Product Models and Metrics

There are a large number of product types: requirements documents, specifications, design, code, specific components, test plans, ...

There are many abstractions of these products that depend on different characteristics
- logical, e.g., application domain, function
- static, e.g. size, structure
- dynamic, e.g., MTTF, test coverage
- use and context related, e.g., design method used to develop

Product models and metrics can be used to
- evaluate the process or the product
- estimate the cost of quality of the product
- monitor the stability or quality of the product over time
Product Models and Metrics

Logical Characteristics

The logical characteristic application can be measured on a nominal scale: flight software, ground support software, ...

The logical characteristic function can be represented as
- a mathematical function abstraction for a program, e.g., $y = f(x)$, a state function abstraction for a module
  or
- a nominal class, e.g., the component represents a mathematical function, data structure, ...

Product Models and Metrics

Static Characteristics

We can divide the static product the characteristics into three basic classes
- Size
- Structure, e.g.,
  - Control Structure
  - Data Structure

Size attempts to model and measure the physical size of the product

Structure models and metrics attempt to capture some aspect of the physical structure of the product, e.g.,
- Control structure metrics measure the control flow of the product
- Data structure metrics measure the data interaction of the product

There are mixes of these metrics, e.g., that deal with the interaction between control and data flow.
Product Models and Metrics

Size

There are many size models and metrics, depending on the product, e.g.,
- source code: lines of code, number of modules
- executables: space requirements, lines of code
- specification: function points
- requirements: number of requirements, pages of documentation
- modules: operators and operands

Size metrics can be used accurately at different points in time
- lines of code is accurate after the fact but can be estimated
- function points can be calculated based upon the specification

Size metrics are often used to
- characterize the product
- evaluate the effect of some treatment variable, such as a process
- predict some other variable, such as cost

Lines of Code Metrics

Lines of code can be measured as:
- all source lines
- all non-blank source lines
- all non-blank, non-commentary source lines
- all semi-colons
- all executable statements

The definition depends on the use of the metric, e.g.,
- to estimate effort we might use all source lines as they all take effort
- to estimate functionality we might use all executable statements as they come closest to representing the amount of function in the system

Lines of code vary with the language being used
- are the most common, durable, cheapest metric to calculate
- are most often used to characterize the product and predict effort
PRODUCT METRICS

Documentation Metrics

(Law of the Core Dump) The thickness of the proposal required to win a multimillion dollar contract is about one millimeter per million dollars. If all the proposals conforming to this standard were piled on top of each other at the bottom of the Grand Canyon, it would probably be a good idea.

-Norman Augustine (Former CEO, Lockheed Martin Corp.)

Product Models and Metrics

Function Points

One model of a product is to view it as a set of interfaces, e.g., files, data passed, etc.

If a system is primarily transaction processing and the “bulk” of the system deals with transformations on files, this is a reasonable view of size.

Function Points were originally suggested as a measure of size by Al Albrecht at IBM, a means of estimating functionality, size, effort

It can be applied in the early phases of a project (requirements, preliminary design)
A function point is a specific user functionality delivered by the application.

It differentiates five types of files or data:

- **Input type**, e.g., screen data, menu selection
- **Output Type**, e.g., report, transferred data, message
- **Query Type**, e.g., request/retrieval combination
- **File type**, e.g., database/record, indexed file
- **External interface**, e.g., reference data, external data bases

There are counting rules:

Only user requested and visible components are counted.

Components such as internally maintained data entries, externally maintained data entries, data maintenance activities, data output and data retrieval are categorized and valued.

The final count is adjusted based upon the general characteristics of the system (distributed functions, performance considerations, complex processing).

The original function point approach was proposed by Albrecht in the late 70s in the IBM Data Processing Division.

There is currently an International Function Point User Group (IFPUG) whose mission is to coordinate that the state of the practice, support users and standardize the approach.

Function Point Counting Practices Manual (Version 4)
Function Points Calculation

Complexity Weights

<table>
<thead>
<tr>
<th></th>
<th>SIMPLE</th>
<th>AVERAGE</th>
<th>COMPLEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Output type</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Query type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_Input part</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>—Output part</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>File type</td>
<td>7</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>External interface</td>
<td>5</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

Function Points Calculation

Application Characteristics

- DATA COMMUNICATIONS
- DISTRIBUTED DATA OR PROCESSING
- PERFORMANCE OBJECTIVES
- HEAVILY-USED CONFIGURATION
- TRANSACTION RATE
- ON-LINE DATA ENTRY
- END USER EFFICIENCY

INFLUENCE SCALE

0  NONE
1  INSIGNIFICANT, MINOR
2  MODERATE
3  AVERAGE
4  SIGNIFICANT
5  STRONG THROUGHOUT
Function Points Calculation

FUNCTION POINTS =

\[ \sum (\text{INPUTS} \times \text{WEIGHTS} + \text{OUTPUTS} \times \text{WEIGHTS}) + \sum \text{QUERIES} \times \text{WEIGHTS} + \sum \text{FILES} \times \text{WEIGHTS} + \sum \text{INTERFACES} \times \text{WEIGHTS} \times (0.65 + 1\% \text{ TOTAL INFLUENCE}) \]

Example

SPELLING CHECKER SPECIFICATION: The checker accepts as input a document file and an optional personal dictionary file. The checker lists all words not contained in either the dictionary or the personal dictionary files. The user can query the number of words processed and the number of spelling 'errors' found at any stage during the processing.

