's reliability is increasing. The third alternative is the most pessimistic. In this case the occurrence of an error is taken as an indication of the likely discovery of additional errors and, even in the absence of such errors, the component can recover its previous reliability status

only by successfully executing a larger number of test cases. Each of these three methods will be formalized and explored.

Method 1: Modify the probability of failure

Three factors are considered in adjusting the current probability value in the case of an error occurrence: the complexity of the component which has failed (C_j), the total number of errors (including the current error) detected in this component (e_j), and the number of test cases in which this component has been executed at least once (n_j). The complexity is included because of Assumption 2, and the number of test cases is included because of Assumption 1. The number of errors is included to penalize those components in which a large number of errors have already been detected. The change in probability is defined as:

$$\Delta P_j = K_3 \frac{C_j \times e_j}{n_j} \tag{7}$$

where K_3 is a constant of proportionality. On one extreme, this equation leads to a large increase in probability for a high complexity component in which a large fraction of the test cases have revealed errors in the component. On the other extreme, a small increase in complexity will be produced for a low complexity component which has successfully executed all but a small number of the test cases. This method is illustrated graphically in Figure 3. Figure 3 shoes the failure probability curve for componenet j when errors are detected in test cases n_1 and n_2. Accordingly, the probability curve is adjusted upward at these points. Using method 1, the complete model is then:

$$P_j(N,C_j) = -K_1 \times n_j \times F_j \times \log(N+1) + K_2 \times C_j + K_3 \frac{C_j \times e_j}{n_j} \tag{8}$$

which results from combining equations (5), (6) and (7).

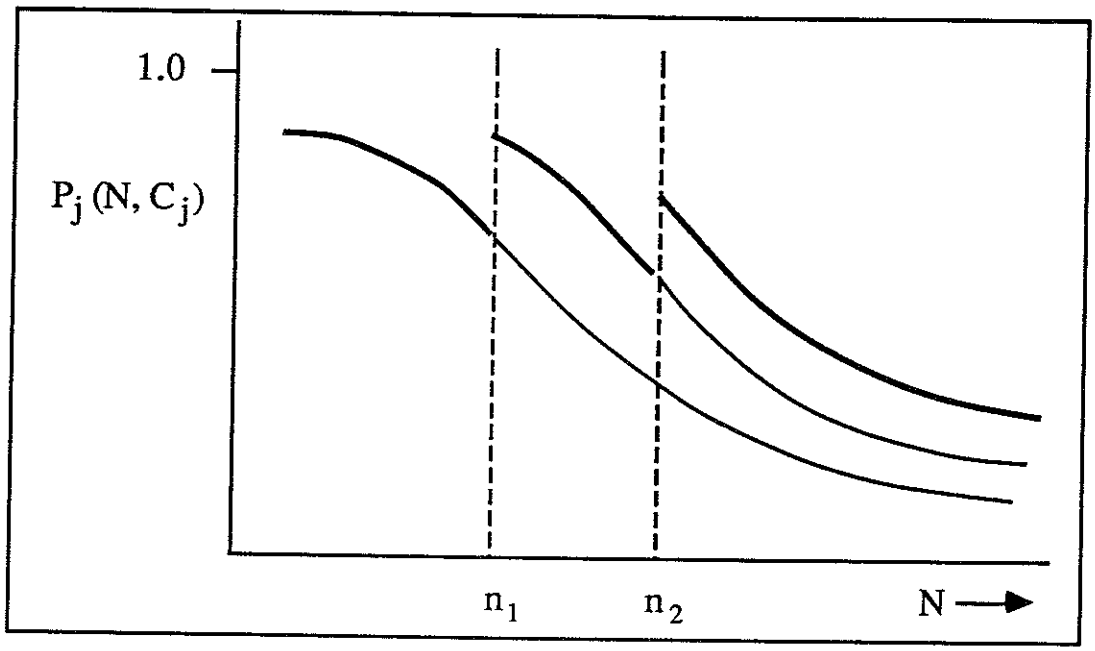

Figure 3: Change in Probability for Method 1

Method 2: Modify the rate of change of the probability

In this method the rate of change of the probability is set to zero as shown in the following equation:

$$\frac{\partial P_j (N,C_j)}{\partial N} = -K_1 \left(\frac{n_j \times F_j}{N+1} \right) + K_4 = 0 \qquad (9)$$

where K_4 is a constant. The effect of this method is shown graphically in Figure 4. The following model results:

$$P_j (N,C_j) = -K_1 \times n_j \times F_j \times \log(N+1) + K_2 \times C_j + K_4 \times N \qquad (10)$$

by integrating with respect to N, as was done earlier.

