Software Engineering Baselines

Contract Number F30602-89-C-0082
(Data & Analysis Center for Software)

18 July 1996

PREPARED FOR:
Rome Laboratory
RL/C3C
Griffiss Business Park
Rome, NY 13441

PREPARED BY:

John Marciniak                      Robert Vienneau
ITT Industries                     ITT Industries
2560 Huntington Avenue             P.O. Box 1400
Alexandria, VA 22303               Rome, NY 13442-1400
data-analyst@dacs.dtic.mil

Acknowledgements:
The author would like to gratefully acknowledge comments on an earlier draft by Mr. Robert Vienneau.

Table of Contents  Abstract & Ordering

DoD DACS Home Page
TABLE OF CONTENTS

Title Page

1.0 INTRODUCTION

2.0 BASELINES

2.1 Productivity

   2.1.1 Functions Points
   2.1.2 Source Lines of Code

2.2 Reliability

   2.2.1 Fault Profiles
   2.2.2 Software Reliability Engineering
   2.2.3 Reliability Prediction

2.3 Complexity

3.0 REFERENCES

List of Tables

List of Figures

Appendices

APPENDIX A. ACRONYMS

APPENDIX B. SOFTWARE COST MODEL VENDORS

APPENDIX C. REFERENCES FOR SOFTWARE RELIABILITY

APPENDIX D. SOME VENDORS OF COMPLEXITY MEASUREMENT TOOLS
1.0 INTRODUCTION

Software measurement programs are of increasing interest in the DoD and industrial practice. These programs run the gambit of scope and purpose. They support the implementation and management of process improvement programs, such as those based on the Software Engineering Institute's Capability Maturity Model (CMM) and the NASA Goddard Space Flight Center Software Engineering Laboratory's Process Improvement Paradigm (PIP), as well as provide individual project management support.

Metrics, however, are a loosely used term. In some cases, they are used to refer to specific measurements, such as those of the complexity of implemented code. In other cases, they are more liberally applied to provide trend indicators, commonly called management indicators, such as the documentation completed according to a project schedule. Whatever the application, few commonly accepted quantitative baselines are known and documented for specific software system implementations.

The purpose of this report is to provide baseline information about a selected set of metrics, specifically productivity, complexity, and reliability. It is not a comprehensive treatment of metrics; indeed, that subject is treated in a number of texts and DoD initiatives. Some of these initiatives include the Joint Logistics Commanders' Practical Software Measurement (PSM) program, the SEI's Software Engineering Measurement and Analysis (SEMA) program (Carleton 92), and the U.S. Army's Software Test and Evaluation Panel (STEP) metrics. These initiatives, for the most part, provide methodologies for the implementation of metrics based on the concept of management indicators. All management indicators are based on the collection of elemental data that we call, in this report, metrics.

The purpose of a broad measurement program is to collect data to assess the progress of a project or the adequacy of a process. In a project, the objective is to assess the process of development and the attainment of the products of the project in order to evaluate the ability of the project to meet its product goals. In an organization, the measurement program further supports the assessment of the development and maintenance process in order to improve that process as part of an organization process improvement program.

When planning or implementing a project, the question of what is an accepted value for a metric often arises. A planner or manager may want to know what the expected value should be for the complexity of the design or implemented code, or the expected productivity of the development lifecycle. In this report, we present metrics in the three areas mentioned above, productivity, reliability, and complexity. We explain their definition and give ranges of values that may be expected based on current practice. The examples presented are illustrative of commonly accepted or popular metrics for each area, but not the only ones in each area. Hopefully, as additional data is collected and made available, this report can be enhanced to form a growing base of norms for measurement practice.
2.0 BASELINES

This section summarizes some basic software metrics, based mainly on the software engineering literature. Users of these results should be aware of current software measurement programs. Recent programs have not yet produced data that could be summarized in this report. A boomlet is currently occurring in software measurement with a number of commercial and government software measurement programs recently having been created. Influential government programs include:

- National Aeronautics and Space Administration Software Engineering Laboratory (NASA/SEL)¹, an early model for a software measurement organization
- National Software Data and Information Repository (NSDIR), initiated by Mr. Lloyd Mosemann, II, Deputy Assistant Secretary of the Air Force, Communications, Computers, & Support Systems (Chruscicki 95)
- Software Engineering Institute (SEI) Capability Maturity Model (CMM), which requires software measurement for higher maturity levels (Paulk 95). Furthermore, the SEI is beginning a metrics repository and is gathering data on the cost and benefits of the CMM.
- U. S. Army Software Test and Evaluation Panel (STEP) metrics (DA 92)
Footnotes

1 NASA / SEL is operated jointly by the NASA Goddard Space Flight Center (GSFC), Computer Science Corporation and the University of Maryland.

2 Some early cost models used size in terms of computer words or object code.

3 A two-tailed Mann-Whitney-Wilcoxon test shows a statistically significant difference between the two distributions of FPs at the 5% level, but not at the 1% level.

4 Person hours data in the Albrecht & Gaffney dataset were converted to person months at the rate of 160 person hours per person month.

5 An Analysis Of Variance (ANOVA) was used to determine that the Cobol projects in the Albrecht & Gaffney and Kemerer datasets could be combined to form one linear regression model. Linear regressions were performed separately on the Cobol projects in the two datasets. These regressions identified two outliers in the Albrecht & Gaffney dataset and one outlier in the Kemerer dataset. Linear regressions were rerun on the Cobol datasets with these outliers removed and on the Cobol projects in the combined dataset. An F test compared the alternate hypothesis model of separate regressions for the two datasets to the null hypothesis model of identical regression lines. This test was nonsignificant at the 10% level.

6 Productivity data on FP per person month were combined for the Cobol projects in the two datasets. The Mann-Whitney-Wilcoxon test was not significant at the 10% level.