**Diagram:**

- USER
  - Document file
  - Personal dictionary
  - Misspelt words report
  - Errors inquiry/# errors
  - Words processed inquiry/# words
- DICTIONARY
  - List of words
- SPELLING CHECKER
### Function Points Calculation

**Example**

- **INPUTS:** DOCUMENT FILE NAME, PERSONAL DICTIONARY NAME
- **OUTPUT:** MISSPELT WORDS REPORT, # WORDS PROCESSED MESSAGE, # ERRORS MESSAGE
- **QUERIES:** ERRORS FOUND, WORDS PROCESSED
- **FILES:** DICTIONARY
- **INTERFACES:** DOCUMENT FILE, PERSONAL DICTIONARY

Assuming average complexity in each case and minor impact

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTS</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>OUTPUTS</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>QUERIES</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>FILES</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>INTERFACES</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

\[
65 \times (0.65 + 0.01) = 51.35 \text{ FUNCTION POINTS}
\]

### Function Points

**Estimating Costs**

- Cost estimation using function points requires
  - Cost data for previous projects
  - Function points counted in previous projects
- The estimation process:
  - The data about the cost of previous projects is plotted against the function points counted in those projects
  - This curve is used to derive the cost of the current project from the value of its function points

Example:

\[
\text{Effort (Days)} = 2.37 \times \text{(Function Points)} + 41.09
\]
Function Points
Assessing Productivity

- Productivity assessment using function points requires:
  - Productivity figures for previous projects
  - Function points counted in previous projects

- The assessment process:
  - The data about the productivity in previous projects is plotted against the function points counted in those projects
  - The expected productivity is the productivity value for the function points of this project
  - Discrepancies between actual and expected are analyzed

![Graph showing actual vs. previous projects] (FP for current project)

Function Points
Reliability of Function Point Based Measures

- Generally a productivity model is considered good if it is capable of giving an estimate with 25% accuracy in 75% of the cases

- Studies conducted in MIS (Management Information Systems) environments show that, for both development and maintenance, function points based measures satisfy the criterion

- Example: A recent study conducted in a Canadian financial institution on maintenance activities (21 projects, 332 average staff days per project, min 52, max 531)

<table>
<thead>
<tr>
<th>Deviation</th>
<th>Projects within range</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number</td>
</tr>
<tr>
<td>+ / - 10%</td>
<td>9</td>
</tr>
<tr>
<td>+ / - 20%</td>
<td>12</td>
</tr>
<tr>
<td>+ / - 26%</td>
<td>17</td>
</tr>
</tbody>
</table>
Software Science

Suppose we view a module or program as an encoding of an algorithm and seek some minimal coding of its functionality

The model would be an abstraction of the smallest number of operators and operands (variables) necessary to compute a similar function

And then the smallest number of bits necessary to encode those primitive operators and operands

This model was proposed by Maurice Halstead as a means of approximating program size in the early 1970s. It is based upon algorithmic complexity theory concepts previously developed by Chaitin and Kolmogorov in the 1960s, as extensions to Shannon’s work on information theory.

Algorithmic complexity

- The randomness (or algorithmic complexity) of a string is the minimal length of a program which can compute that string.
- Considered the 192 digits in the string 123456789101112...9899100.
  - Its complexity is less than 192 since it can be described by the 27 character Pascal statement for I:=1 to 100 do write(I).
  - But a random string of 192 digits would have no shorter encoding. The 192 characters in the string would be its own encoding.
  - Therefore the given 192 digits is less random and more structured than 192 random digits.
- If we increase the string to 5,888,896 digits 1234...9999991000000, then we only need to marginally increase the program complexity from 27 to 31 for I:=1 to 1000000 do write(I).
  - The 5,888,704 additional digits only add 4 characters of complexity to the string.
  - It is by no means that 31 is even minimal. It is assuming a Pascal interpreter. The goal of Chaitin’s research was to find the absolute minimal value for any sequence of data items.
- Halstead tried to apply these concepts to estimating program size.
Software Science

• MEASURABLE PROPERTIES OF ALGORITHMS

\[ n_1 = \# \text{Unique or distinct operators in an implementation} \]
\[ n_2 = \# \text{Unique or distinct operands in an implementation} \]
\[ N_1 = \# \text{Total usage of all operators} \]
\[ N_2 = \# \text{Total usage of all operands} \]
\[ f_{1,j} = \# \text{Occurrences of the } j^{th} \text{ most frequent operator} \quad j = 1, 2, \ldots, n_1 \]
\[ f_{2,j} = \# \text{Occurrences of the } j^{th} \text{ most frequent operand} \quad j = 1, 2, \ldots, n_2 \]

THE VOCABULARY \( n \) IS \( n = n_1 + n_2 \)

THE IMPLEMENTATION LENGTH IS \( N = N_1 + N_2 \)

\[
N_1 = \sum_{j=1}^{n_1} f_{1,j} \quad N_2 = \sum_{j=1}^{n_2} f_{2,j} \quad N = \sum_{j=1}^{n_1} \sum_{j=1}^{n_2} f_{j,j} \]

