Software Measurement, Complexity, & Defect Prediction

Dr. Mark C. Paulk

Mark.Paulk@utdallas.edu**, **Mark.Paulk@ieee.org

Jonsson School of Engineering and Computer Science

Measurement & Complexity Topics

→ **Goal-driven measurement**

Operational definitions

Driving behavior

What is complexity?

Possible software complexity measures

Using software complexity measures

Evaluating software complexity measures

Two Key Measurement Questions

Are we measuring the right thing?

- **Goal / Question / Metric (GQM)**
- **business objectives ⇔ data**
  - **cost (dollars, effort)**
  - **schedule (duration, effort)**
  - **functionality (size)**
  - **quality (defects)**

Are we measuring it right?

- **operational definitions**

Goal-Driven Measurement

Goal / Question / Metric (GQM) paradigm

- *V.R. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984.*

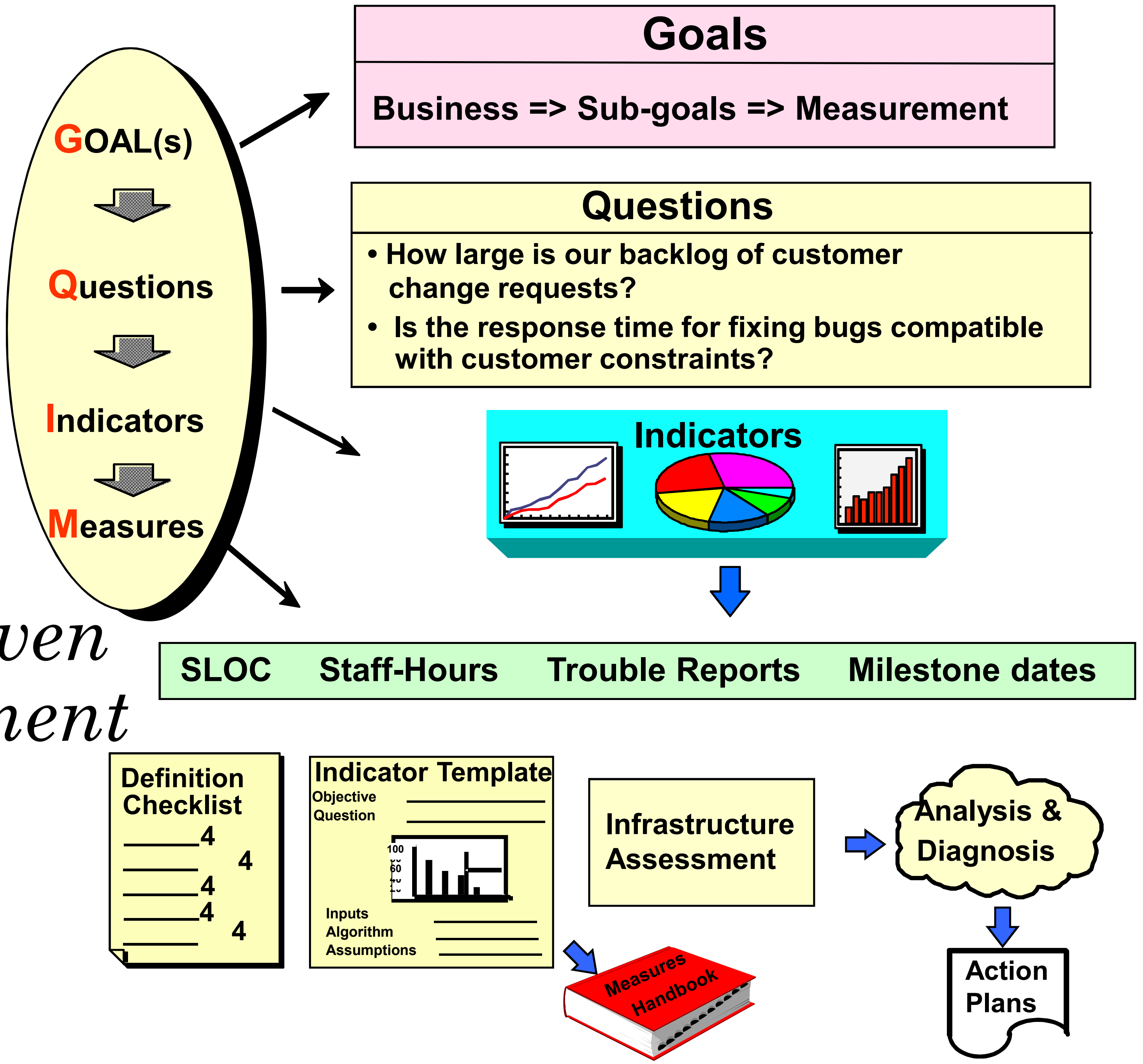
SEI variant: goal-driven measurement

- *R.E. Park, W.B. Goethert, and W.A. Florac, "Goal-Driven Software Measurement – A Guidebook," CMU/SEI-96-HB-002, August 1996.*

ISO 15939 and PSM variant: measurement information model

- *J. McGarry, D. Card, et al., Practical Software Measurement: Objective Information for Decision Makers, Addison-Wesley, Boston, MA, 2002.*

Goal-Driven Measurement



- How large is our backlog of customer change requests?
- Is the response time for fixing bugs compatible with customer constraints?

Measurement & Complexity Topics

Goal-driven measurement

→ Operational definitions

Driving behavior

What is complexity?

Possible software complexity measures

Using software complexity measures

Evaluating software complexity measures

Operational Definitions

The rules and procedures used to capture and record data

What the reported values <u>include</u> and <u>exclude</u>

Operational definitions should meet two criteria

- ***Communication*** **– will others know what has been measured and what has been included and excluded?**
- ***Repeatability*** **– would others be able to repeat the measurements and get the same results?**

SEI Core Measures

Dovetails with SEI's adaptation of goal-driven software measurement

Checklist-based approach with strong emphasis on operational definitions

Measurement areas where checklists have already been developed include:

- **effort**
- **size**