7 Development mode has little influence on schedule length in COCOMO.

8 The graphs in Figures 2.2-1 and 2.2-2 are for illustration only, and do NOT show real data.

9 Many other models make this assumption. Others assume that faults that contribute more to the failure rate found earlier, or even more complicated relationships. These models do not exhibit a linear relationship between the failure rate and the expected number of failures.
2.1 Productivity

**Metric:** Productivity

**Measures:** The output produced for a unit input in a software organization.

**Related Metrics:** Size as measured by Function Points (FPs) or Source Lines of Code (SLOC), effort, schedule.

**Applications:** Assessing productivity of a project, new technologies, etc.; estimating size, effort, cost, and schedule in project planning.

**Definition:** FPs per person month or SLOC per person month. FPs are a weighted sum of the number of program inputs, outputs, user inquiries, files, and external interfaces.

**Range:**
- **Productivity:** From 2 to 23 FPs per Person Month with a median of 5.6 FPs per Person Month, or from 80-400 SLOC per Person Month.
- **Size:** From 100 to 2,300 FPs with a median of 993 FPs, or from 2 to 512 KSLOC.
- **Function Point Conversion:** From 6 to 320 SLOC per FP.

**Notes:** FPs are most applicable to Management Information Systems (MIS) and other business applications.

**For More Information:**

International Function Points User Group (IFPUG)
5009-28 Pine Creek Drive
Westerville, OH 43081-4899
Voice: (614) 895-7130; FAX: (614) 895-3466
E-mail: 102214.2013@compuserve.com
URL: [http://www.ifpug.org/ifpug](http://www.ifpug.org/ifpug)


Productivity, the output produced for a unit of input, is a crucial measure for software organizations. Management cannot assess the worth of process improvement efforts and the impact of new tools, techniques, and methodologies without some mechanism for measuring productivity, as well as quality. Software system size and productivity are often useful in project planning. Budgets and schedules can be estimated from size measures. Ideally, a size measure should be easily estimated in the requirements
phase of a software project.

Source Lines Of Code (SLOC) is an early measure of the software size, and a number of cost models were developed on this basis. Although relating to a software engineering perspective, this measure has a number of paradoxes. Functionally equivalent systems can vary in SLOC, depending on how tightly code is developed. Hence, measuring productivity by SLOC per person month rewards inefficient and sloppy coding. The expressiveness of source code varies with language level. Software can usually be coded in a higher level language with less SLOC. If used incautiously, SLOC productivity measures can show decreased productivity when, in fact, productivity has increased (Jones 86). Finally, estimates of SLOC made early in the lifecycle can exhibit great variability and may be inappropriate as the principal driver to a software cost model.

Allan Albrecht (79), collaborating with John Gaffney, Jr. (Albrecht 83), designed Function Points (FPs) to be a direct measure of functionality capable of measurement from data typically available during the requirements phase. Use of FPs is becoming more widespread, particularly among software cost modelers. FPs are most appropriate for information systems, but modifications have been proposed to adapt them for more general applications. For example, Capers Jones' Feature Points accounts for the algorithmic complexity typical of embedded systems.

This report presents engineering norms for software productivity in two forms. Section 2.1.1 discusses productivity with FP measures. Section 2.1.2 uses the more traditional SLOC measures. A number of software cost model vendors and consultants on software productivity measurement maintain proprietary databases. Appendix B lists contact information for the most well-known vendors.

**2.1.1 Functions Points**

Function Points (FPs) are a weighted sum of the number of inputs, outputs, user inquiries, files, and interfaces to a system. Table 2.1-1, adapted from (Albrecht 83), presents a worksheet summarizing the FP algorithm. The latest counting rules are defined in Release 3.0 (1990) of *Function Point Counting Practices Manual*, by the International Function Points Users Group (IFPUG).

<table>
<thead>
<tr>
<th>Count</th>
<th>Weighting Factor($C_i$)</th>
<th>Function Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Inputs</td>
<td>$X_1$</td>
<td>3</td>
</tr>
<tr>
<td>Number of Outputs</td>
<td>$X_2$</td>
<td>4</td>
</tr>
<tr>
<td>Number of User Inquiries</td>
<td>$X_3$</td>
<td>3</td>
</tr>
<tr>
<td>Number of Files</td>
<td>$X_4$</td>
<td>7</td>
</tr>
</tbody>
</table>
What are typical system sizes for systems in terms of FPs? Two small datasets seem to be frequently referenced by FP researchers - Allan Albrecht & John Gaffney's (Albrecht 83) and Chris Kemerer's (Kemerer 87). These datasets contain mostly business applications written in Cobol. Cost modeling vendors have developed their own databases, but these are proprietary and not generally available.

The sizes of the systems examined by Albrecht & Gaffney and Kemerer are from different populations. Systems in Kemerer's data, which were collected later, tend to be larger than those in Albrecht & Gaffney's data. Figure 2.1-1 shows the distribution of FPs in Kemerer's Cobol projects. These projects are medium size systems, ranging from 39 to 450 Thousand Source Lines Of Code (KSLOC). Values of the FP metric are read along the X axis, while the height of the boxes represents the proportion of systems with FPs within the indicated intervals. This graph helps the reader understand whether a system is large or small, based on the number of FPs in the system. For example, a system with 1,000 FPs is a medium size system.

Productivity is measured as FP produced per person month. Person months can be predicted from FP. The Albrecht & Gaffney and Kemerer datasets can be used to construct a relationship for predicting effort from FP and for examining the distribution of productivity. Figure 2.1-2 shows a linear relationship for predicting effort from FP for Cobol projects. Figure 2.1-3 presents a histogram showing the distribution of productivity. Readers of this report are encouraged to develop their own baselines from data collected in their environment.

It may be useful to map FPs to SLOC. The amount of SLOC represented by a single FP varies by
language, with a higher level language requiring less SLOC for each FP. Table 2.1-2, based on (Jones 86) shows estimates of the SLOC per FP.