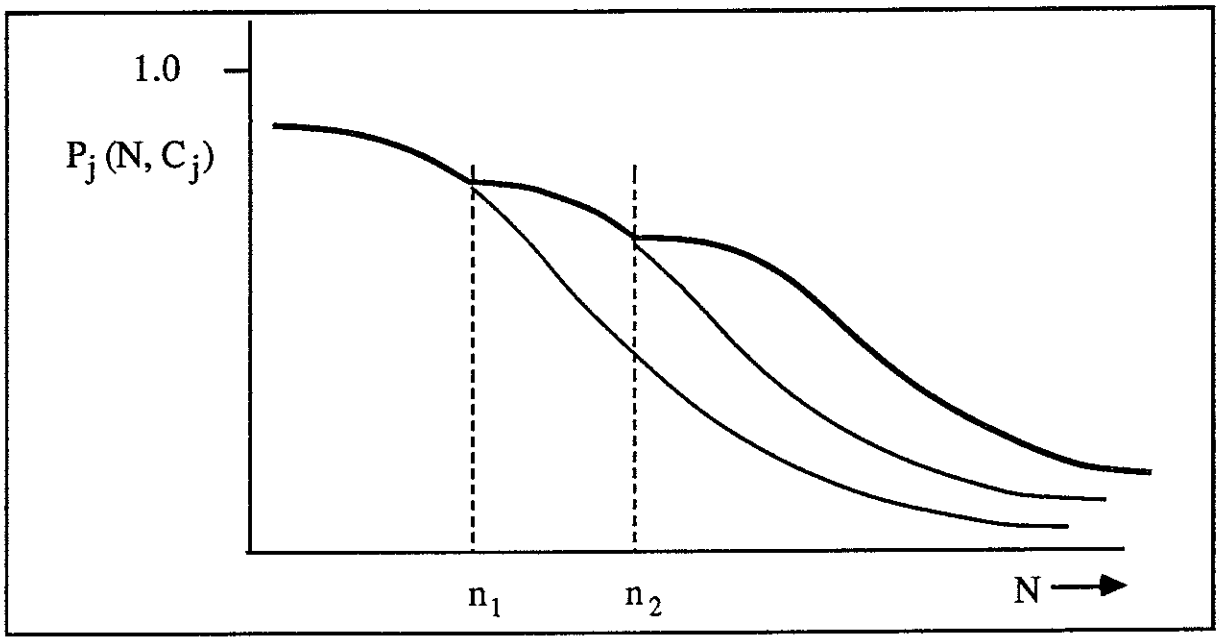

Figure 4: Change in Probability for Method 2

Method 3: Modify both the probability of failure and the rate of change

In this method both of the adjustments of the other two methods are applied simultaneously. This approach is shown graphically in Figure 5. It should be relatively obvious that this method is described by:

$$P_j(N,C_j) = -K_1 \times n_j \times F_j \times \log(N+1) + K_2 \times C_j + K_3 \frac{C_j \times e_j}{n_j} + K_4 \times N \qquad (11)$$

which is simply a combination of equations (8) and (10).