- **schedule**
- **quality**

See http://www.sei.cmu.edu/measurement/index.cfm

SLOC Definition Considerations

Whether to include or exclude

- **executable and/or non-executable code statements**
- **code produced by programming, copying without change, automatic generation, and/or translation**
- **newly developed code and/or previously existing code**
- **product-only statements or also include support code**
- **counts of delivered and/or non-delivered code**
- **counts of operative code or include dead code**
- **replicated code**

When the code gets counted

- **at estimation, at design, at coding, at unit testing, at integration, at test readiness review, at system test complete**

Common Software Information Categories (McGarry 2002)

Schedule and progress **– achievement of milestones, completion of work units**

Resources and cost **– balance between work to be performed and personnel resources assigned**

Product size and stability **– stability of functionality**

Product quality **– ability of product to support user's needs without failure**

Process performance **– capability of the supplier relative to the project needs**

Technical effectiveness **– viability of proposed technical approach**

Putnam and Myers' Five Core Metrics

Size

- **quantity of function, usually in SLOC or function points**

Productivity

- **functionality produced for the time and effort expended**

Time

- **duration of the project in calendar months**

Effort

- **amount of work expended in person-months**

Reliability

- **defect rate (or mean time to defect)**

Measurement & Complexity Topics

Goal-driven measurement

Operational definitions

→ Driving behavior

What is complexity?

Possible software complexity measures

Using software complexity measures

Evaluating software complexity measures

Dysfunctional Behavior

Austin's Measuring and Managing Performance in Organizations

- **motivational versus information measurement**

Deming strongly opposed performance measurement, merit ratings, management by objectives, etc.

Dysfunctional behavior resulting from organizational measurement is inevitable unless

- **measures are made "perfect"**
- **motivational use impossible**

Measurement & Complexity Topics

Goal-driven measurement

Operational definitions

Driving behavior

→ What is complexity?