Table 2.1-2: Sloc Per FP By Language

<table>
<thead>
<tr>
<th>Language</th>
<th>SLOC per FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>320</td>
</tr>
</tbody>
</table>
2.1.2 Source Lines of Code

Source Lines of Code (SLOC) provide a more traditional basis for assessing productivity. Barry Boehm's Constructive Cost Model (COCOMO) is an accepted open cost model that encapsulates productivity relationships based on SLOC (Boehm 81). Table 2.1-3 shows typical SLOC counts for a range of project sizes. The trend is for organizations to attempt bigger systems over time. Microsoft provides a commercial example. Microsoft Basic had 4 KSLOC in 1975, while the current version is approximately 500 KSLOC. Microsoft Word 1.0 had 27 KSLOC; Word is now about 2,000 KSLOC (Brand 95).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macro Assembler</strong></td>
<td>213</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>150</td>
</tr>
<tr>
<td><strong>Algol</strong></td>
<td>106</td>
</tr>
<tr>
<td><strong>Chill</strong></td>
<td>106</td>
</tr>
<tr>
<td><strong>Cobol</strong></td>
<td>106</td>
</tr>
<tr>
<td><strong>Fortran</strong></td>
<td>106</td>
</tr>
<tr>
<td><strong>Jovial</strong></td>
<td>106</td>
</tr>
<tr>
<td><strong>Pascal</strong></td>
<td>91</td>
</tr>
<tr>
<td><strong>RPG</strong></td>
<td>80</td>
</tr>
<tr>
<td><strong>PL/I</strong></td>
<td>80</td>
</tr>
<tr>
<td><strong>Modula-2</strong></td>
<td>71</td>
</tr>
<tr>
<td><strong>Ada</strong></td>
<td>71</td>
</tr>
<tr>
<td><strong>Prolog</strong></td>
<td>64</td>
</tr>
<tr>
<td><strong>Lisp</strong></td>
<td>64</td>
</tr>
<tr>
<td><strong>Forth</strong></td>
<td>64</td>
</tr>
<tr>
<td><strong>Basic</strong></td>
<td>64</td>
</tr>
<tr>
<td><strong>Logo</strong></td>
<td>53</td>
</tr>
<tr>
<td><strong>4th Generation Database</strong></td>
<td>40</td>
</tr>
<tr>
<td><strong>Strategem</strong></td>
<td>35</td>
</tr>
<tr>
<td><strong>APL</strong></td>
<td>32</td>
</tr>
<tr>
<td><strong>Objective - C</strong></td>
<td>26</td>
</tr>
<tr>
<td><strong>Smalltalk</strong></td>
<td>21</td>
</tr>
<tr>
<td><strong>Query Languages</strong></td>
<td>16</td>
</tr>
<tr>
<td><strong>Spreadsheet Languages</strong></td>
<td>6</td>
</tr>
</tbody>
</table>

**Table 2.1-3: Basic COCOMO Size Ranges**
COCOMO describes three "modes" of software development- organic, semidetached, and embedded.

In the organic mode, relatively small software teams develop software in a highly familiar, in-house environment. Most people connected with the project have extensive experience in working with related systems within the organization, and have a thorough understanding of how the system under development will contribute to the organization's objectives...An organic-mode project is relatively relaxed about the way the software meets its requirements and interface specifications...

The semidetached mode of software development represents an intermediate stage between the organic and embedded modes. "Intermediate" may mean either of two things:

1. An intermediate level of the project characteristics
2. A mixture of the organic and embedded mode characteristics.

The major distinguishing factor of an embedded-mode software project is a need to operate within tight constraints. The product must operate within (is embedded in) a strongly coupled complex of hardware, software, regulations, and operational procedures, such as an electronic funds transfer system or an air traffic control system. In general, the costs of changing the other parts of this complex are so high that their characteristics are considered essentially unchangeable, and the software is expected both to conform to their specifications, and to take up the slack on any unforeseen difficulties encountered or changes required within the other parts of the complex...The embedded-mode project is generally charting its way through unknown territory to a greater extent than the organic-mode project. This leads the project to use a much smaller team of analysts in the early stages...Once the embedded-mode project has completed its product design, its best strategy is to bring on a very large team of programmers to perform detailed design, coding, and unit testing in parallel. (Boehm 81)

Figure 2.1-4 shows the effort required for each mode as a function of size in KSLOC. These are exponential curves in COCOMO, not straight lines. Figure 2.1-5 shows schedule as a function of KSLOC \(7\), and Figure 2.1-6 shows productivity. COCOMO modifies these curves with certain "multipliers" reflecting product, computer platform, personnel, and project attributes.

<table>
<thead>
<tr>
<th>Size</th>
<th>KSLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>2</td>
</tr>
<tr>
<td>Intermediate</td>
<td>8</td>
</tr>
<tr>
<td>Medium</td>
<td>32</td>
</tr>
<tr>
<td>Large</td>
<td>128</td>
</tr>
<tr>
<td>Very Large</td>
<td>512</td>
</tr>
</tbody>
</table>

Table 2.1-3: Basic COCOMO Size Ranges
Figure 2.1-4: Effort as a Function of KSLOC

Figure 2.1-5: Schedule as a Function of KSLOC
Figure 2.1-6: Productivity as a Function of KSLOC
2.2 Reliability

