

Product Models and Metrics

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

Product Models and Metrics

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 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, as a means of estimating functionality, size, effort

It can be applied in the early phases of a project (requirements, preliminary design)

Albrecht

Product Models and Metrics

Function Points

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

Product Models and Metrics

Function Points

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

	SIMPLE	AVERAGE	COMPLEX
Input	3	4	6
Output type	4	5	7
Query type			
-Input part	3	4	6
-Output part	4	5	7
File type	7	10	15
External interface	5	7	10

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

ON-LINE UPDATE
COMPLEX PROCESSING
REUSABILITY
CONVERSION & INSTALLATION EASE
OPERATIONAL EASE
MULTIPLE-SITE
FACILITATE CHANGE

INFLUENCE SCALE



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

Function Points Calculation

FUNCTION POINTS =

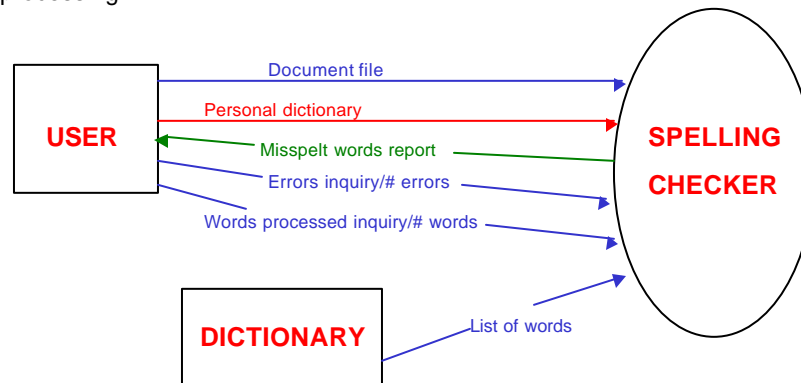
$(\Sigma \text{ INPUTS} * \text{WEIGHTS} + _ \text{ OUTPUTS} * \text{WEIGHTS}$

$\Sigma \text{ QUERIES} * \text{WEIGHTS} + _ \text{ FILES} * \text{WEIGHTS} +$

$\text{S INTERFACES} * \text{WEIGHTS}) * (0.65 + 1\% \text{ TOTAL INFLUENCE})$

Function Points Calculation 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.



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

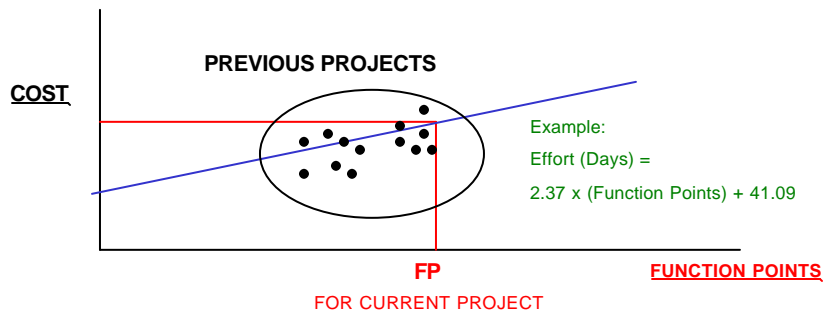
ASSUMING AVERAGE COMPLEXITY IN EACH CASE
AND MINOR IMPACT

INPUTS	2	4	8
OUTPUTS	3	5	15
QUERIES	2	9	18
FILES	1	10	10
INTERFACES	2	7	14

$$65 * (0.65 + 0.01 * 14) = 51.35 \text{ FUNCTION POINTS}$$

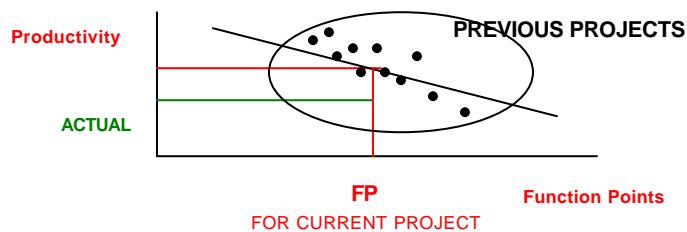
FUNCTION POINTS Estimating Costs

- Cost Estimation using Function Points requires
 - Cost or effort data for previous projects
 - Function Points counted in previous projects
- The estimation process
 - The cost (or effort) data 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