Possible software complexity measures

Using software complexity measures

Evaluating software complexity measures

Complexity from a Business Perspective

S. Kelly and M.A. Allison, The Complexity Advantage, 1999.

- **nonlinear dynamics**
- **open and closed systems**
- **feedback loops**
- **fractal structures**
- **co-evolution**
- **natural elements of human group behavior**
  - **exchange energy (competition to collaboration)**
  - **share information (limited to open and fully)**
  - **align choices for interaction (shallow to deep)**
  - **co-evolve (from on-the-fly to with-coordination)**

Nonlinear dynamics → small differences at the start may lead to vastly different results

- **the butterfly effect**

Open systems → the boundaries permit interaction with the environment

Feedback loops → a series of actions, each of which builds on the results of prior action and loops back in a circle to affect the original state

- **amplifying and balancing feedback loops**

Fractal structures → nested parts of a system are shaped into the same pattern as the whole

- self-similarity
- software design patterns may contain other patterns…

Co-evolution → continual interaction among complex systems; each system forms part of the environment for all other systems

- system of systems
- simultaneous and continual change
- species survive that are most capable of adapting to their environment as it changes over time

Software Complexity

Complexity is everywhere in the software life cycle… usually an undesired property… makes software harder to read and understand… harder to change

- ***I. Herraiz and A.E. Hassan, "Beyond Lines of Code: Do We Need More Complexity Metrics?" Chapter 8 in <u>Making Software: What Really Works, and Why We Believe It</u>, A. Oram and G. Wilson (eds), 2011, pp. 125-141.***

Dependencies between seemingly unrelated parts of a system… (unplanned) couplings between otherwise independent system components

- ***G.J. Holzmann, "Conquering Complexity," IEEE Computer, December 2007.***

A Vague Concept

Not always clear what "complexity" is measuring...

Characteristics include difficulty of implementing, testing, understanding, modifying, or maintaining a program.

E.J. Weyuker, "Evaluating Software Complexity Measures," September 1988.

Measurement & Complexity Topics

Goal-driven measurement

Operational definitions

Driving behavior

What is complexity?

→ Possible software complexity measures

Using software complexity measures

Evaluating software complexity measures

Potential Software Complexity Measures

Lines of code

Source lines of code

Number of functions

McCabe cyclomatic complexity
- **maximum of all functions**
- **average over functions**

Coupling and cohesion

Halstead’s software science
- length
- volume
- level
- mental discriminations

Oviedo’s data flow complexity

Chidamber and Kemerer’s object oriented measures

Knot measure
- for a structured program, the knot measure is always 0

Fan-in, fan-out

Henry and Kafura's measure depends on procedure size and the flow of information into procedures and out of procedures.

- **length x (fan-in x fan-out)**
  - ***S. Henry and D. Kafura, "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics," Software Practice and Experience, June 1984.***

And so forth…

(Source) Lines of Code

LOC – total number of lines in a source code file, including comments, blank lines, etc.

- **countable using the Unix wc utility**

SLOC – any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line

- **includes program headers, declarations, executable and non-executable statements**

I. Herraiz and A.E. Hassan, "Beyond Lines of Code: Do We Need More Complexity Metrics?" Chapter 8 in <u>Making Software: What Really Works, and Why We Believe It</u>, A. Oram and G. Wilson (eds), 2011, pp. 125-141.

McCabe Cyclomatic Complexity

In the control flow graph for a procedure reachable from the main procedure containing

- **N nodes**
- **E edges**
- **p connected procedures**
  - **only procedures that are reachable from the main procedure**

V(G) = E – N + 2p

T. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, September 1976.

Using p

Does p allow analysis of a collection of programs?

- **programs with nested functions**
- **typically only look at a single program rather than a “library” with many disconnected routines**

Herraiz and Hassan (2011) use the maximum or average cyclomatic complexity for all functions in a file.

Recommended Ranges for Cyclomatic Complexity

V(G) should be less than 10

- **commonly accepted range**

Mathur recommends less than 5

Some suggest that 10-20 should be classified as “challenging”

Halstead's Software Science

M.H. Halstead, Elements of Software Science, 1977.

N_1	**number of operators in a program**
N_2	**number of operands in a program**
η_1	**number of unique operators in a program**
η_2	**number of unique operands in a program**
η	**program vocabulary** $= \eta_1 + \eta_2$
N	**program length** $= N_1 + N_2$
V	**program volume** $= N \times \log_2 \eta$
D	**difficulty** $= (\eta_1 / 2) \times (N_2 / \eta_2)$
lv	**level** $= 1 / D$
E	**effort (number of mental discriminations)** $= D \times V$
B	**number of delivered bugs** $= V / 3000$

Counting Rules for Halstead

Do you count ; and , as operators?