<table>
<thead>
<tr>
<th>Metric: Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source: Original sources are by Zygmund Jelinski and Paul Moranda (1972) and by Martin Shooman (1972).</td>
</tr>
<tr>
<td>Measures: User-oriented quality as shown by failure behavior of a system.</td>
</tr>
<tr>
<td>Related Metrics: Mean Time Between Failure (MTBF), Failure Intensity, Failure Rate, Fault Density</td>
</tr>
<tr>
<td>Applications: Quality assurance, planning and monitoring system testing, operations and maintenance planning.</td>
</tr>
<tr>
<td>Definition: The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system, as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered.</td>
</tr>
<tr>
<td>Range: Operational failure rate ranges at least from 3x10⁻⁶ to 55x10⁻⁶ Failures per CPU Second. MTBF ranges at least from 5.1 to 92.6 CPU hours. Fault density at the start of System Test ranges from 1 to 10 Faults per KSLOC, with an average of 6 Faults per KSLOC. KSLOC is counted here as delivered executable source lines, excluding reused code, data declarations, comments, etc. Expected number of faults removed per failure: 0.955 Faults</td>
</tr>
</tbody>
</table>

For More Information:


Although many factors are used to describe software quality - for example, portability, usability, and maintainability - reliability is most commonly used. Because it is based on the occurrence of observable problems or failures in the product, it is among the easiest of the quality factors to measure. It is also an easy concept for a user to relate to - the user expects the product to be error free, thus, highly reliable. Maintainability and availability are two other quality factors closely related to reliability. Maintainability is often measured by Mean Time To Repair (MTTR) and availability by the ratio of the Mean Time
Between Failures (MTBF) to the sum of MTBF and MTTR. The calculation of each of these is again, based on tracking failures.

Reliability is one of the best known and oldest metrics. Software managers typically tracked software bugs even before software reliability concepts were explicitly formalized. Collecting and tracking so-called Software Problem Reports (SPRs) or Software Trouble Reports (STRs) is still a common procedure for monitoring the software development process to achieve reliability goals (Section 2.2.1). Additionally, much work since the seventies has been directed toward conceptual clarification and modeling of software reliability. This has produced a consensus view of the software failure process, which is reflected in the definitions in Table 2.2-1. Basically, failures, observed while operating software, are caused when the software enters a state in which a fault or defect in the product changes the resulting output. Since inputs, which effect state changes, and the location of faults are not completely known a priori, software reliability is modeled as a stochastic process. This conceptual understanding provides the foundation for an engineering technology for modeling and controlling software reliability (Section 2.2.2), as well as procedures for predicting reliability (Section 2.2.3). This technology can only be presented very briefly here. Appendix C provides additional resources for Software Reliability Engineering.

### 2.2.1 Fault Profiles

Reliability was originally monitored by tracking faults found throughout the lifecycle. Faults are collected and cataloged in forms known by various names, such as Software Trouble Reports (STRs) or Software Problem Reports (SPRs). Various graphical displays of SPR data can provide management insight into the software development process. For example, Figures 2.2-1 and 2.2-2 show two of five graphs defined by the United States Army's Software Test and Evaluation Panel (STEP) "Fault Profiles" metric.8

<table>
<thead>
<tr>
<th><strong>Table 2.2-1: Definitions for Software Reliability</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error</strong> (1) A discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. (2) Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification. This is not a preferred usage.</td>
</tr>
<tr>
<td><strong>Failure</strong> (1) The inability of a system or system component to perform a required function with specified limits. A failure may be produced when a fault is encountered and a loss of the expected service to the user results. (2) The termination of the ability of a functional unit to perform its required function. (3) A departure of program operation from program requirements.</td>
</tr>
<tr>
<td><strong>Failure Rate</strong> (1) The ratio of the number of failures of a given category or severity to a given period of time; for example, failures per month. Synonymous with failure intensity. (2) The ratio of the number of failures to a given unit of measure; for example, failures per unit of time, failures per number of transactions, failures per number of computer runs.</td>
</tr>
<tr>
<td><strong>Fault</strong> (1) A defect in the code that can be the cause of one or more failures. (2) An accidental condition that causes a functional unit to fail to perform its required function. Synonymous with bug.</td>
</tr>
</tbody>
</table>
**Quality** The totality of features and characteristics of a product or service that bears on its ability to satisfy given needs.

**Software Quality** (1) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, to conform to specifications. (2) The degree to which software possesses a desired combination of attributes. (3) The degree to which a customer or user perceives that software meets his or her composite expectations. (4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

**Software Reliability** (1) The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system, as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. (2) The ability of a program to perform a required function under stated conditions for a stated period of time.

Definitions extracted from (AIAA 92)
Graphs such as these can be examined throughout a software project to detect trends, to assess
development progress, and for comparisons with similar projects. Often, faults are summarized by
normalizing against the size of the product. For example, Table 2.2-2 shows faults per Thousand Source
Lines Of Code (KSLOC). The reader can use Table 2.2-2 as typical fault density values in the absence of
further data from his environment.

Table 2.2-2: Fault Densities

<table>
<thead>
<tr>
<th>Application</th>
<th>Systems</th>
<th>KSLOC</th>
<th>Average Faults/KSLOC</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airborne</td>
<td>7</td>
<td>541</td>
<td>12.8</td>
<td>9.4</td>
</tr>
<tr>
<td>Strategic</td>
<td>21</td>
<td>1,794</td>
<td>9.2</td>
<td>14.0</td>
</tr>
<tr>
<td>Tactical</td>
<td>5</td>
<td>88</td>
<td>7.8</td>
<td>6.1</td>
</tr>
<tr>
<td>Process Control</td>
<td>2</td>
<td>140</td>
<td>1.8</td>
<td>0.3</td>
</tr>
<tr>
<td>Production Center</td>
<td>12</td>
<td>2,575</td>
<td>8.5</td>
<td>9.5</td>
</tr>
<tr>
<td>Developmental</td>
<td>4</td>
<td>97</td>
<td>12.3</td>
<td>9.3</td>
</tr>
<tr>
<td><strong>Total/Average</strong></td>
<td><strong>51</strong></td>
<td><strong>5,236</strong></td>
<td><strong>9.4</strong></td>
<td><strong>11.0</strong></td>
</tr>
</tbody>
</table>

(Reproduced from McCall 87)
2.2.2 Software Reliability Engineering

While tracking Software Problem Reports is a useful methodology for monitoring the software development process, it does not result in a predictive user-oriented reliability metric. Software reliability models, which relate failure data to a statistical model of the software failure process, are used to specify, predict, estimate, and assess the reliability of software systems.