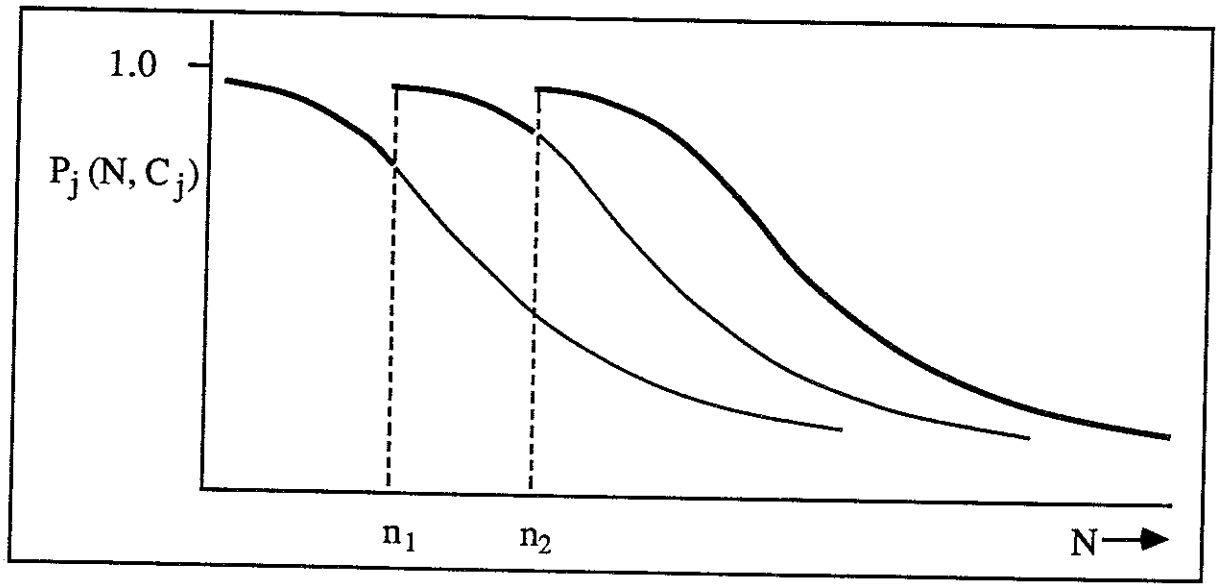

Figure 5: Change in Probability for Method 3

Equations (8), (10) and (11) are three alternative models for determining the probability of failure of each component based on that component's complexity and testing history. Another reliability measure, the probability of system, failure will also be developed. System failure is determined by the composition of the component failures. This composition is based on the following assumption of component independence:

Assumption 9: the failures in each component are independent of the failures in all other components.

One may imagine situations in which this assumption would not hold between particular components. For example, let us assume that two components shared the same false assumption. Then an error, related to this incorrect assumption, that is discovered in one component would imply a higher likelihood of discovering errors in the other component as well. However, we adopt this assumption for three reasons. First, it is not clear that the dependencies between components need to be modeled in order to obtain reasonable reliability estimates. Second, it is not clear how such dependencies can be modeled or measured. Third, as shown below, adopting this assumption leads to a simple model of system failures.

The model of system failure adopted from Shooman [SHOM83] is shown in equation (12) and is derived as follows. The term $(1\text{-}P_j(N,C_j))$ is the probability of *success* for component j (i.e., the probability that component j will *not* fail on the next test case). Given assumption 9, the probability of *system success* is simply the product of the component probabilities. The probability of *system failure* is then obtained by subtraction.

$$1 - \prod_{j=1}^{T} (1 - P_j(N, C_j)) \qquad (12)$$

This completes the development and presentation of the model.

Validation Process

The utility of the model presented above will be determined by the accuracy with which the reliability of actual production systems is estimated by the model. The validation cannot usefully be accomplished in one encompassing step. Therefore, the following three step process has been outlined. Each step is more realistic than the previous step. The three steps are:

- the use of simulation techniques to generate a hypothetical system and test scenario
- the use of a small-scale system to which error seeding and test case generation strategies will be applied
- the use of a production scale system using actual error histories and test cases developed during the testing process.

The first step allows highly controlled experiments with the model. These experiments will lead to a better understanding of the basic characteristics and sensitivities of the model. For example, it can determine how sensitive the reliability measures are to the variation in component complexities. At the heart of this first step is a bivariate normal model which will be used to generate the characteristics of the hypothesized system. This statistical model will be calibrated from actual project data.

The second step, more realistic than the first, will employ a small-scale system. In this step actual complexities will be used. The intent is to maintain some degree of control over the validation by using a small-scale system into which different error patterns can be injected and for which a variety of realistic test scenarios can be generated.

The final, and most realistic, step will be based on a production scale software system. Historical data collected during the system's development and testing will be used. Thus, all of the complexities, errors and test cases will be realistic. While this is the most realistic step in the validation it is also the least controllable.

III. Definition of New Software Metrics

This section concentrates on the new and existing software quality metrics which will be used as parameters to the reliability model previously discussed. Recent studies have shown that existing software metrics can predict the maintainability of software systems [LEWJ88,WAKS88], but more importantly, that software metrics can predict the software system quality at design time [SELC87,GOFR87].

Henry and Selig demonstrated that using software quality metrics with software designs the quality of a software system at design time. The system designs were produced using a program design language (PDL). This study lend credence to the fact that software quality metrics can and should be used early in the software life cycle [HENS88a].

Another study by Henry and Goff showed similar results using a graphical design language, GPL [HARH85]. Additionally, this work compared the quality of designs written in a textual design language with those written in a graphical design language. The GPL produced designs and resultant source code with much better quality than a textual design language [HENS88b].

These studies support our belief that using software quality factors collected early in the software life cycle will improve not only the reliability model but will allow the reliability model to be used early in the software development life cycle.

The remainder of this section briefly discusses and references the existing metrics used in previous studies and presents preliminary definitions of new metrics which will be incorporated into the reliability model. The first of the new metrics focuses on the interfaces among system components. A second new type of metric is based on the dynamic aspects of system execution. Previously, all software quality metrics have measured static attributes of the system design or code, this new collection of metrics will attempt to measure more dynamic aspects of the system.