Example: Euclid’s Algorithm

IF (A = O)
LAST: BEGIN GCD := B; RETURN END;
IF (B = O)
BEGIN GCD := A; RETURN END;
HERE: G := A/B; R := A - B X G;
IF (R = O) GO TO LAST;
A := B; B := R; GO TO HERE
### Operator Parameters

**Greatest Common Divisor Algorithm**

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>( j )</th>
<th>( f_{1j} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>:=</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>( ) or BEGIN...END</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>IF</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>=</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>/</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>GO TO HERE</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>GO TO LAST</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ n_1 = 10 \quad N_1 = 31 \]

---

### Operand Parameters

**Greatest Common Divisor Algorithm**

<table>
<thead>
<tr>
<th>OPERAND</th>
<th>( j )</th>
<th>( f_{2j} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>O</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>R</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>GCD</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ n_2 = 6 \quad N_2 = 21 \]
**Software Science Metrics**

**PROGRAM LENGTH:**

\[ n \sim N = n_1 \log_2 n_1 + n_2 \log_2 n_2 \]

\(N = \) The number of bits necessary to represent all things that exist in the program at least once

\(N = \) The number of bits necessary to represent the symbol table

**PROGRAM VOLUME:** (Size of an implementation)

\[ V = N \log_2 n \]

\(B = \) The number of bits necessary to represent the program

**POTENTIAL VOLUME:**

\[ V = (2 + n_2) \log_2 (2 + n_2) \]

\(V = \) A measure of the specification for an algorithm

\(n_2 = \) represents the number of input/output parameters

**PROGRAM LEVEL:** (Level of an implementation)

\[ L = V / V \]

\[ L = 2n_2 \]

\(D = 1/L = \) Difficulty
Software Science Metrics

PROGRAMMING EFFORT:

\[ E = V D = V/L = V^2/V^* \]

\( E \) = The effort required to comprehend an implementation rather than produce it

\( E \) = A measure of program clarity

TIME:

\[ T = E/S = V/SL = V^2/SV^* \]

\( T \) = the time to develop an algorithm

ESTIMATED BUGS:

\[ \hat{B} = (LE)/E_o = V/E_o \]

WHERE \( E_o \) = The mean effort between potential errors in programming

\[ \hat{B} \] = the number of errors expected in a program

Cyclomatic Complexity

- The Cyclomatic Number \( V \) (\( G \)) of a graph \( G \) with \( n \) vertices, \( e \) edges, and \( p \) connected components is

\[ v(G) = e - n + p(2) \]

- In a strongly connected graph \( G \), the cyclomatic number is equal to the maximum number of linearly independent circuits

\[ G = \]

\[ \text{ENTRY NODE} \]

\[ \text{EXIT NODE} \]

- \( V(G) = 9 - 6 + 2 = 5 \) linearly independent circuits, e.g.,

\( (a \ b \ e \ f \ a), (b \ e \ b), (a \ b \ e \ a), (a \ c \ f \ a), (a \ d \ c \ f \ a) \)

- Suppose we view a program as a directed graph, an abstraction of its flow of control, and then measure the complexity by computing the number of linearly independent paths, \( v(G) \)
Properties of Cyclomatic Complexity

1) \( v(G) \geq 1 \)
2) \( v(G) = \# \) linearly independent paths in \( G \); it is the size of a basis set
3) Inserting or deleting functional statements to \( G \) does not affect \( v(G) \)
4) \( G \) has only one path iff \( v(G) = 1 \)
5) Inserting a new edge in \( G \) increases \( v(G) \) by 1
6) \( v(G) \) depends only on the decision structure of \( G \)
   - For more than 1 component
     \[
     v(M \cup A \cup B) = e - n + 2p = 13 - 13 + 2(3) = 6
     \]
   - For a collection of components
     \[
     v(C) = \sum_{C_i} v(C_i) = \sum_{k} C_i
     \]

Simplification

\( \Theta = \# \) Function nodes

\( \Pi = \# \) Predicate nodes

\( = \# \) Collecting nodes

\[ e = 1 + \Theta + 3\pi \]
\[ n = \Theta + 2\pi + 2 \]

Assuming \( p = 1 \) and \( v = e \cdot n + 2p \) yields
\[ v = (1 + \Theta + 3\pi) \cdot (\Theta + 2\pi + 2) + 2 = \pi + 1 \]

I.E., \( v(G) \) of a structured program equals the number of predicate nodes plus 1
Simplification

THE RESULT GENERALIZES TO NONSTRUCTURED PROGRAMS

\[ V(G) = \text{NUMBER OF DECISIONS} + 1 \]

The concept of cyclomatic complexity is tied to the complexity of testing program.
As a metric, it is easy to compute.
It has been well studied.

McCABE RECOMMENDS A MAXIMUM \( V(G) \) OF 10 FOR ANY MODULE.

SEL
Evaluating and Comparing Software Metrics

GOALS

DO MEASURES LIKE CYCLOMATIC COMPLEXITY AND THE SOFTWARE SCIENCE METRICS RELATE TO EFFORT AND QUALITY?

DOES THE CORRESPONDENCE INCREASE WITH GREATER ACCURACY OF DATA REPORTING?

HOW DO THESE METRICS COMPARE WITH TRADITIONAL SIZE METRICS SUCH AS NUMBER OF SOURCE LINES OR EXECUTABLE STATEMENTS?

HOW DO THESE METRICS RELATE TO ONE ANOTHER?