FUNCTION POINTS Assessing Productivity

- Productivity assessment using Function Points requires
 - productivity figures for previous projects
 - Function Points counted in previous projects
 -
- The assessment process:
 - Data about the productivity in previous projects is plotted against the FP count in those projects
 - The expected productivity is the productivity value for the FPs of this project
 - Discrepancies between actuals and expected are analyzed



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 often satisfy the criterion
- Example: Canadian financial institution study on maintenance activities (21 projects, 332 average staff days per project, min 52, max 532)

DEVIATION	PROJECTS WITHIN RANGE	
	NUMBER	%
+ / - 10%	9	43%
+ / - 20%	12	57%
+ / - 26%	17	81%

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.

Software Science

- MEASURABLE PROPERTIES OF ALGORITHMS

n_1 = # Unique or distinct operators in an implementation

n_2 = # Unique or distinct operands in an implementation

N_1 = # Total usage of all operators

N_2 = # Total usage of all operands

$f_{1,j}$ = # Occurrences of the j^{th} most frequent operator
 $j = 1, 2, \dots, n_1$

$f_{2,j}$ = # Occurrences of the j^{th} most frequent operand
 $j = 1, 2, \dots, n_2$

THE VOCABULARY n IS

$$n = n_1 + n_2$$

THE IMPLEMENTATION LENGTH IS

$$N = N_1 + N_2$$

and
$$N_1 = \sum_{j=1}^{n_1} f_{1,j} \quad N_2 = \sum_{j=1}^{n_2} f_{2,j} \quad N = \sum_{i=1}^2 \sum_{j=1}^{n_i} f_{ij}$$

Example: Euclid's Algorithm

LAST: IF (A = 0)
BEGIN GCD := B; RETURN END;
IF (B = 0)
BEGIN GCD := A; RETURN END;

HERE: G := A/B; R := A - B X G;
IF (R = 0) GO TO LAST;
A := B; B := R; GO TO HERE

Operator Parameters
Greatest Common Divisor Algorithm

OPERATOR	j	f _{1j}
;	1	9
:=	2	6
() or BEGIN...END	3	5
IF	4	3
=	5	3
/	6	1
-	7	1
x	8	1
GO TO HERE	9	1
GO TO LAST	10	1
	n ₁ = 10	N ₁ = 31

Operand Parameters
Greatest Common Divisor Algorithm

OPERAND	j	f _{2j}
B	1	6
A	2	5
O	3	3
R	4	3
G	5	2
GCD	6	2
	n ₂ = 6	N ₂ = 21

Software Science Metrics

PROGRAM LENGTH:

$$N \sim \hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

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

\hat{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: (Minimal size of an implementation)

$$V^* = (2 + n^*_2) \log_2 (2 + n^*_2)$$

Where n^*_2 represents the number of input/output parameters

V^* = A measure of the specification for an algorithm

Software Science Metrics

PROGRAM LEVEL: (Level of an implementation)

$$L = V^* / V$$

$$D = 1/L = \text{Difficulty}$$

PROGRAMMING EFFORT:

$$E = V D = V/L = V^2/V$$

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

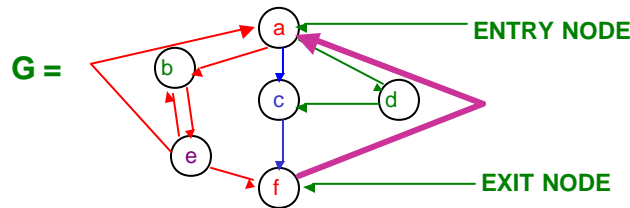
E = A measure of program clarity

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



- $V(G) = 9 - 6 + 2 = 5$ 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)