Existing Metrics

Code Metrics

Metrics that are directly "countable" from the source code are refered to as code metrics. Many code metrics have been proposed in the recent past. An effort has been made to limit this discussion to a few of the more popular ones that are typical of this type of measure. They include lines of code, selections from Halstead's Software Science, and McCabe's Cyclomatic Complexity. Each of these metrics are widely used and have been extensively validated.

Lines of Code

The most familiar software measure is the count of the lines of code with a unit of LOC or for large programs KLOC (thousands of lines of code). Unfortunately, there is no consensus on exactly what constitutes a line of code. Most researchers agree that a blank line should not be counted, but cannot agree on comments, declarations, null statements such as the Pascal BEGIN, etc. Another problem arises in free format languages which allow multiple statements on one textual line or one executable statement spread over more that one line of text.

For this study, the definition used is: A line of code is counted as the line or lines between semicolons, where intrinsic semicolons are assumed at both the beginning and the end of the source file. This specifically includes all lines containing program headers, declarations, executable and non-executable statements.

Halstead's Software Science

A natural weighting scheme used by Halstead in his family of metrics (commonly called Software Science [HALM77]) is a count of the number of "tokens," which are units distinguishable by a compiler. All of Halstead's metrics are based on the following definitions:

n_1 = the number of unique operands.

n_2 = the number of unique operators.

N_1 = the total number of operands.

N_2 = the total number of operators.

Three software science metrics will be discussed, specifically N, V, and E.

The metric N is simply a count of the total number of tokens expressed as the total number of operands plus the total number of operators, i.e., $N = N_1 + N_2$. An operand is defined as a symbol used to represent data and an operator is any keyword or symbol used to express an action [CONS86].

The second Halstead metric considered is volume. The volume, V, represents the number of bits required to store the program in memory. Given n as the number of unique operators plus the number of unique operands, i.e., $n = n_1 + n_2$, then $\log_2 (n)$ is the number of bits needed to encode every token in the program. Therefore, the number of bits necessary to store the entire program is:

$$V = N \times \log_2 (n)$$

The final Halstead metric examined is effort (E). The effort metric, which is used to indicate the effort of understanding, is a function of the volume (V) and the difficulty (D). The difficulty is estimated as:

$$D = \frac{n_1 \times N_1}{2 \times n_2}$$

Given V and D, the effort is calculated as:

$$E = V \times D$$

The unit of measurement of E is the elementary mental discriminations which represent the difficulty of making the mental comparisons required to implement the algorithm.

McCabe's Cyclomatic Complexity

McCabe's metric [MCCT76] is designed to indicate the testability and maintainability of a procedure by measuring the number of "linearly independent" paths through the code. To determine the paths, the code is represented as a strongly connected graph with one unique entry and exit point. The nodes are sequential blocks of code, and the edges are decisions causing a branch. The complexity is given by:

$$V(G) = E - N + 2$$

where

E = the number of edges in the graph

N = the number of nodes in the graph.

According to McCabe, V(G) = 10 is a reasonable upper limit for the complexity of a single component of a program.

Structure Metrics

It seems reasonable that a more complete measure needs to do more than simple counts of lines or tokens in order to fully capture the complexity of a module. This is due to the fact that within a program there is a great deal of interaction between modules. Code metrics ignore these dependencies implicitly assuming that each individual component of a program is a separate entity. Conversely, structure metrics attempt to quantify the module interactions using the assumption that the inter-dependencies involved contribute to the overall complexity of the program's units, and ultimately that of the entire program. In this study, the structure metric examined is Henry and Kafura's Information Flow Metric.

Henry and Kafura's Information Flow Metric

Henry and Kafura [HENS79] [HENS81a] developed a metric based on the information flow connections between a procedure and its environment called fan-in and fan-out which are defined as:

fan-in - the number of local flows into a procedure plus the number of global data structures from which a procedure retrieves information.

fan-out - the number of local flows from a procedure plus the number of global data structures which the procedure updates.

To calculate the fan-in and fan-out for a procedure, a set of relations is generated that reflects the flow of information through input parameters, global data structures and output parameters. From these relations, a flow structure is built that shows all possible program paths through which updates to each global data structure may propagate. The complexity for a procedure is defined as:

$$C_p = (\text{fan-in}_p \times \text{fan-out}_p)^2$$

where

C_p = the complexity of procedure p

fan-in_p = the number of fan-ins to procedure p

fan-out_p = the number of fan-outs from procedure p.

The term (fan-in x fan-out) is squared to represent the idea that the complexity due to the inter-relationship between components is non-linear.