DEFINITIONS

EFFORT: THE NUMBER OF MAN-HOURS PROGRAMMERS AND MANAGERS SPENT FROM THE BEGINNING OF FUNCTIONAL DESIGN TO THE END OF ACCEPTANCE TESTING.

QUALITY: THE NUMBER OF PROGRAM FAULTS REPORTED DURING THE DEVELOPMENT OF THE PRODUCT.
**Metric Evaluation in the SEL**

**Size and Complexity Measures Investigated**

**THE DATA:**

**COMMERCIAL SOFTWARE:** *SATELLITE GROUND SUPPORT*

SYSTEMS CONSIST OF 51,000 TO 112,000 LINES OF FORTRAN SOURCE CODE

**TEN TO SIXTY-ONE PERCENT OF SOURCE CODE MODIFIED FROM PREVIOUS PROJECTS**

DEVELOPMENT EFFORT RANGES FROM 6900 TO 22,300 MAN-HOURS

**THIS ANALYSIS FOCUSES ON:**

- DATA FROM SEVEN PROJECTS
- ONLY NEWLY DEVELOPED MODULES (I.E., SUBROUTINES, FUNCTIONS, MAIN PROCEDURES AND BLOCK DATA’S)

---

**METRIC EVALUATION IN THE SEL**

**SIZE and COMPLEXITY MEASURES INVESTIGATED**

**OBJECTIVE SIZE AND COMPLEXITY MEASURES INVESTIGATED**

<table>
<thead>
<tr>
<th>Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Lines of Code</td>
<td></td>
</tr>
<tr>
<td>Source Lines of Code Excluding Comments</td>
<td></td>
</tr>
<tr>
<td>Executable Statements</td>
<td></td>
</tr>
</tbody>
</table>

**SOFTWARE SCIENCE METRICS**

- **N**: Length in Operators and Operands
- **V**: Volume
- **V***: Potential Volume
- **L**: Program Level
- **E**: Effort
- **B**: Bugs

**CYCLOMATIC COMPLEXITY**

- Cyclostatic Complexity Excluding Compound Decisions (Referred to as Cyclo_CMPLX_2)
- Number of Procedure and Function Calls
- Calls Plus Jumps
- Revisions (Versions) of the Source in the Program Library
- Number of Changes to the Source Code
### METRICS’ RELATION TO ACTUAL EFFORT

**Spearman Correlations (R Values - All Signif. at P = 0.001)**

<table>
<thead>
<tr>
<th>Metric</th>
<th>All Projects</th>
<th>Single Project</th>
<th>Single Programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validity Ratio</td>
<td>All</td>
<td>80%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>731</td>
<td>79</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>31</td>
</tr>
<tr>
<td># Modules</td>
<td>E^^</td>
<td>.49</td>
<td>.70</td>
</tr>
<tr>
<td></td>
<td>.75</td>
<td>.80</td>
<td>.79</td>
</tr>
<tr>
<td></td>
<td>CYCLO_CMPLX_2</td>
<td>.47</td>
<td>.76</td>
</tr>
<tr>
<td></td>
<td>.79</td>
<td>.79</td>
<td>.68</td>
</tr>
<tr>
<td>CALLS &amp; JUMPS</td>
<td>.49</td>
<td>.78</td>
<td>.81</td>
</tr>
<tr>
<td></td>
<td>.82</td>
<td>.70</td>
<td></td>
</tr>
<tr>
<td>CALLS &amp; JUMPS</td>
<td>.52</td>
<td>.69</td>
<td>.67</td>
</tr>
<tr>
<td></td>
<td>.73</td>
<td>.82</td>
<td>.79</td>
</tr>
<tr>
<td>SOURCE LINES</td>
<td>.46</td>
<td>.69</td>
<td>.71</td>
</tr>
<tr>
<td></td>
<td>.78</td>
<td>.78</td>
<td>.75</td>
</tr>
<tr>
<td>EXECUT. STMTS</td>
<td>.45</td>
<td>.68</td>
<td>.72</td>
</tr>
<tr>
<td></td>
<td>.80</td>
<td>.80</td>
<td>.68</td>
</tr>
<tr>
<td>REVISIONS</td>
<td>.53</td>
<td>.68</td>
<td>.72</td>
</tr>
<tr>
<td></td>
<td>.80</td>
<td>.80</td>
<td>.68</td>
</tr>
</tbody>
</table>

**Some Relation to Effort Across All Projects**

Relation improve with:

- Individual Projects
- Validated Data
- Individual Programmers

### METRICS’ RELATION TO PROGRAM FAULTS

The number of program faults for a given module is the number of system changes that listed the module as affected by an error correction.