The discipline of Software Reliability Engineering evolved from the development and application of these probabilistic models of the software failure process.

Many models are in use with good results. Customarily, these models are applied during system test or maintenance by collecting failure data, fitting the model, and updating results based on additional data. If the fit is good, the model can be used with relative confidence to provide an assessment of current reliability or to predict future failure behavior. For example, if a model has been proven accurate time and again for previous increments of a software product, then its use results in trust in new results. For explicitness, this report describes Musa's Basic Execution Time Model, one of the most well-accepted software reliability models.

Suppose a software system is operating in an environment with an unchanging operational profile. In other words, the distribution of the types of user demands or requests for system capabilities do not vary with time (Musa 93). Furthermore, suppose no changes are made to the software during operations. Then the software might be modeled with a constant failure rate $\hat{\lambda}$. Then the expected number of failures in a time interval is proportional to the length of that time interval. The probability that the software will operate without failure, the reliability $R(\tau)$, becomes smaller the longer the time period under consideration. Figure 2.2-3 shows this relationship in which reliability decreases exponentially with execution time $\tau$. For systems with a constant failure rate, the Mean Time Between Failure, calculated as the reciprocal of the failure rate, is often used to summarize reliability.
Software will be changed during system test to remove faults uncovered by operating the system. Removing observable faults will presumably increase the reliability of the software. Therefore, system test is modeled to exhibit reliability growth. The Musa Basic Execution Time Model is based on the assumption that all faults contribute equally to the failure rate. Thus, the failure rate is a decreasing linear function of the expected number of faults $\mu(\tau)$ (Figure 2.2-4). Basic parameters of the Musa model are $V_0$, the expected number of failures required to remove all faults, and $\beta$, the expected decrease in the failure rate per failure. These parameters define other functions of interest. Figure 2.2-5 shows the cumulative expected number of failures, and Figure 2.2-6 shows how the failure rate varies with time.
Given the parameters of the model, $\beta$ and $V_0$, one can determine how long system test must proceed until any reliability goal is met. Confidence bounds should be used for a reliability stopping rule instead of point estimates. For example, testing might stop when a 90% upper confidence bound on the number of remaining failures is below a required target bound. Alternatively, testing could stop when total lifecycle cost is minimized. The cost of a failure is greater in the field than in system test. The marginal benefit of testing for an increment of execution time is the expected decrease in the cost of failure, accounting for the expected number of failures in that increment. The marginal cost of testing is the resources needed to test for an increment of execution time. To minimize total cost, testing should proceed until the marginal benefit falls below the marginal cost (Vienneau 91).
Figure 2.2-5: Expected Failures vs. Time

Figure 2.2-6: Failure Rate vs. Time

\[
\mu(\tau) = v(1 - e^{-\beta \tau})
\]

\[
\lambda = \beta v_0 e^{-\beta \tau}
\]
2.2.3 Reliability Prediction

The Musa Basic Execution Time Model describes the failure behavior of a system during operations and system test, given model parameters $\beta$ and $\nu$, the expected decrease in the failure rate per failure and the number of failures needed to remove all faults, respectively. Reliability can be predicted by estimating these parameters from other parameters known early in the lifecycle. A recent commercial standard, the *American National Standard Recommended Practice for Software Reliability* (AIAA 93), and a draft military standard, *Military Handbook Hardware/Software Reliability Assurance and Control* (DRAFT) (RL 91), contain similar procedures for using the Musa model to predict reliability. These prediction procedures are based on a set of three parameters, from which the Musa model parameters can be derived:

- The initial failure rate, $\lambda_0$
- The number of faults in the program at start of system test, $\omega_0$
- The fault reduction factor, $B$

The failure rate per fault, $\phi$, is defined in terms of these parameters:

$$\phi = \frac{\lambda}{\omega_0}$$  \hspace{1cm} (2.2-1)

The failure rate per fault, $\phi$, differs from the failure rate per failure, $\beta$, because of errors made during debugging. Sometimes, debugging after a failure will not reduce the number of faults. The fault reduction factor, $B$, models imperfect debugging. The fault reduction factor is the expected number of faults removed per failure. The fault reduction factor is generally less than unity. Equations 2.2-2 and 2.2-3 show some relationships between these parameters:

$$\beta = B \cdot \phi = \frac{\lambda}{\omega_0}$$ \hspace{1cm} (2.2-2)

$$\nu = \frac{\omega}{\phi}$$ \hspace{1cm} (2.2-3)

The draft military handbook recommends the use of a value for $B$ of 0.955 faults per failure.

The number of inherent faults, $\omega_0$, is predicted as the product of the size of the software and the expected fault density. Size is measured in KSLOC, excluding reused and non-executable source code. Fault density is the number of faults per KSLOC. A typical range for fault density at the start of system test is from 1 fault per KSLOC to 10 faults per KSLOC. The lower value is for programmers with extensive background in the specific application in a highly disciplined environment (AIAA 92). The draft military handbook recommends the use of 6 faults per KSLOC (RL 91). Table 2.2-2 (in Section 2.2.1) presents some empirical data on fault densities based on operational data from a range of systems.