Procedural complexity is used to find those procedures with heavy data traffic and those that are not adequately refined. Module complexity reveals overloaded data structures as well as improper modularization. The level complexity may be used to detect missing levels of abstraction or to compare alternate system designs. Module and level complexities are fully defined in Henry and Kafura's work [HENS79].

Hybrid Metrics

Many researchers feel that in order to fully measure the complexity of a system both code and structure metrics must be included. This combination yields hybrid metrics.

Interface Metrics

The interface metric is a new set of metrics being developed through this research to be incorporated into the reliability model. This new measure is a hybrid metric concerned with the complexity of communicating modules/procedures. This metric finds its basis in the information flow concept as put forth by Henry and Kafura [HENS81a]. It also takes into account the structure of the code similar to that of McClure's complexity [MCCC76].

The motivation behind the interface metric is that previous metrics fell short of maintaining all the information needed to model a non-trivial indicator of the complexity of code and interfaces. While Henry and Kafura [HENS81a] modeled metrics along data flow lines, they ignored the inherent complexities that different data structures enter into the picture. McClure weights the different structures by either selection or repetition; however, this approach neglects the complexity associated with each selection/repetition construct control clause. Also, various styles of repetition constructs were put into a single class; there could possibly be a difference among various structures inherent complexity.

Within the calculation of the Interface Complexity Metric there are several underlying code metrics that must be generated. Instead of using the predefined 'classical' metrics, several metrics have been defined with properties similar to Halstead's Software Science measurements [HALM77]. These new measurements define expression complexities of all mathematical and relational formulas. With these measurements, the structure complexity measures associated with selection and repetition clauses can be better addressed.

To facilitate the needs of the interface complexity measure, several different complexities must be defined. These complexities try to capture the 'weight' of data being passed through the 'complexity' of the given code at the communication invocation. This implies that the complexity must take into account the types, expression, and flow.

These metrics consider the differences inherent in types, operators, and flow structures. The complexities to be defined are: Type, Operator, Expression, Flow-Control, Environment, Variable and Interface Complexities. These complexities are broken into two different classes:

(1) Type, Operator, Flow Control, and

(2) Expression, Environment, Variable (Ref_Read and Ref_Write), and Interface Complexity.

The first class is calculated directly from the source code. The second class is calculated from combinations of the first class.

Type Complexity

The type complexity attempts to capture the inherent differences among the variable types. This allows the metric to differentiate between the inherently easier INTEGER type and the more complex FLOATING-POINT type variables.

Most procedural programming languages contain two classes of types: Basic Types and Group Types. The basic types define single value variables such as: BOOLEAN, INTEGER, REAL (FLOATING-POINT, FIXED-POINT), CHARACTER, etc., while the group types define collections of data, i.e., RECORDS, ARRAYS, FILES, etc. These collections, as good

programming practice dictates, are specified with regards to a logical unit. The TYPE complexity is therefore a weight associated with each variable with regards to its type.

Operator Complexity

It should be obvious that there are complexity differences between operators. This is most easily seen when '+' (plus) and '**' (exponentiation) are considered. While both are binary operators, their associated complexity is far different, with exponentiation being the more complex. Therefore, a weighting scheme, the operator complexity, is needed to characterize the relationship among the operators and their complexities.

In considering these operators, the ordering must by its very nature relate the relational and numerical operators. Therefore, considerations must be taken to retain the relationship between the complexities.

Expression Complexity

Given that there are type and operator complexities, then the next logical step is to define a complexity for a given expression. This complexity is defined as a function of the operator complexities and the operand type complexities. With this backdrop, it is possible to rank expressions in a way to define highly complex expressions. The motivation is that highly complex expressions yield more probable locations of errors.

Flow-Control Complexity

The possible complexity associated with different paths through the source code must also be considered. There is a complexity difference among the flow control statements. In general there are two classes: selection and repetition. Selection statements, in most languages, include IF-THEN-ELSE (and all the derivatives) and CASE. Repetition statements include the two looping constructs: Test-Before and Test-After cases.

There is certainly a difference between the selection and repetition statements, and each adds a different complexity to the communication (interface complexity). However, is there a difference among the the selection statements? The question must also be answered for the repetition statements. The weighting scheme derived here will rank flow-control statements, and allow the metric to differentiate the complexities associated with each and how it affects the communication.

Environment Complexity

The environment of the communication invocation is very important to the interface complexity measure. It is at the point of the invocation that all aspects of the source code come

together to produce correct communication, and thus an error at invocation results in the propagation of the error.

The environment complexity is a function of the expression complexities and the flow-control complexities.