**Weighted Faults (W_FLTS) is a measure of the amount of effort spent isolating and fixing faults in a module.**

**Spearman Correlation (R Values - All Signif. at P = 0.0001, Except (*) Signif. at P = 0.05)**

<table>
<thead>
<tr>
<th>Metric</th>
<th>All Projects</th>
<th>Single Project</th>
<th>Single Programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules</td>
<td>652</td>
<td>132</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Faults</td>
<td>W_FLTS</td>
<td>Faults</td>
</tr>
<tr>
<td>E^^</td>
<td>.16</td>
<td>.19</td>
<td>.58</td>
</tr>
<tr>
<td>CYCLO_CMPLX_2</td>
<td>.19</td>
<td>.20</td>
<td>.55</td>
</tr>
<tr>
<td>SOURCE LINES</td>
<td>.24</td>
<td>.25</td>
<td>.57</td>
</tr>
<tr>
<td>SOURCE LINES</td>
<td>.26</td>
<td>.27</td>
<td>.65</td>
</tr>
<tr>
<td>EXECUT. STMTS</td>
<td>.18</td>
<td>.20</td>
<td>.54</td>
</tr>
<tr>
<td>B</td>
<td>.17</td>
<td>.19</td>
<td>.54</td>
</tr>
<tr>
<td>REVISIONS</td>
<td>.38</td>
<td>.38</td>
<td>.78</td>
</tr>
<tr>
<td>EFFORT</td>
<td>.32</td>
<td>.33</td>
<td>.64</td>
</tr>
</tbody>
</table>

Relations low overall; # revisions strongest

Relations improve with individual projects or programmers.
RELATIONSHIP BETWEEN SIZE and COMPLEXITY METRICS

SPEARMAN R VALUES
(ALL SIGNIF. AT P = 0.001)

1794 MODULES

<table>
<thead>
<tr>
<th>SOURCE LINES (SLOC)</th>
<th>REVI-SIONS</th>
<th>CALLS &amp; JUMPS</th>
<th>CALLS</th>
<th>CYCLO-CMPLX</th>
<th>CYCLO-CMPLX_2</th>
<th>EXECUT STMTS</th>
<th>SLOC-CMNTS</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>E^2</td>
<td>.83</td>
<td>.37</td>
<td>.89</td>
<td>.62</td>
<td>.89</td>
<td>.88</td>
<td>.95</td>
<td>.86</td>
</tr>
<tr>
<td>V</td>
<td>.82</td>
<td>.35</td>
<td>.87</td>
<td>.57</td>
<td>.87</td>
<td>.87</td>
<td>.96</td>
<td>.86</td>
</tr>
<tr>
<td>SLOC-CMNTS</td>
<td>.93</td>
<td>.49</td>
<td>.88</td>
<td>.68</td>
<td>.86</td>
<td>.85</td>
<td>.91</td>
<td></td>
</tr>
<tr>
<td>EXECUT STMTS</td>
<td>.85</td>
<td>.38</td>
<td>.91</td>
<td>.61</td>
<td>.92</td>
<td>.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLO-CMPLX</td>
<td>.81</td>
<td>.39</td>
<td>.95</td>
<td>.55</td>
<td>.99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLO-CMPLX_2</td>
<td>.82</td>
<td>.38</td>
<td>.94</td>
<td>.56</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALLS</td>
<td>.66</td>
<td>.41</td>
<td>.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALLS &amp; JUMPS</td>
<td>.85</td>
<td>.44</td>
<td>.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REVI-SIONS</td>
<td>.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CONCLUSIONS

CAN USE COMMERCIALY OBTAINED DATA TO VALIDATE SOFTWARE METRICS

VALIDITY CHECKS AND ACCURACY RATINGS ARE USEFUL

NONE OF THE METRICS SEEM TO Satisfactorily EXPLAIN EFFORT OR DEVELOPMENT FAULTS

NEITHER SOFTWARE SCIENCE’S E METRIC, CYCLOMATIC COMPLEXITY NOR SOURCE LINES RELATES CONVINCINGLY BETTER WITH EFFORT THAN THE OTHERS

THE STRONGEST EFFORT CORRELATIONS ARE DERIVED WITH MODULES FROM INDIVIDUAL PROGRAMMERS OR CERTAIN VALIDATED PROJECTS

THE MAJORITY OF EFFORT CORRELATIONS INCREASE WITH THE MORE RELIABLE DATA

THE NUMBER OF REVISIONS CORRELATES WITH DEVELOPMENT FAULTS BETTER THAN EITHER SOFTWARE SCIENCE’S B METRIC, E METRIC, CYCLOMATIC COMPLEXITY OR SOURCE LINES OF CODE

SOME OF THE SIZE AND COMPLEXITY MEASURES RELATE WELL WITH EACH OTHER
DATA BINDINGS

- A SEGMENT - GLOBAL - SEGMENT DATA BINDING \((p, r, q)\) IS AN OCCURRENCE OF THE FOLLOWING:

  1. SEGMENT \(p\) MODIFIES GLOBAL VARIABLE \(r\)
  2. VARIABLE \(r\) IS ACCESSED BY SEGMENT \(q\)
  3. \(p \neq q\)

- EXISTENCE OF A DATA BINDING \((p, r, q)\) \(\iff\) \(q\) DEPENDENT ON THE PERFORMANCE OF \(p\) BECAUSE OF \(r\)

- BINDING \((p, r, q)\) \(\neq\) BINDING \((q, r, p)\)

- \((p, r, q)\) REPRESENTS A UNIQUE COMMUNICATION PATH BETWEEN \(p\) AND \(q\)

- THE TOTAL \# DATA BINDINGS REPRESENTS THE DEGREE OF A CERTAIN KIND "CONNECTIVITY" (I. E., BETWEEN SEGMENT PAIRS VIA GLOBALS) WITHIN A COMPLETE PROGRAM

BASILI/TURNER

INT A, B, C, D

```
PROC P1
  /* USES A, B */
  . . .
PROP P2
  /* USES A, B */
  . . .
  CALL P3 (X)
  . . .
PROC P3 (INT E)
  /* USES C, D */
  . . .
PROC P4
  /* USES C, D */
  . . .
```

DATA BINDINGS

\((P1, A, P2)\)
\((P1, B, P2)\)
\((P3, C, P4)\)
\((P3, D, P4)\)
\((P2, E, P3)\)
**LEVELS OF DATA BINDING (DB)**

- **Potential DB** is an ordered triple \((P, X, Q)\) for components \(P\) and \(Q\) and variable \(X\) in the scope of \(P\) and \(Q\).