The draft military handbook also includes prediction procedures for the initial failure rate $\lambda_0$. These procedures become increasingly detailed as the lifecycle proceeds. The initial failure rate is estimated...
during the requirements phase:

\[ \lambda_s = k \omega_s r \]  \hspace{1cm} (2.2-4)

where \( \omega_s \) is the number of inherent faults predicted above, \( r \) is the average processor speed in instructions per second, \( \lambda_s \) is the number of object instructions, and \( k \) is a proportionality constant, the "fault exposure ratio". The average processor speed, a characteristic of the machine and the instruction mix, can be found by benchmarking. The number of object instructions is found by multiplying the number of SLOC by a code expansion ratio. Representative code expansion ratios are shown in Table 2.2-3. In the absence of historical data for a particular environment, one should use \( 4.20 \times 10^{-7} \) failures per fault as a value for \( k \). Recommended values for constants used in predicting the parameters of the Musa Basic Execution Time Model are summarized in Table 2.2-4. Table 2.2-5 shows an example of how to combine these results to predict the MTBF of a moderate size software system.

### Table 2.2-3: Code Expansion Ratios

<table>
<thead>
<tr>
<th>Programming Language</th>
<th>Expansion Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>1</td>
</tr>
<tr>
<td>Macro Assembler</td>
<td>1.5</td>
</tr>
<tr>
<td>C</td>
<td>2.5</td>
</tr>
<tr>
<td>Cobol</td>
<td>3</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>3</td>
</tr>
<tr>
<td>Jovial</td>
<td>3</td>
</tr>
<tr>
<td>Ada</td>
<td>4.5</td>
</tr>
</tbody>
</table>

From (RL 91)

### Table 2.2-4: Some Parameters for Reliability Prediction

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Expected Value</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault Reduction Factor, ( B )</td>
<td>0.955 Faults per Failure</td>
<td></td>
</tr>
<tr>
<td>(Average Number of faults corrected per failure)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault Density</td>
<td>6 Faults per KSLOC</td>
<td>1 to 10</td>
</tr>
<tr>
<td>Fault Exposure Ratio, ( \kappa )</td>
<td>( 4.20 \times 10^{-7} ) Failures per Fault</td>
<td>( 1.41 \times 10^{-7} ) to ( 10.6 \times 10^{-7} )</td>
</tr>
</tbody>
</table>
### System Characteristics

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System size:</strong></td>
<td>50 KSLOC</td>
</tr>
<tr>
<td><strong>Fault Density:</strong></td>
<td>6 Faults per KSLOC (Average)</td>
</tr>
<tr>
<td><strong>Average Processor Speed:</strong></td>
<td>$r = 100 \times 10$ Instructions per Second</td>
</tr>
<tr>
<td><strong>Code Expansion Ratio:</strong></td>
<td>3 Object Instructions per SLOC (Cobol)</td>
</tr>
<tr>
<td><strong>Fault Exposure Ratio:</strong></td>
<td>$\kappa = 4.20 \times 10^{-7}$ Failures per Fault</td>
</tr>
<tr>
<td><strong>Fault Reduction Factor:</strong></td>
<td>$B = 0.955$ Faults per Failure</td>
</tr>
<tr>
<td><strong>System Test Period:</strong></td>
<td>1000 CPU Hours</td>
</tr>
</tbody>
</table>

### Calculations

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object Instructions:</strong></td>
<td>$I = 50,000 \times 3 = 150,000$ Instructions</td>
</tr>
<tr>
<td><strong>Inherent Faults:</strong></td>
<td>$\omega_a = 50 \times 6 = 300$ Faults</td>
</tr>
<tr>
<td><strong>Initial Failure Rate:</strong></td>
<td>$\lambda_a = (4.20 \times 10^{-7})(300)(100 \times 10^{-7})/(150,000)$</td>
</tr>
<tr>
<td></td>
<td>$\lambda_a = 0.084$ Failures per Second</td>
</tr>
<tr>
<td></td>
<td>$\lambda_a = 2.3 \times 10^{-5}$ Failures per Hour</td>
</tr>
<tr>
<td><strong>Failure Rate per Fault:</strong></td>
<td>$\phi = (2.3 \times 10^{-5})/(300)$</td>
</tr>
<tr>
<td></td>
<td>$\phi = 7.8 \times 10^{-8}$ Failures per Hour per Fault</td>
</tr>
</tbody>
</table>

### Musa Basic Execution Time Model Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expected Number of Failures:</strong></td>
<td>$\nu_a = (300)/(0.955) = 314$ Failures</td>
</tr>
<tr>
<td><strong>Failure Rate per Failure:</strong></td>
<td>$\beta = (0.955)(7.8 \times 10^{-8})$</td>
</tr>
<tr>
<td></td>
<td>$\beta = 7.4 \times 10^{-8}$ Failures per Hour per Failure</td>
</tr>
<tr>
<td><strong>Failure Rate at Release:</strong></td>
<td>$\lambda_a = (7.4\times10^{-8})(314) e^{-0.000074}$</td>
</tr>
<tr>
<td></td>
<td>$\lambda_a = 2.3 \times 10^{-5}$ Failures per CPU Hour</td>
</tr>
<tr>
<td><strong>MTBF at Release:</strong></td>
<td>43,000 CPU Hours</td>
</tr>
</tbody>
</table>
2.3 Complexity

| Metric: Complexity |
| Measures: Size, control flow, data structures, intermodule structure, etc. |
| Related Metrics: Source Lines of Code (SLOC), McCabe's Cyclomatic Complexity metric, Halstead's Software Science metrics, Henry and Kafura's Information Flow metric, test coverage metrics. |
| Applications: Quality assurance during development, prediction of operational characteristics, test planning. |
| Definition: Cyclomatic complexity is the graph theoretical complexity of a computer program's flowchart. In other words, the number of areas enclosed by a flowchart. |
| Range: Usually consciously determined by management decision. Cyclomatic complexity must be positive with an upper bound of 10 being recommended. |
| Notes: Most well-developed for application to individual subroutines. |

Productivity is chiefly a management concern, while reliability is a quality factor directly visible to users of software systems. These externally visible attributes of software processes and products are strongly influenced by engineering attributes of software such as complexity. Well-designed software exhibits a minimum of unnecessary complexity. Unmanaged complexity leads to software difficult to use, maintain, and modify. It causes increased development costs and overrun schedules. But certain elements of software are inherently complex - conformance to arbitrary external interfaces, pressure to adapt to changing user requirements, lack of obvious representations for visualizing software (Brooks 87).