Variable Complexities

Each variable that affects the communication between two modules (used loosely) must be analyzed as to the extent that its complexity will add to the interface. These variables become more complex as they are modified, and as they modify others their complexity must be represented in the modified variable. There are two complexities associated with each variable, Ref_Read and Ref_Write, these complexities try to capture the variable's usage.

Ref_Read

Each time a variable is referenced, or read, the Ref_Read complexity will contain the environment information. This is useful in finding orders of most used variables and the complexity in which they are used.

Ref_Write

Each modification to a variable causes the variable to become more complex. Therefore, at each modification the Ref_Write complexity retains information about the expression complexity of the right hand side of the assignment. In this way a variable may be identified as to its modifications and their complexities.

Interface Complexity

The interface complexity is now defined to be a 2-tuple containing information about the environment of the call and the nature of the parameter complexity. The environment element is a function of the environment complexity at each invocation of the communication or interface; the parameter complexity is a function of the expression complexities and the Ref_Write complexities of the parameters at each invocation.

In summary, to generate the interface complexity there are several underlying complexities that are defined. Type complexities measure the differences between variable types, in that an ordering is preserved such that more difficult types have a higher order. Operator complexities, much like type complexities, assign a weight to the various operators. Flow Control complexities assign weights to the various selection/repetition statements found in the source language due to the difference of their semantics. Expression complexities are a function of the underlying type and operator complexities, while Environment complexities are a function of expression and flow control complexities. Finally, Interface complexities are a function of the environment and

parameters at each invocation to the calls. The following figure should help to clear up any ambiguities.

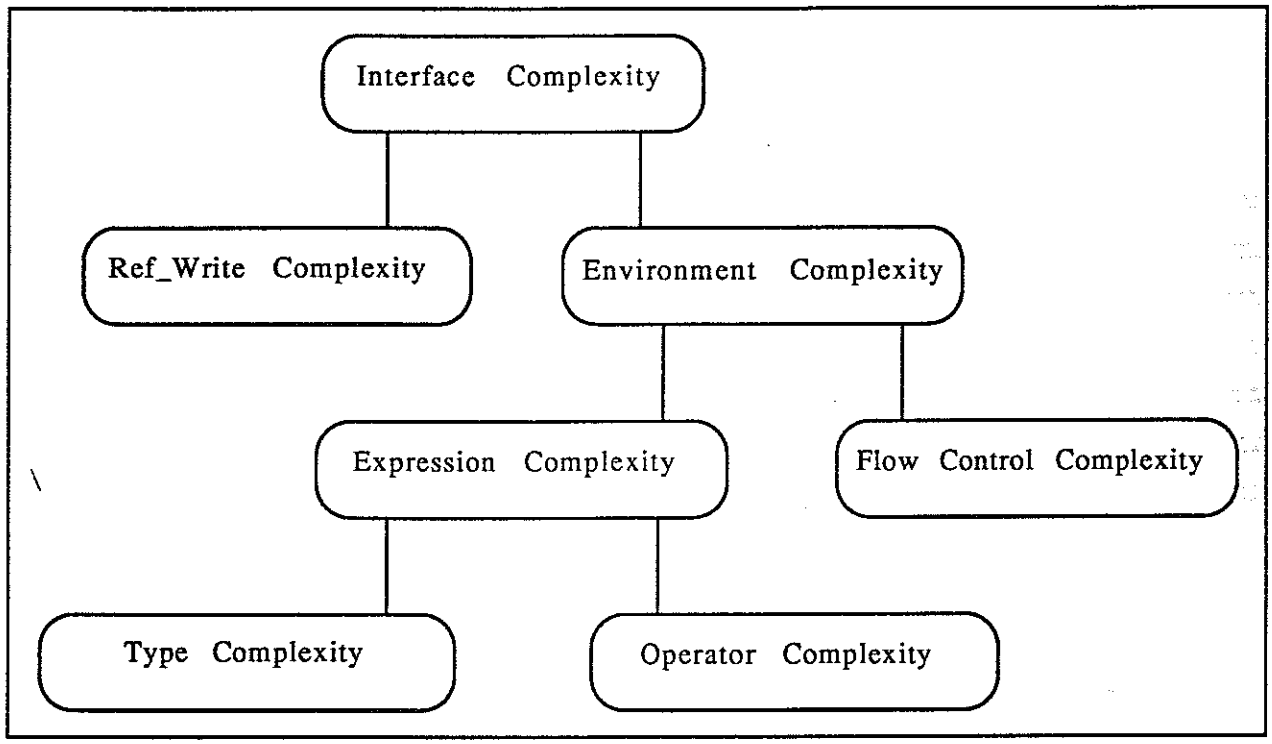

Figure 6. Chart of Complexity Build Up

Dynamic Metrics

Software metrics research has focused exclusively on measures which are determined by a static analysis of the system under study. Such metrics hypothesize that the static, pre-execution relationships among statements or the relationships among components are important influences on the accuracy or economy with which basic software engineering tasks can be performed. A large and growing body of empirical evidence supports, at least to some degree, this hypothesis.