Do you count “paired” reserved words as distinct operators?

- **{}, (), if-then, begin-end, …**

Do you count syntactical markers?

- **end if, end loop, …**

Do you count (unary | binary) operators (e.g., minus or negative) as a single operator? Or two distinct operators?

Halstead's E

Halstead's E is in terms of discriminations per second

- **Stroud number is 18 discriminations / second**

To convert Halstead's E to Schneider's E, one correction factor is

18 discriminations/sec * 60 sec/min * 60 min/hr * 8 hr/day * 17 day/mon

= 8,812,800 discriminations/month

Oviedo's Data Flow Complexity

Given the basic blocks from a control flow graph…

A block's data flow complexity is the count of all prior definitions of locally exposed variables in block i which reach block i.

Data flow complexity of a program is the sum of the data flow complexities of each block in the program body.

- **only interblock data flow contributes to the complexity of a program body**
- **closely related to the all-uses test adequacy criterion**

Lack of Cohesion of Methods (LCOM)
(Chidamber and Kemerer, 1994)

Take each pair of methods in a class. If they access disjoint sets of instance variables, increase P by one. If they share at least one variable access, increase Q by one.

LCOM = P – Q	**if P > Q**
LCOM = 0	**otherwise**

- **LCOM = 0 indicates a cohesive class.**
- **LCOM > 0 indicates that the class needs or can be split into two or more classes, since its variables belong in disjoint sets.**
- **Classes with a high LCOM have been found to be fault-prone.**
- **A high LCOM value indicates disparateness in the functionality provided by the class.**

Tight and Loose Class Cohesion
(Bieman and Kang, 1995)

Methods a and b are related if

- **they both access the same class-level variable**
- **the call trees starting at a and b access the same class-level variable.**
  - **if a call goes outside the class, we stop following that call branch**

When two methods are related this way, we call them <u>directly connected</u>.

When two methods are not directly connected, but they are connected via other methods, we call them <u>indirectly connected</u>.

NP = maximum number of possible connections
= N * (N-1) / 2
where N is the number of methods

NDC = number of direct connections
- number of edges in the connection graph

NID = number of indirect connections

Tight class cohesion (connection density)
- TCC = NDC / NP

Loose class cohesion (overall connectedness)
- LCC = (NDC + NID) / NP

TCC is in the range 0…1

LCC is in the range 0…1

TCC<=LCC

The higher TCC and LCC, the more cohesive the class is.

TCC < 0.5 and LCC < 0.5 are considered non-cohesive classes.

- **LCC = 0.8 is considered "quite cohesive"**
- **TCC = LCC = 1 is a maximally cohesive class: all methods are connected**

Measuring Coupling *(Wikipedia)*

For data and control flow coupling

- **d_i: number of input data parameters**
- **c_i: number of input control parameters**
- **d_o: number of output data parameters**
- **c_o: number of output control parameters**

For global coupling

- **g_d: number of global variables used as data**
- **g_c: number of global variables used as control**

For environmental coupling

- **w: number of modules called (fan-out)**
- **r: number of modules calling the module under consideration (fan-in)**

$$\text{Coupling (C)} = 1 - \frac{1}{d_i + 2\,c_i + d_o + 2\,c_o + g_d + 2\,g_c + w + r}$$

Coupling(C) is larger the more coupled the module is.

- **This number ranges from approximately 0.67 (low coupling) to 1.0 (highly coupled).**

Measurement & Complexity Topics

Goal-driven measurement

Operational definitions

Driving behavior

What is complexity?

Possible software complexity measures

→ Using software complexity measures

Evaluating software complexity measures

Defects and Reliability

Defect prediction models

- **predict the number of defects in a module or system**
- **predict which modules are defect-prone**

Reliability models

- **predict failures (usually mean-time-to-failure MTTF)**

Be wary of attempts to equate defect densities with failure rates!

Who Uses?

Defect prediction models are used during development.

- **by project management and the development team**
- **to focus effort on the parts of the system that need the most attention**
- **to understand the impact of selected processes, techniques, and tools on quality**

Reliability models can be used during testing to determine where the software is ready to release.