- **Usage DB** is a potential DB such that \(P\) and \(Q\) both use \(X\) for reference or assignment.

- **Feasible DB** (actual) is a usage DB such that \(P\) assigns to \(X\) and \(Q\) references \(X\).

- **Control Flow DB** is a feasible DB such that flow analysis allows the possibility of \(Q\) being executed after \(P\) has started.

---

**SEGMENT GLOBAL USAGE PAIRS**

- A segment-global usage pair \((p, r)\) is an instance of a global variable \(r\) being used by a segment \(p\) (i.e., \(r\) is either modified or set by \(p\)).

- Each usage pair represents a unique "use connection" between a global and a segment.

- Let actual usage pair (AUP) represent the count of true usage pairs (i.e., \(r\) is actually used by \(p\)).

- Let possible usage pair (PUP) represent the count of potential usage pairs (i.e., given the program's globals and their scopes, the scope of \(r\) contains \(p\) so that \(p\) could potentially modify \(r\)) (worst case).

- Then the relative percentage usage pairs (RUP) is

\[
RUP = \frac{AUP}{PUP}
\]

And is a way of normalizing the number of usage pairs relative to the problem structure.

- The RUP metric is an empirical estimate of the likelihood that an arbitrary segment uses an arbitrary global.
METRICS ACROSS TIME
Measurement Across Time

Measures are sometimes difficult to understand in the absolute

However the relative changes in metrics over the evolution of the system can be very informative

This evolution may be within one development cycle of the product

  e.g., Requirements --> Design --> Code --> ...

Or

Multiple versions of the same product

  e.g., Code_1 --> Code_2 --> Code_3 --> ...

METRICS ACROSS TIME
Development/Maintenance Vector

A vector of metrics, m_1, m_2, m_n, can be defined dealing with various aspects of the product, i.e., effort, changes, defects, logical, physical, and dynamic attributes, environmental considerations, ...

For example, some physical attributes might include (decisions, interaction of data, interaction of data, size) across modules within a module

The vector characterizes the product at some point in time

We can view it at various stages of product evolution to monitor how the product is changing

We can provide a set of bounds for the metrics to signal potential problems and anomalies
Various metrics were used during different points in the development of a software product

The product was a compiler for a structured programming language
– about 6,500 high level source statements
– about 17,000 lines of source code

We will examine the changes of the values of various metrics across time
– to provide insight into how the product was progressing
– to allow us to evaluate the quality of the product

There were 17 enhancements of the product for this study
– we will look at 5 major iterations
– there were iterations after the last one

### METRICS ACROSS TIME

#### Case Study

Statistics from Compilers at 5 Selected Points in the Iterative Process

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER OF STATEMENTS</td>
<td>3404</td>
<td>4217</td>
<td>5181</td>
<td>5847</td>
<td>6350</td>
</tr>
<tr>
<td>NUMBER OF PROCEDURES AND FUNCTIONS</td>
<td>89</td>
<td>189</td>
<td>213</td>
<td>240</td>
<td>289</td>
</tr>
<tr>
<td>NUMBER OF SEPARATE COMPILED MODULES</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>37</td>
</tr>
<tr>
<td>AVERAGE NESTING LEVEL</td>
<td>3.4</td>
<td>2.9</td>
<td>2.9</td>
<td>2.9</td>
<td>2.8</td>
</tr>
<tr>
<td>AVERAGE NUMBER OF TOKENS PER STATEMENT</td>
<td>5.7</td>
<td>6.3</td>
<td>6.6</td>
<td>7.2</td>
<td>7.3</td>
</tr>
</tbody>
</table>
### METRICS ACROSS TIME

Statistics from Compilers at 5 Selected Points in the Iterative Process

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>USAGE COUNT OF (SEGMENT, GLOBAL) PAIRS</td>
<td>611</td>
<td>786</td>
<td>941</td>
<td>1030</td>
<td>974</td>
</tr>
<tr>
<td>TOTAL POSSIBLE COUNT OF (SEGMENT, GLOBAL) PAIRS</td>
<td>4128</td>
<td>8707</td>
<td>10975</td>
<td>6930</td>
<td>4584</td>
</tr>
<tr>
<td>PERCENTAGE USE OF GLOBALS</td>
<td>14.8</td>
<td>9.0</td>
<td>8.6</td>
<td>14.9</td>
<td>21.2</td>
</tr>
<tr>
<td>NUMBER OF ACTUAL DATA BINDINGS</td>
<td>2610</td>
<td>6662</td>
<td>8759</td>
<td>12006</td>
<td>10442</td>
</tr>
<tr>
<td>NUMBER OF POSSIBLE DATA BINDINGS</td>
<td>243780</td>
<td>814950</td>
<td>1337692</td>
<td>497339</td>
<td>342727</td>
</tr>
<tr>
<td>PERCENTAGE OF POSSIBLE BINDINGS THAT OCCURED</td>
<td>1.1</td>
<td>0.8</td>
<td>0.7</td>
<td>2.4</td>
<td>3.0</td>
</tr>
</tbody>
</table>