Without diminishing the importance of the static system structure, it should also be realized that a software program or system is not a static entity. Software systems have "moving parts": loops which iterate, data which flows from one component to another, tasks which execute in nondeterministic sequences that may not even be repeatable. Thus, we hypothesize that the dynamic (execution time) behavior of the system must be considered in assessing the system's quality factors. The dynamic behavior is particularly important if, as is the case in this study, reliability is a quality factor of prime concern.

A dynamic metric is a measure of complexity computed during execution of the software system. Because dynamic metrics are computed at execution time they are more accurate at establishing actual relationships between system components. However, dynamic metrics only measure what occurred during execution, not what might occur during a future execution. Static metrics can be improved by information from dynamic analysis in some cases. Used other ways, static measures may be more descriptive because they measure the potential complexity of the components.

There are two primary advantages of dynamic over static metrics. First, dynamic metrics are more accurate because static program analysis may not be able in some cases to establish a precise relationship between system components. The greater accuracy of dynamic metrics is particularly true of languages like ADA which contain constructs whose exact meaning may not be determined until run-time. Consider, for example, an ADA procedure which raises an exception. It is not possible for a static analysis to determine which system component will handle this exception because the handler is determined at run-time. The best that a static analysis could hope to accomplish would be to identify all of the possible handlers. Thus, only a dynamic metric – one which is based on run-time information – would be able to correctly identify the true relationship between the component raising the exception and the component handling the exception. Similar examples can be given for other ADA constructs such as generic packages and tasks.

The second major difference between static and dynamic metrics is the weighting accorded to different features of the system. Consider the following passage of code:

```
procedure A is
        .
        .
        if x < y
              then B(x,y)
              else C(x,y)
        .
        .
end A;
```

A static analysis would treat the information flow relationship between A and B as equal to that of the information flow relationship between A and C. Suppose, however, that in execution over a substantial set of input data it was determined that the conditional test "x<y" was true in 90% of the cases. This additional knowledge about the dynamic relationships involving A, B and C would

lead us to assign a greater weight to the information flow between A and B since this flow occurs with much higher probability.

Frequency and Duration metrics are the easiest dynamic metrics to capture. Frequency simply refers to the number of times a procedure is called and duration is the amount of time each procedure executes. Using existing tools in a UNIX environment, frequency and duration metrics are easily generated. This information can then be incorporated in the reliability model.

Execution time monitoring has traditionally been used in testing, configuring of systems, hardware monitoring and debugging [PLAB81]. These all involve an interactive approach with the user examining the results from the monitor and determining what should be done next. From a dynamic metric standpoint, it is not necessary that an execution monitor interact with the user. This is because we do not wish to alter the target program, but measure it as it executes.

An execution monitor or profiler monitors program execution, gathers statistics and returns the results. The most common statistics collected are timings in the form of clock ticks spent in each unit of the program, and counts which are a count of the number of times each unit is executed. These units range from lines of code to subprograms.

Plattner [PLAB81] suggests four possibilities for the execution monitor:

1. the execution monitor interprets the target program,
2. generate a trap after each instruction so a trap handler can arbitrate between the execution monitor and the target program,
3. patch instructions into the target program, and
4. augment the system with a hardware device to trap when certain memory is accessed.

Bishop [Bism87] lists four ways to implement a profiler. First, the compiler can be modified to generate monitoring code. This is the method traditionally used by UNIX. Second, use a preprocessor to insert special code into the source program. Third, use a postprocessor to insert special code into the assembly language program. Finally, use an execution monitor as discusses before.

The dynamic metrics, collected by the execution monitor, play a key role in the development of the proposed reliability model. As illustrated above, the dynamic metrics provide the basis for a probabilistic weighting of complexity factors. This consideration of the probability of events within a system is necessary in assessing that system's reliability. Consider, for example, two systems which have one extremely complex component which has been subjected to the same degree of testing. In one system the component is frequently executed while in the other system the component is rarely executed. It would appear that the system where the the complex

component is rarely executed should be assigned a higher reliability measure because its execution avoids the potential faults latent in the complex component. By contrast, the other system would be assigned a lower reliability measure because it is frequently exposed to the reliability hazards of the complex component.

In summary, dynamic software metrics provide a weighting of software complexity factors based on run-time information. This run-time information could be acquired either through an execution monitor or an instrumentation of the system's code.