McCabe

CYCLOMATIC COMPLEXITY

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

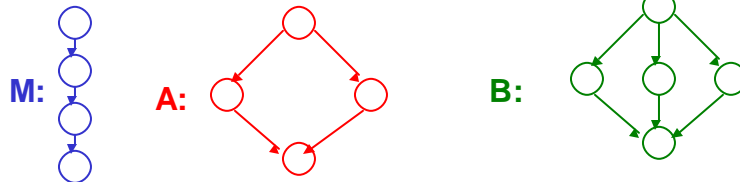
CYCLOMATIC COMPLEXITY

- For a collection of components
 - The cyclomatic number of the collection is the sum of the cyclomatic numbers of the individual components

$$v(C) = \sum_i v(C_i), \text{ where } C = \cup C_i$$

- In the example: For more than 1 component

$$v(M \cup A \cup B) = e - n + 2p = 13 - 13 + 2(3) = 6$$



CYCLOMATIC COMPLEXITY

It can be shown that

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

The concept of cyclomatic complexity
tied to complexity of testing program
easy to compute
has been well studied

Cyclomatic complexity is used mostly on modules

It has been recommended that the cyclomatic complexity of a module
be kept small, e.g. less than 10

There is some evidence to support this

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?

How do these metrics compare with traditional size metrics such as the number of source lines of code or the number of executable statements?

How do these metrics relate to one another?

DEFINITIONS

EFFORT: The number of staff hours programmers and managers spend 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 software

Systems consist of 51K to 112K lines of FORTRAN source code

Ten to sixty one percent of the source code was modified from previous projects

Development effort ranges from 7K to 22K staff hours

This analysis focuses on:

Data from 7 projects

only newly developed modules (i.e., subroutines, functions, main procedures and block data)

Metric Evaluation in the SEL Size and Complexity Measures Investigated

Objective Size And Complexity Measures Investigated

Source Lines Of Code

Source Lines Of Code Excluding Comments

Executable Statements

Software Science Metrics

N : Length In Operators And Operands

V : Volume

V* : Potential Volume

L : Program Level

E : Effort

B : Bugs

Cyclomatic Complexity

Cyclomatic 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

Metric Evaluation in the SEL Size and Complexity Measures Investigated

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

	All Projects	Single Project		Single Programmer	
	All	All	80%	90%	92.5%
Validity Ratio					
# Modules	731	79	29	20	31
E^^	.49	.70	.75	.80	.79
CYCLO_CMLPX_2	.47	.76	.79	.79	.68
CALLS & JUMPS	.49	.78	.81	.82	.70
SOURCE LINES	.52	.69	.67	.73	.86
EXECUT. STMTS	.46	.69	.71	.78	.75
V	.45	.68	.72	.80	.68
REVISIONS	.53	.68	.72	.80	.68

Some Relation To Effort Across All Projects

Relation Improve With:

Individual Projects

Validated Data

Individual Programmers

Metric Evaluation in the SEL Size and Complexity Measures Investigated

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)

MODULES	ALL PROJECTS 652		SINGLE PROJECT 132		SINGLE PROGRAMMER 21	
	FAULTS	W_FLTS	FAULTS	W_FLTS	FAULTS	W_FLTS
	E^^	.16	.19	.58	.52	.67
CYCLO_CMPLX_2	.19	.20	.55	.49	.48*	.45*
CALLS & JUMPS	.24	.25	.57	.52	.60*	.56*
SOURCE LINES	.26	.27	.65	.62	.66	.65
EXECUT. STMTS	.18	.20	.54	.51	.58*	.53*
B	.17	.19	.54	.50	.68	.66
REVISIONS	.38	.38	.78	.69	.83	.81
EFFORT	.32	.33	.64	.62	.67	.62