- **to understand the quality of the operational software**

Causal Factors for Defects

Difficulty of the problem

Complexity of designed solution

Programmer/analyst skill

Design methods and procedures used

N.E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," IEEE Transactions on Software Engineering, September/October 1999.

Explanatory Variables for Predicting Defects

Size measures (LOC)

Complexity measures

- **McCabe cyclomatic complexity**
- **Halstead software science: effort**
- **count of procedures**
- **Henry and Kafura’s Information Flow Complexity**
- **Hall and Preisser’s Combined Network Complexity**

OO structural measures (Chidamber and Kemerer)

Code churn measures

- **amount of change between releases**

Process change and fault measures

- experience
- number of developers making changes
- number of defects in previous releases
- number of LOC added/changed/deleted

Limits of Using Size and Complexity Measures to Predict Defects

Models using size and complexity metrics are structurally limited to assuming that defects are solely caused by the internal organization of the software design and cannot explain defects introduced because

- **the “problem” is “hard”**
- **problem descriptions are inconsistent**
- **the wrong “solution” is chosen and does not fulfill the requirements**

Techniques Used

Regression models

- **multicollinearity is a problem**

Factor analysis / principal component analysis

Bayesian belief networks

Artificial neural networks

Capture-recapture

Capture-Recapture

Uses the overlap between the sets of defects found by different reviewers to estimate residual defects

Assumptions

- **reviewers work independently of each other**
- **searching is performed before, and not during, an inspection meeting**

If the overlap is large, few defects are left to be detected.

If the overlap is small, many faults are undetected.

History of Capture-Recapture

First known use of capture–recapture was by Laplace (1786), who used it to estimate the population size of France

In biology, capture–recapture is used to estimate the population size of animals in an area

Lincoln-Petersen Method

N = M C / R

N – estimate of total population size

M – total number of animals captured and marked on the first visit

C – total number of animals captured on the second visit

R – number of animals captured on the first visit that were then recaptured on the second visit

Example

Capture 10 specimens on a first visit and mark them

Capture 15 specimens on a second visit

- **5 are marked from the first visit**

N = M C / R = (10) (15) / 5 = 30

Chapman Estimator

A less biased estimator for small samples

N = [(M + 1) (C + 1) / (R + 1)] – 1

var(N) = [(M + 1) (C + 1) (M – R) (C – R)] /
[(R + 1) (R + 1) (R + 2)]

Example

- **N = [(10 + 1) (15 + 1) / (5 + 1)] -1 = 29.3**
- **var(N) = (11*16*5*10) / (6*6*7) = 34.9**
- **std(N) = sqrt(34.9) = 5.9**

Capture-Recapture Models in Software Engineering

Basic model (M0) assumes that all faults are equally probable to be found and that all reviewers have equal abilities to find faults

Mh model – the probabilities of fault detection vary

Mt model – abilities of reviewers vary

Mth model – both the probabilities of fault detection and the abilities of reviewers vary

Capture-Recapture Estimators

M0

- **M0–ML – maximum likelihood (Otis, 1978)**

Mt

- **Mt–ML – maximum likelihood (Otis, 1978)**
- **Mt–Ch – Chao’s estimator (Chao, 1989)**

Mh

- **Mh–JK – Jackknife (Burnham, 1978)**
- **Mh–Ch – Chao’s estimator (Chao, 1987)**

Mth

- **Mth–Ch – Chao’s estimator (Chao, 1992)**

Goodness of Capture-Recapture

For four reviewers and more, Mh–JK is preferable

Mt–Ch is the best estimator for two reviewers

Most models underestimate, but false positives inflate the estimate

Reinspections

If a reinspection is made, knowledge grows about the artifact

Biffl and Grossman (2001) approaches to using additional information

a) **first combine the data from the inspections and then estimate**
b) **add the number of faults detected in the first inspection to an estimate of the reinspection**
c) **estimate the first inspection and the reinspection separately and then add their results**

Best approach is (a), which improved estimators significantly

Bayesian Belief Networks

Fenton and Neil (1999) concluded that Bayesian belief nets were the best solution.