IV. Summary and Future Work

This paper describes our initial work on defining both a new reliability model and new class of software complexity metrics based on interface properties and dynamic execution characteristics. Our three year research plan is aimed at producing a new model/metric pair each year in the following order:

- A reliability model using existing static metrics and the parallel development of new interface metrics
- A reliability model incorporating the interface metrics and the parallel development of new dynamic metrics.
- A reliability model incorporating the dynamic metrics.

Each of the reliability model must, of course, be subjected to as rigorous a validation as possible.

Our near term goals for the reliability model development are:

- Conduct the first stage of the validation process described in Section II of this paper. This first stage uses a synthetic system whose complexity and error characteristics are determined by a statistical model calibrated from actual project data.
- Identify a small-scale system which can be used for the second step of the validation process and select the error seeding and test case generation strategies to be used.
- Identify a large-scale system with complete source code, test cases, and error history. While such a system may be difficult to procure for this

research project, this step represents the most realistic step of the validation process.

Our near term goals for the metrics development are:

- Completely specify the definition of the interface metrics and develop an interface metric generator to incorporate into the current metric analyzer.
- Verify the results of the interface metrics to see if they adequately measure the important properties.
- Automate the straightforward dynamic metrics for frequency and duration.

Acknowledgements

The authors would like to acknowledge the valuable contributions made to this work by their sponsors at the Software Productivity Consortium, including Chuck Davis and John Gaffney, and the other members of the research group at Virginia Tech: Bryan Chappell, and Matt Humphrey. We also give special thanks to Joy Davis for all of her work in assembling and revising this document.

References

[BISM87] Bishop, M., "Profiling under UNIX by Patching," **Software Practice and Experience,** October 1987, pp. 729-739.

[CONS86] Conte, S.D., Dunsmore, H. E., Shen, V. Y., **Software Engineering Metrics and Models,** Menlo Park, CA, The Benjamin/Cummings Publishing Company, Inc., 1986.

[GOFR87] Goff, R. **Complexity Measurement of a Graphical Programming Language and Comparison of a Graphical and Textual Design Language,** Masters Thesis, Virginia Tech, Department of Computer Science, June 1987.

[HALM77] Halstead, M., **Elements of Software Science,** New York, NY, Elsevier North Holland, Inc., 1977.

[HARH85] Hartson, H.R., **Advances in Human-Computer Interaction,** Norwood, NJ, Ablex Publishing Company, 1985.

[HENS79] Henry, S., **Information Flow Metrics for the Evaluation of Operating Systems' Structure,** Ph. D. Disertation, Iowa State University, Computer Science Department, 1979.

[HENS81a] Henry, S.M., Kafura, G.D., "Software Structure Metrics Based on Information Flow", IEEE **Transactions on Software Engineering,** September 1981.

[HENS81b] Henry, S.M., Kafura, G.D., Harris, K., "On the Relationships Among Three Software Metrics", **Performance Evaluation Review,** Vol. 10, No. 1, Spring 1981.

[HENS88a] Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information fo Researchers", **Journal of Systems and Software,** Vol. 8, pp. 3-12, January,1988.

[HENS88b] Henry, S.M., Goff, R., "Complexity Measurement of a Graphical Programming Language", submitted to **IEEE Software.**

[KAFD84] Kafura, D., J. Canning, and G. Reddy, *"The Independence of Software Metrics Taken at Different Life-Cycle Stages", Proceedings:* **Ninth Annual Software Engineering Workshop, NASA Goddard Space Flight Center,** November 28, 1984.

[LEWJ88] Lewis, J., **Using Software Quality Metrics and Iterative Enhancement to Promote Maintainability in Large Scale Real-Time System Software,** Masters Thesis, Virginia Tech, Department of Computer Science, will defend 1988.

[MCCT76] McCabe, T., "A Complexity Measure", **IEEE Transactions on Software Engineering,** December 1976.

[MCCC78] McClure, C. L., "A Model for Program Complexity Analysis", **Proceedings of the 3rd Conference on Software Engineering,** pp. 149-157, 1978.

[PLAB81] Plattner, B., Nievergelt, J., "Monitoring Program Execution: A Survery," **IEEE Computer,** November 1981, pp. 78-93.

[SELC87] Selig, C.L., **ADLIF - A Structured Design Language for Metric Analysis,** Masters Thesis, Virginia Tech, Department of Computer Science, August 1987.

[SHOM83] Shooman, Martin, **Software Engineering,** New York, NY, McGraw-Hill Book Company, 1983.

[WAKS88] Wake, S., **Predicting Maintainability with Software Quality Metrics,** Masters Thesis, Virginia Tech, Department of Computer Science, will defend 1988.