Controlling and measuring complexity is a challenging engineering, management, and research problem. Metrics have been created for measuring various aspects of complexity such as sheer size, control flow, data structures, and intermodule structure. The most well-accepted are probably Source Lines of Code (Section 2.1.2) and Cyclomatic complexity (McCabe 76). Complexity metrics not otherwise discussed here include Halstead's Software Science metrics and Henry and Kafura's Information Flow metric. Halstead's metrics are controversial. Henry and Kafura's metric is one of the first of a family addressing design complexity, the complexity embedded in connections among modules. Although important, measurement of this aspect of complexity, unlike the measurement of complexity within a single module, is still a research issue.

Cyclomatic complexity, \( V(G) \), applies graph theory to measure control flow complexity. Intuitively, the metric is the number of connected regions in a flowchart, for a subroutine with one entrance and one exit. Figure 2.3-1 shows a sample flowchart for which McCabe's metric is 3. McCabe's metric can also be
calculated from the number of edges and nodes in the flowchart. For theoretical reasons, an imaginary edge is first added from the terminal node to the initial node. With this change, McCabe's metric is calculated as:

$$V(G) = (\# \text{ Edges}) - (\# \text{ Nodes}) + 1$$  (2.3-1)

There are 7 edges and 5 nodes in the example. So once again, the cyclomatic complexity metric is 3.

Figure 2.3-1: A Flow Chart Example

Complexity metrics are collected by a static analyzer from source code (Appendix D). They can be collected from designs if designs are written in a formal language, that is, in a Program Design Language (PDL). Their main use is in quality assurance. Quantifiable guidelines can be enforced for all modules. For example, some organizations mandate that all subroutines contain fewer lines of code than will fit on a page and have a Cyclomatic metric less than 10, with special procedures being needed to obtain a waiver. For example, a waiver might be granted to a subroutine that is one large & statement. Mr. George Stark et. al. (94) describe an application of Cyclomatic complexity in which individual routines are allowed to exceed a metric value of 10, but the distribution of the metric over all subroutines in a system must follow certain guidelines. Figure 2.3-2 graphically displays these guidelines. One can determine the fraction of subroutines below a specified metric value as a function of the metric value. The guidelines require that this Cumulative Distribution Function (CDF) exceed the lower curve in Figure 2.3-2, and it is recommended that they exceed the curve separating the "Straightforward" and "Standard" regions.

Figure 2.3-2 illustrates these guidelines by applying it to the Common Ada Missile Packages (CAMP).
CAMP is a collection of reusable Ada packages consisting of over 100,000 Source Lines of Code organized into 2,507 Ada packages in 10 categories (CAMP 87). The curve for CAMP in Figure 2.3-2 is derived from measurements of 1,474 Ada procedures, functions, or tasks in CAMP (Staff 92). Note that the vast majority of CAMP subprograms had a cyclomatic complexity lower than the recommended upper bound of 10, and that the CAMP curve lies in the "Straightforward" region for all but the smallest values of Cyclomatic complexity. So CAMP is a straightforward system by this measurement. This curve for CAMP cannot be taken as representative of all systems since the distribution of Cyclomatic complexity varies significantly across systems, even systems in the same programming language (Staff 92).

![Figure 2.3-2: Distribution of Cyclomatic Complexity](image)
3.0 REFERENCES


(Staff 92) Staff, Ada Usage Profiles, Kaman Sciences Corporation, October 30, 1992.


APPENDIX A: ACRONYMS

ANOVA - Analysis Of Variance
CAMP - Common Ada Missile Packages
CDF - Cumulative Distribution Function
CMM - Capability Maturity Model
COCOMO - Constructive Cost Model
CPU - Central Processing Unit
CSC - Computer Sciences Corporation
CSCI - Computer Software Configuration Item
CSU - Computer Software Unit
DoD - Department of Defense
FP - Function Point
GSFC - Goddard Space Flight Center
IFPUG - International Function Points User's Group
KSLOC - Thousands of Source Lines of Code
MIS - Management Information System
MTBF - Mean Time Between Failures
MTTR - Mean Time To Repair
NASA/SEL - National Aeronautics and Space Administration Software Engineering Laboratory
NSDIR - National Software Data and Information Repository
PDL - Program Design Language
PIP - Process Improvement Paradigm
PSM - Practical Software Measurement
SEI - Software Engineering Institute
SEMA - Software Engineering Measurement and Analysis
SPR - Software Problem Report
SRE - Software Reliability Engineering
STEP - Software Test and Evaluation Panel
STR - Software Trouble Report
SLOC - Source Lines of Code
### Appendix B: Software Cost Model Vendors

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Address</th>
<th>Contact Details</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galorath Associates, Inc.</td>
<td>GA Seer Technologies Division</td>
<td>PO Box 90579, Los Angeles, CA 90009</td>
<td><a href="mailto:info@galorath.com">info@galorath.com</a>; <a href="http://www.galorath.com">www.galorath.com</a></td>
</tr>
<tr>
<td>Lockheed-Martin Price Systems</td>
<td>700 East Gate Drive - Suite 200, Mount Laurel, NJ 08054</td>
<td>Voice: (800) 437-7423; FAX: (609) 866-6789</td>
<td><a href="mailto:Bruce.Fad@DEN.MMC.Com">Bruce.Fad@DEN.MMC.Com</a></td>
</tr>
<tr>
<td>Quantitative Software Management, Inc.</td>
<td>2000 Corporate Ridge - Suite 900, McLean, VA 22102</td>
<td>Voice: (800) 424-6755; FAX: (703) 749-3795</td>
<td><a href="mailto:76274.72@compuserve.com">76274.72@compuserve.com</a> ; <a href="http://www.qsm.com">www.qsm.com</a></td>
</tr>
<tr>
<td>Rubin Systems, Inc.</td>
<td>5 Winterbottom Lane, Pound Ridge, NY 10576</td>
<td>Voice: (914) 764-4931; FAX: (914) 764-0536</td>
<td><a href="mailto:71031.377@compuserve.com">71031.377@compuserve.com</a></td>
</tr>
<tr>
<td>Softstar Systems</td>
<td>Dan Ligett, P.O. Box 1360, Amherst, NH 03031</td>
<td>Voice: (603) 672-0987; FAX: (603) 672-3460</td>
<td><a href="mailto:info@softstarltems.com">info@softstarltems.com</a> ; <a href="http://www.SoftstarSystems.com">www.SoftstarSystems.com</a></td>
</tr>
</tbody>
</table>
APPENDIX C: REFERENCES FOR SOFTWARE RELIABILITY