Relations Low Overall; # Revisions Strongest
Relations Improve With Individual Projects Or Programmers

Metric Evaluation in the SEL Size and Complexity Measures Investigated

SPEARMAN R VALUES

(ALL SIGNIF. AT P = 0.001)

1794 MODULES

	SOURCE								
	LINES (SLOC)	REVISIONS	CALLS & JUMPS	CALLS	CYCLO-CMPLX	CYCLO-CMPLX	EXECUT STMTS	SLOC-CMMTS	V
E^^	.83	.37	.89	.62	.89	.88	.95	.86	.98
V	.82	.35	.87	.57	.87	.87	.96	.86	
SLOC-CMMTS	.93	.49	.88	.68	.86	.85	.91		
EXECUT STMTS	.85	.38	.91	.61	.92	.91			
CYCLO-CMPLX	.81	.39	.95	.55	.99				
CYCLO-CMPLX_2	.82	.38	.94	.56					
CALLS	.66	.41	.75						
CALLS & JUMPS	.85	.44							
REVISIONS	.50								

Metric Evaluation in the SEL Size and Complexity Measures Investigated

Conclusions

Used commercially obtained data to validate software metrics
Common environment
Was able to perform validity checks and accuracy ratings

No metric
seemed to satisfactorily explain effort or development faults
related convincingly better with effort than the others

The strongest effort correlations
come from individual programmers or certain validated projects
increase with the more reliable data

The number of revisions correlates with development faults better than
any other metric

Many size and complexity measures relate well with each other

Product Models and Metrics

Data Structure Metrics

Data structure metrics measure the data interaction of the product

Major Concepts:

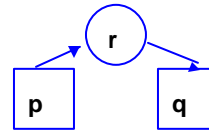
Coupling: refers to the degree of **interdependence between** parts of a design, usually the amount of data shared by parts

Cohesion: refers to the degree of **dependence within** parts of the design, usually the strength of the interaction

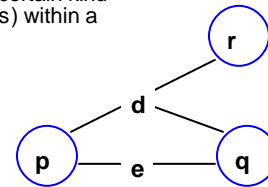
The definitions of these concepts depends on the design paradigm or notation used to expressed the design. It is often programming language dependent

Data Structure Metrics: Data bindings

- A SEGMENT - GLOBAL - SEGMENT DATA BINDING (p, r, q) is an occurrence of the following:
 - segment p modifies global variable r
 - variable r is accessed by segment q
 - $p \neq q$



- Existence of a data binding (p, q, r) \Rightarrow q dependent on the performance of p because of r
- DB (p, r, q) \neq DB (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