- **Explicit modeling of "ignorance" and uncertainty in estimates, as well as cause-effect relationships.**
- **Makes explicit those assumptions that were previously hidden - hence adds visibility and auditability to the decision-making process.**
- **Intuitive graphical format makes it easier to understand chains of complex and seemingly contradictory reasoning.**
- **Ability to forecast with missing data.**
- **Use of "what-if?" analysis and forecasting of effects of process changes.**
- **Use of subjectively or objectively derived probability distributions.**
- **Rigorous, mathematical semantics for the model.**

Performance of Defect Prediction Models

Precision – proportion of units predicted as faulty that were faulty

Recall – proportion of faulty units correctly classified

F-Measure – harmonic mean of precision and recall

- **(2 * recall * precision) / (recall + precision)**

T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "Developing Fault-Prediction Models," IEEE Software, November/December 2011.

Most models peak at about 70% recall.

Models based on naïve Bayes and logistic regression seem to work best.

Models that use a wide range of metrics perform relatively well.

- **source code, change data, data about developers**

Models using LOC metrics performed surprisingly well.

Successful defect prediction models are built or optimized to specific contexts.

Challenges in Using Defect Prediction Models *(Fenton, 1999)*

Difficult to determine in advance the seriousness of a defect

Great variability in the way systems are used by different users, resulting in wide variations of operational profiles

Difficult to predict which defects are likely to lead to failures (or to commonly occurring failures)

- **33% of defects led to failures with a MTTF greater than 5,000 years**
- **proportion of defects which led to a MTTF of less than 50 years was around 2%**

Software Reliability

J.D. Musa, A. Iannino, and K. Okumoto, Software Reliability: Measurement, Prediction, Application, 1987.

Probability of failure-free operation of a computer program for a specified time in a specified environment.

Reliability is defined with respect to time.

- **execution time**
- **calendar time**

Characterizing failure occurrences in time

- **time of failure**
- **time interval between failures**
- **cumulative failures experienced up to a given time**
- **failures experienced in a time interval**

The Random Nature of Failures

Mistakes by programmers, and hence the introduction of defects, is a complex, unpredictable process.

Conditions of execution of a program are generally unpredictable.

Failure behavior is affected by two principal factors

- **number of defects in the software being executed**
- **execution environment or operational profile of execution**

Nonhomogenous Processes

A random process whose probability distribution varies with time is called nonhomogeneous.

Musa’s basic execution time model and logarithmic Poisson execution time model assume that failures occur as a (NHPP) nonhomogeneous Poisson process.

Predicting Reliability

Stochastic reliability growth models can produce accurate predictions of the reliability of a software system providing that a reasonable amount of failure data can be collected for that system in representative operational use.

Unfortunately, this is of little help in those many circumstances when we need to make predictions before the software is operational.

N. Fenton and M. Neil, "Software Metrics: Successes, Failures, and New Directions," The Journal of Systems and Software, July 1999.

Measurement & Complexity Topics

Goal-driven measurement

Operational definitions

Driving behavior

What is complexity?

Possible software complexity measures

Using software complexity measures

→ **Evaluating software complexity measures**

Properties of Measures (Kearney 1986)

J.K. Kearney, R.L. Sedlmeyer, W.B. Thompson, M.A. Gray, and M.A. Adler, "Software Complexity Measures," Communications of the ACM, November 1986.

- **Robustness**
  - **not reduce the measure via incidental changes**
- **Normativeness**
  - **identify an acceptable level of complexity**
- **Specificity**
  - **identify what contributes to complexity**
- **Prescriptiveness**
  - **suggest methods to reduce complexity**
- **Property definition**
  - **determine whether properties are satisfied**

Weyuker's Properties

E.J. Weyuker, "Evaluating Software Complexity Measures," IEEE Transactions on Software Engineering, September 1988.

Propose properties that permit us to formally compare software complexity models.

- **not an informal discussion of pros and cons**
- **not an empirical study of correlation**

Weyuker Property 1

A measure that rates all programs as equally complex is not really a measure.

There exists P, Q, such that $|P| \neq |Q|$.

Weyuker Property 2

Let c be a nonnegative number. There are only finitely many programs of complexity c.