### SPAN

- A SPAN is the number of statements between two textual references to the same identifier

\[
\begin{align*}
X &: = Y ; \\
Z &: = Y ; \\
X &: = Y ;
\end{align*}
\]

- SPAN (X) = count of # statements between first and last statements (assuming no intervening references to X) Y has two spans
- For n appearances of an identifier in the source text, n - 1 spans are measured
- All appearances are counted except those in declare statements
- If SPAN > 100 statements, then one new item of information must be remembered for 100 statements till read again
**SPAN**

- COMPLEXITY \( \sim \) number of SPANS at any point (take max, average, median)
- OR \( \sim \) number of Statements a variable must be remember (on the average) [average span]

**VARIATION**

- Do a live/dead variable analysis
- Complexity proportional to number of variables alive at any statement
- How does one scale up this measure?

\[
C(M) = \sum_{j=1}^{\# \text{stmts}} \frac{n_j \cdot s(n_j)}{\# \text{stmts}}
\]

Where \( n_j \) = number of spans of size \( S(n_j) \)

**VARIABLE SPAN**

Variable span has been shown to be a reasonable measure of complexity

For commercial PL/1 programs, one study showed that a programmer must remember approximately 16 items of information when reading a program

Elshoff-GM
Product Models and Metrics

Dynamic Characteristics

We can divide the dynamic product the characteristics into two basic classes

We can view them as checking on the
  Behavior of the input to the code, e.g., coverage metrics
  Behavior of the code itself, e.g., reliability metrics

PRODUCT METRICS

Coverage Metrics

Based upon checking what aspects of the product are affected by a set of inputs

For example,

  procedure coverage - which procedures are covered by the set of inputs
  statement coverage - which statements are covered by the set of inputs
  branch coverage - which parts of a decision node are covered by the set of inputs
  path coverage - which paths are covered by the set of inputs
  requirements section coverage - which parts if the requirements documents have been read

Used to,
  check the quality of a test suite
  support the generation of new test cases
**TEST METRICS**

<table>
<thead>
<tr>
<th></th>
<th>PASCAL</th>
<th>FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># TEST CASES:</strong></td>
<td><strong>32</strong></td>
<td><strong>68</strong></td>
</tr>
<tr>
<td>SUBPROGRAM COVERAGE</td>
<td>.81</td>
<td>.91</td>
</tr>
<tr>
<td>BRANCH PATH COVERAGE</td>
<td>.59</td>
<td>1.00</td>
</tr>
<tr>
<td>I/O COVERAGE</td>
<td>.23</td>
<td>.35</td>
</tr>
<tr>
<td>DO LOOP ENTRY</td>
<td>.92</td>
<td>.74</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>.85</td>
<td>.48</td>
</tr>
<tr>
<td>OTHER EXECUTABLE</td>
<td>.74</td>
<td>.66</td>
</tr>
<tr>
<td>CODE COVERAGE</td>
<td>.70</td>
<td>.80</td>
</tr>
</tbody>
</table>

**ANOTHER PROGRAM**

**RELIABILITY**

*How Can We Use Reliability Metrics*

**SYSTEM ENGINEERING**

- Determine the best trade-off between reliability and cost, schedule, etc.
- Optimize life cycle cost
- Specify reliability to the designer

**PROJECT MANAGEMENT**

- Progress monitoring
- Scheduling
- Investigation of alternatives

**OPERATIONAL SOFTWARE MANAGEMENT**

**EVALUATION OF SOFTWARE ENGINEERING MANAGEMENT**

Musa / Goel /Littlewood
RELIABILITY
Software Reliability Models

TIME DEPENDENT APPROACHES
TIME BETWEEN FAILURES (Musa Model)
FAILURE COUNTS IN SPECIFIED INTERVALS (Goel/Okumoto)

TIME-INDEPENDENT APPROACHES
ERROR SEEDING
INPUT DOMAIN ANALYSIS

PROBLEMS WITH USE OF RELIABILITY MODELS
LACK OF CLEAR UNDERSTANDING OF INHERENT STRENGTHS AND WEAKNESSES
UNDERLYING ASSUMPTIONS AND OUTPUTS NOT FULLY UNDERSTOOD BY USER
NOT ALL MODELS APPLICABLE TO ALL TESTING ENVIRONMENTS

RELIABILITY MODELS
Musa

ASSUMPTIONS:

1. ERRORS ARE DISTRIBUTED RANDOMLY THROUGH THE PROGRAM
2. TESTING IS DONE WITH REPEATED RANDOM SELECTION FROM THE ENTIRE RANGE OF INPUT DATA
3. THE ERROR DISCOVERY RATE IS PROPORTIONAL TO THE NUMBER OF ERRORS IN THE PROGRAM
4. ALL FAILURES ARE TRACED TO THE ERRORS CAUSING THEM AND CORRECTED BEFORE TESTING RESUMES
5. NO NEW ERRORS ARE INTRODUCED DURING DEBUGGING

\[ T = \frac{1}{K} \frac{K t}{E} \]