Data Structure Metrics: Data bindings

INT A, B, C, D

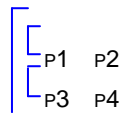
```

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)



Data Structure Metrics: Data bindings

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

Basili/Hutchens

Data Structure Metrics

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 $RUP = AUP/PUP$ 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

Measurement Across Time

Measures are sometimes difficult to understand in the absolute

However the relative changes in metrics over the 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., **Code₁** → **Code₂** → **Code₃** → ...

Measurement 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

Measurement 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

Measurement Across Time Case Study

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

Measurement Across Time Case Study

Statistics from Compilers at 5 Selected Points in the Iterative Process

	1	2	3	4	5
• Number of statements	3404	4217	5181	5847	6350
• Number of procedures and functions	89	189	213	240	289
• Number of separate Compiled modules	4	4	7	15	37
• Average nesting level	3.4	2.9	2.9	2.9	2.8
• Average number of Tokens per statement	5.7	6.3	6.6	7.2	7.3

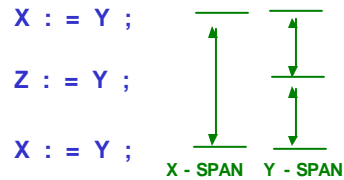
Measurement Across Time Case Study

Statistics from Compilers at 5 Selected Points in the Iterative Process

	1	2	3	4	5
Usage count of (Segment, Global) pairs (AUP)	611	786	941	1030	974
Total possible count of (Segment, Global) pairs	4128	8707	10975	6930	4584
Percentage use of globals (PUP)	14.8	9.0	8.6	14.9	21.2
Total token size	19403	26567	34194	42098	46355
Number of Data Bindings	2610	6662	8759	12006	10442
Number of Data Bindings per Thousand tokens	13.4	17.7	25.6	28.5	22.5

Psychological Complexity: SPAN

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



- 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

Elishoff-GM

Psychological Complexity: SPAN

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

VARIATION

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

$$C(M) = \frac{\sum_{j=i}^S n_i \cdot S(n_i)}{\# \text{ stmts}}$$

Where $n_i = \#$ spans of size S (n_i)

Psychological Complexity: 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 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

TEST METRICS

PASCAL

# TEST CASES:	<u>32</u>	<u>36</u>
SUBPROGRAM COVERAGE	.81	.92
BRANCH PATH COVERAGE	<u>.59</u>	.67
I/O COVERAGE	<u>.23</u>	.54