The measure should not be too “coarse.”

There should be more than just a few complexity classes.

LOC, Halstead fulfill Property 2.

Cyclomatic complexity, data flow complexity do not.

Cyclomatic complexity does not distinguish between prgrams that perform little computation and those that do massive amounts if they have the same control structure.

Weyuker Property 3

We do not want too fine a measure – do not assign to every program a unique complexity (e.g., a Gödel numbering).

There are distinct programs P and Q such that |P| = |Q|.

Weyuker Property 4

Details of a program's implementation determine its complexity.

There exists P, Q, such that P is equivalent to Q and |P| ≠ |Q|.

Since program equivalence is undecidable, no usable measure can divide programs into complexity classes based on the equivalence of computations.

From a practical perspective, Properties 1 and 4 are equivalent.

Weyuker Property 5

For every P, Q then |P| ≤ |P;Q| and |Q| ≤ |P;Q|.

Complexity increases monotonically as programs are composed.

Property 5 does not hold for data flow complexity or Halstead effort.

Effort → It is difficult to imagine an argument that it would take more effort to produce the initial part of a program than to produce the entire program.

Weyuker Property 6

a) There exists P, Q, R such that |P| = |Q| and |P;R| ≠ |Q;R|

b) There exists P, Q, R such that |P| = |Q| and |R;P| ≠ |R;Q|

Does concatenation of programs affect the complexity of the resulting program in a uniform way?

Neither cyclomatic complexity nor LOC satisfy Property 6.

Property 6 holds for data flow complexity and Halstead effort.

Weyuker Property 7

Program complexity should be responsive to the order of the statements, and hence the potential interaction among statements.

There are P and Q such that Q is formed by permuting the order of the statements of P and |P| ≠ |Q|.

Property 7 does not hold for LOC, cyclomatic complexity, nor Halstead effort.

It does for data flow complexity.

Weyuker Property 8

If P is a renaming of Q, then |P| = |Q|.

Property 8 holds for LOC, cyclomatic complexity, Halstead, and data flow complexity.

It would not hold for a Gödel numbering measure.

Weyuker Property 9

At least in some cases, because of interaction, the complexity of concatenated programs is greater than the sum of their complexities.

There exists P, Q such that $|P| + |Q| < |P;Q|$.

Property 9 does not hold for LOC or cyclomatic complexity.

Property 9 holds for data flow complexity and Halstead effort.

An Interesting Question

Should the complexity of a program be no less than the sum of the complexities of each of its parts?

In general, a measure that views the complexity of a program as independent of its context will satisfy this property.

Would it take twice as much time to implement or understand P;P as to implement or understand P?

Consider this an interesting open question…

Summary of Weyuker's Findings

Property	LOC (statement count)	Cyclomatic complexity	Halstead effort	Data flow complexity
1	Yes	Yes	Yes	Yes
2	Yes	No	Yes	No
3	Yes	Yes	Yes	Yes
4	Yes	Yes	Yes	Yes
5	Yes	Yes	No	No
6	No	No	Yes	Yes
7	No	No	No	Yes
8	Yes	Yes	Yes	Yes
9	No	No	Yes	Yes

Criticisms of Weyuker's Properties

Not predicated on a single consistent view of complexity.

- **N.E. Fenton and S.L. Pfleeger, Software Metrics: A Rigorous & Practical Approach, Second Edition, 1997.**

Not consistent with the principles of scaling.

- **H. Zuse, "Properties of Software Measures," Software Quality Journal, 1992.**

May only give necessary but not sufficient conditions for good complexity measures.

- **J.C. Cherniavsky and C.H. Smith, "On Weyuker's Axioms for Software Complexity Measures," IEEE Transactions on Software Engineering, Vol. 17, 636-638, 1991.**

Do We Need More Complexity Measures? (Herraiz 2011)

All of the complexity measures they examined were highly correlated with lines of code.

Header files showed poor correlation between cyclomatic complexity and the rest of the measures.

Cyclomatic complexity → a great indicator for the number of paths that need to be tested

Halstead → there are always several ways of doing the same thing in a program

Syntactic complexity measures cannot capture the whole picture of software complexity.

Questions and Answers