**Standards**

- American National Standard Recommended Practice for Software Reliability (AIAA 92)
- Military Handbook Hardware/Software Reliability Assurance and Control (RL 91)
- IEEE Standard Dictionary of Measures to Produce Reliable Software (IEEE 88a)
- Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software (IEEE 88b)

**Textbooks**

- Software Reliability: Measurement, Prediction, Application (Musa 87)

**Software Tools**

Computer Aided Software Reliability Estimation (CASRE) Available from:

- COSMIC
- University of Georgia
- 382 East Broad Street
- Athens, GA 30602-4272
- Phone: (706) 542-3265; Fax: (706) 542-4807
- Internet: service@cossack.cosmic.uga.edu
- Bitnet: COSMIC@UGA
- URL: http://www.cosmic.uga.edu/
LIST OF TABLES

- 2.1-1. Function Points Worksheet
- 2.1-2. Sloc Per FP By Language
- 2.1-3. Basic COCOMO Size Ranges
- 2.2-1. Definitions for Software Reliability
- 2.2-2. Fault Densities
- 2.2-3. Code Expansion Ratios
- 2.2-4. Some Parameters for Reliability Prediction
- 2.2-5. Example Prediction

LIST OF FIGURES

- 2.1-1. Kemerer Cobol Data
- 2.1-2. Effort versus Function Points
- 2.1-3. Productivity In Cobol Projects
- 2.1-4. Effort as a Function of KSLOC
- 2.1-5. Schedule as a Function of KSLOC
- 2.1-6. Productivity as a Function of KSLOC
- 2.2-1. Cumulative Software Problem Reports
- 2.2-2. Average SPR Age 20.
- 2.2-3. Reliability Function
- 2.2-4. Failure Rate vs. Expected Failures
- 2.2-5. Expected Failures vs. Time
- 2.2-6. Failure Rate vs. Time
- 2.3-1. A Flow Chart Example
- 2.3-2. Distribution of Cyclomatic Complexity
# APPENDIX D: SOME VENDORS OF COMPLEXITY MEASUREMENT TOOLS

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Address</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaMat</td>
<td>Dynamic Research Corporation</td>
<td>60 Frontage Road, Andover, MA 01810&lt;br&gt;Voice: (800) 522-7321; Voice: (508) 475-9090&lt;br&gt;FAX: (508) 475-2157&lt;br&gt;E-mail: <a href="mailto:cmcquire@s1.drc.com">cmcquire@s1.drc.com</a></td>
</tr>
<tr>
<td>Amadeus</td>
<td>Amadeus Software Research, Inc.</td>
<td>President: Richard W. Selby&lt;br&gt;10 Young Court, Irvine, CA 92715&lt;br&gt;Voice: (714) 725-6400&lt;br&gt;FAX: (714) 725-6411&lt;br&gt;E-mail: <a href="mailto:selby@amadeus.com">selby@amadeus.com</a></td>
</tr>
<tr>
<td>AdaQuest</td>
<td>General Research Corporation</td>
<td>PO Box 6770, 5383 Hollister Ave, Santa Barbara, CA 93160-6770&lt;br&gt;Voice: (805) 964-7724</td>
</tr>
<tr>
<td>McCabe &amp; Associates, Inc.</td>
<td>Mr. Thomas McCabe&lt;br&gt;Twin Knolls Professional Park, 5501 Twin Knolls Road, Suite 111, Columbia, Maryland 21045&lt;br&gt;Voice: (800) 638-6316&lt;br&gt;FAX: (410) 995-1528&lt;br&gt;URL: <a href="http://www.mccabe.com">http://www.mccabe.com</a></td>
<td></td>
</tr>
</tbody>
</table>
SET Laboratories Incorporated
PO Box 868
Mulino, OR 97042
Voice: (503) 289-4758
FAX: (503) 829-7220
URL: http://www.molalla.net/~setlabs/

Logiscope
Verilog, Inc.
3010 LBJ Freeway, Suite 900
Dallas, Texas 75234
Voice: (214) 241-6595
FAX: (214) 241-6594
E-mail: info@logtech.com
Abstract:
Software measurement programs are of increasing interest in the DoD and industrial practice. These programs run the gamut of scope and purpose. The programs support the implementation and management of process improvement programs.

The purpose of this report is to provide baseline information about a selected set of metrics, specifically productivity, complexity, and reliability. The question of what is an accepted value for a metric often arises when planning or implementing a project. A planner or manager may want to know what the expected value should be for the complexity of the design or implemented code, or the expected productivity of the development life cycle.

Productivity, reliability, and complexity metrics are presented in the report. Definitions and the range of values that may be expected based on current practice are provided for each metric. The examples are illustrative of commonly accepted or popular metrics for each area, but not the only metrics for each area. As additional data is collected and made available we hope to enhance this report to form a growing base of norms for measurement practice.

Ordering Information:
A bound version of this report, (see full table of contents) is available for $30 and may be ordered from the DACS Product Orderform.