WHERE:
- \( E \) IS TOTAL ERRORS IN THE SYSTEM
- \( t \) IS THE ACCUMULATED RUN TIME (STARTS @ 0)
- \( T \) IS THE MEAN TIME TO FAILURE

Musa, Iannino, Okumoto - Software Reliability
RELIABILITY
Failures Experienced vs Cumulative Execution Time

\[ m = M_0 \left[ 1 - \exp \left( -\frac{C_t}{M_0 T_o} \right) \right] \]

CUMULATIVE EXECUTION TIME \( t \)

RELIABILITY
Present MTTF vs Cumulative Execution Time

\[ T = T_0 e^{C_t / M_0 T_o} \]

CUMULATIVE EXECUTION TIME \( t \)
Software Reliability Estimation Execution Time Model

RELIABILITY

Combination of Approaches

CLEAN ROOM

DEVELOPER USES READING TECHNIQUES, TOP DOWN DEVELOPMENT
TESTING DONE BY INDEPENDENT ORGANIZATION AT INCREMENTAL STEPS
RELIABILITY MODEL USED TO PROVIDE DEVELOPER WITH QUALITY ASSESSMENT

FUNCTIONAL TESTING/COVERAGE METRICS

USE FUNCTIONAL TESTING APPROACH
COLLECT ERROR DISTRIBUTIONS, E.G., OMISSION vs COMMISSION
OBTAIN COVERAGE METRICS
KNOWING NUMBER OF ERRORS OF OMISSION, EXTRAPOLATE

ERROR ANALYSIS AND RELIABILITY MODELS

ESTABLISH ERROR HISTORY FROM PREVIOUS PROJECTS
DISTINGUISH SIMILARITIES AND DIFFERENCES TO CURRENT PROJECT
DETERMINE PRIOR ERROR DISTRIBUTIONS FOR THE CURRENT PROJECT
SELECT A CLASS OF STOCHASTIC MODELS FOR THE CURRENT PROJECT
UPDATE PRIOR DISTRIBUTIONS AND COMPARE ACTUAL DATA WITH THE PRIORS FOR THE CURRENT PROJECT
AUTOMATABLE CHANGE AND ERROR METRICS

AUTOMATABLE METRIC

No interference to the developer
Computation algorithmically from quantifiable sources
Reproducible on other projects with the same algorithms

USEFUL METRIC

Sensitive to externally observable differences in the development environment
Relative values correspond to some intuitive notion about characteristic differences in the environment

EXAMPLES

Program changes
Textual revision in the source code representing one conceptual change

Job steps
The number of computer accesses during development or maintenance

PROGRAM CHANGES

Textual revisions in the source code of a module during the development period

One program change should represent one conceptual change to the program

A program change is defined as:

One or more changes to a single statement
One or more statements inserted between existing statements
A change to a single statement followed by the insertion of new statements

The following are not counted as program changes:

The deletion of one or more existing statements
The insertion of standard output statements or special compiler-provided debugging directives
The insertion of blank lines or comments, the revision of comments and reformatting without alteration of existing statements

Program changes have been shown to correlate well with faults

GANNON / DUNSMORE
JOB STEPS

THE NUMBER OF COMPUTER ACCESSES

A SINGLE PROGRAMMER-ORIENTED ACTIVITY PERFORMED
ON THE COMPUTER AT THE OPERATING SYSTEM
COMMAND LEVEL

BASIC TO THE DEVELOPMENT EFFORT AND INVOLVES NON-
TRIVIAL EXPENDITURES OF COMPUTER OR HUMAN
RESOURCES

EXAMPLES: TEXT EDITING, MODULE COMPILATION,
PROGRAM COMPILATION, LINK EDITING, PROGRAM
EXECUTION

PRODUCT METRICS

Coverage Metrics

Based upon checking what aspects of the product are effected by a set of inputs

For example,
procedure coverage - which procedures are covered by the set of inputs
statement coverage - which statements are covered by the set of inputs
branch coverage - which parts of a decision node are covered by the set of inputs
path coverage - which paths are covered by the set of inputs
requirements section coverage - which parts of the requirements
document have been read

Used to
check the quality of a test suite
support the generation of new test cases
PRODUCT METRICS

Coverage Metrics

Based upon checking what aspects of the product are effected by a set of inputs

For example,

- procedure coverage - which procedures are covered by the set of inputs
- statement coverage - which statements are covered by the set of inputs
- branch coverage - which parts of a decision node are covered by the set of inputs
- path coverage - which paths are covered by the set of inputs
- requirements section coverage - which parts of the requirements document have been read

Used to

- check the quality of a test suite
- support the generation of new test cases

Additional Augustine Laws

Related to Metrics

- One tenth of the participants produce over one third of the output. Increasing the number of participants merely reduces the average output. (A variation on Brook’s Law-Adding people to speed up a late project just makes it later. )
- The last 10% of performance generates one third of the cost and two thirds of the problems.
- (Law of Unmitigated Optimism) Any task can be completed in only one-third more time than is currently estimated.
- (Law of Inconstancy of Time) A revised schedule is to business what a new season is to an athlete or a new canvas to an artist.
- Law of Propagation of Misery) If a sufficient number of management layers are superimposed on top of each other, it can be assured that disaster is not left to chance.
- VP of Boeing on 767 project: "is further ahead at the halfway point than any new airliner program in Boeing history."