DO LOOP ENTRY	.92	.94
ASSIGNMENT	.85	.91
OTHER EXECUTABLE	.74	.78
CODE COVERAGE	<u>.70</u>	.80

FORTRAN

# TEST CASES:	<u>68</u>] — ANOTHER PROGRAM
SUBROUTINE COVERAGE	.91	
FUNCTION COVERAGE	1.00	
BRANCH PATH	.63	
I/O	.35	
DO-LOOP	.74	
ASSIGNMENT	.48	
OTHER EXECUTABLE	.66	

Stucki - Boeing

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

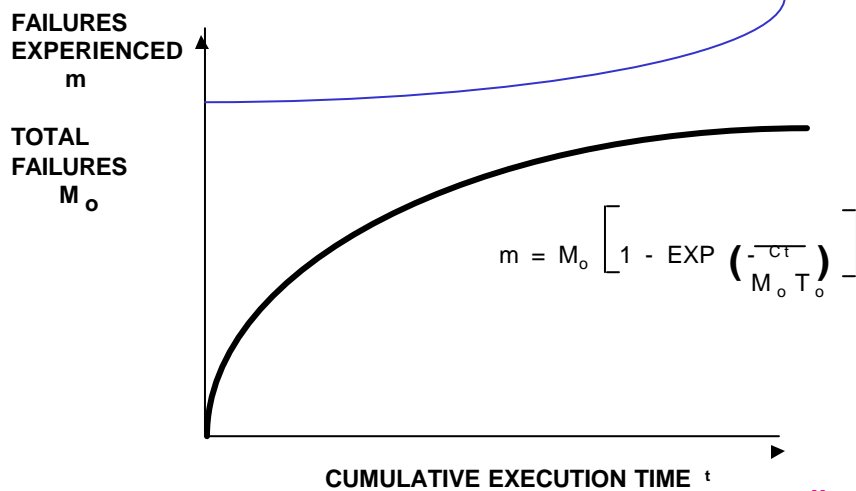
$$T = \frac{1}{KE} e^{Kt}$$

WHERE **E** is total errors in the system
t is the accumulated run time (starts @ 0)
T is the mean time to failure

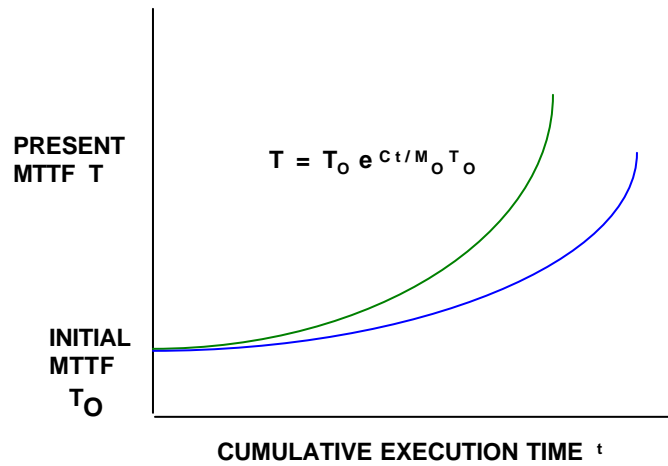
Musa, Iannino, Okumoto - Software Reliability

RELIABILITY

Failures Experienced vs Cumulative Execution Time

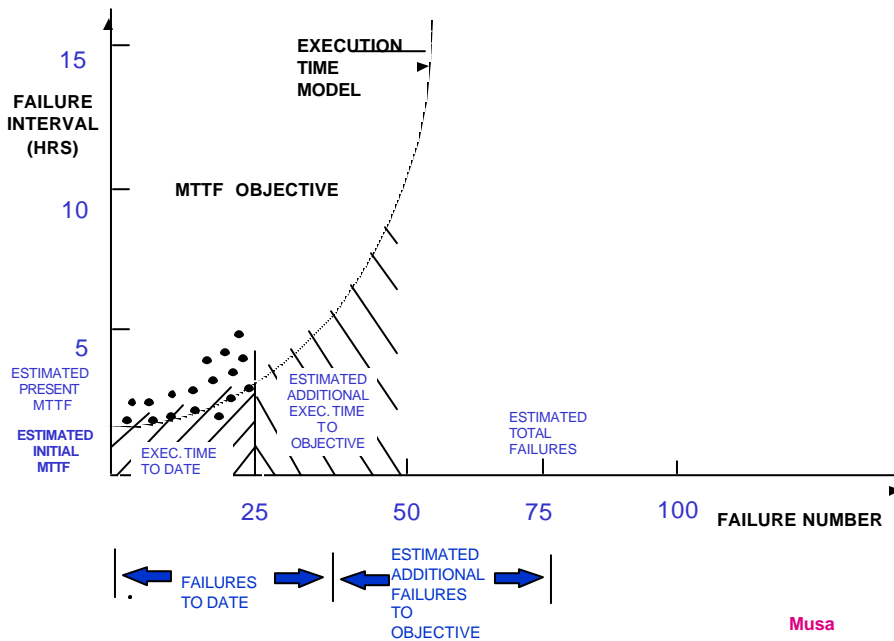


RELIABILITY Present MTTF vs Cumulative Execution Time



Musa

RELIABILITY Software Reliability Estimation Execution Time Model



Musa

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
- Computed 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

Automatable Change And Error Metrics

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

Automatable Change And Error Metrics

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

Basili / Reiter

Product Models and Metrics References

- Victor R. Basili and Albert J. Turner, Iterative Enhancement: A Practical Technique for Software Development, IEEE Transactions on Software Engineering, vol. 1, #4, December 1975 (update appears as Portfolio 14-01-05, Auerbach Information Management Series 1978).
- Victor R. Basili and Robert Reiter, Jr., An Investigation of Human Factors in Software Development, IEEE Computer Magazine, pp 21-38, December 1979.
- Victor R. Basili & Robert Reiter, Jr., A Controlled Experiment Quantitatively Comparing Software Development Approaches, IEEE Transactions on Software Engineering, pp 299-320 (IEEE Computer Society Outstanding Paper Award), May 1981.)
- Victor R. Basili and David Hutchens, An Empirical Study of a Syntactic Complexity Family, IEEE Transactions on Software Eng. , vol. SE-9, #6, pp 664-672, November 1983.
- Victor R. Basili, Richard Selby and Tsai-Yun Phillips, Metric Analysis and Data Validation Across FORTRAN Projects, IEEE Transactions on Software Engineering, vol. SE-9, #6, pp 652-663, November 1983.
- David H. Hutchens and Victor R. Basili, System Structure Analysis: Clustering with Data Bindings, IEEE Transactions on Software Engineering, pp 749-757, August 1985.
- Maurice Halstead, Elements of Software Science, North Holland, 1979.