Metrics That Matter

Tim Menzies, Justin S. Di Stefano, Mike Chapman, Ken McGill
Lane Department of Computer Science, West Virginia University
PO Box 6109, Morgantown, WV, 26506-6109, USA
tim@menzies.com, justin@lostportal.net, Robert.M.Chapman@ivv.nasa.gov, Kenneth.G.Mcgill@ivv.nasa.gov

Abstract

Within NASA, there is an increasing awareness that software is of growing importance to the success of missions. Much data has been collected, and many theories have been advanced on how to reduce or eliminate errors in code. However, learning requires experience. This article documents a new NASA initiative to build a centralized repository of software defect data; in particular, it documents one specific case study on software metrics. Software metrics are used as a basis for prediction of errors in code modules, but there are many different metrics available. McCabe is one of the more popular tools used to produce metrics, but, as will be shown in this paper, other metrics can be more significant.

1 Introduction

Software metrics are attributes of software which can describe numerous things, including, but not limited to, complexity, effort, quality and reliability. In the 21^{st} century, we have more than enough data to check for the relative merits of different software metrics. We shall show that, for the application studied here as well as for other applications, only a few metrics are superior error predictors. Which metrics are superior is not a constant, however, but differs for each project. We will argue that blind faith in any one metric is not a good practice, and all available metrics should be carefully analyzed for as many projects as possible. In this way, more information will be made available for consideration when selecting a metric, which will lead to informed and scientific decisions instead of blind faith and assumptions.

This paper specifically documents a case study performed on a NASA IV&V project. In that project, testing was partially driven by the McCabe complexity metrics. However, closer analysis revealed that McCabe was *not* the most useful predictor of error-prone modules for that project. In fact, two entirely different metrics turned out to be better predictors for error density in modules, specifically L(Halstead's Program Level) and LOC(Lines of Code).

The rest of this paper is structured as follows: §2 presents the background and mission of the NASA IV&V facility. §3 is a brief overview of the metrics which were collected for this study. §4 is a description of the project on which the research was performed, and §5 is the result of that research. §6 is a discussion of these results, and §7 is a summary of our conclusions.

2 Background

The NASA Independent Verification and Validation (IV&V) Facility in Fairmont, West Virginia is responsible for verifying that software developed or acquired to support NASA missions complies with the stated requirements. Additionally, the Facility validates that the software is suitable for its intended use. In short, the Facility ensures that the software is being developed properly, and that the right software is being developed or acquired.

Due to cost constraints, IV&V is generally applied to software modules which are determined to be most critical to mission success. While the Facility must always fully address those mission critical modules, there is a need for a quick and easy way to identify other modules which are not as critical, but may be more (or equally) error prone. A primary purpose of the repository is to identify early lifecycle measures which may predict for error prone software modules, thus allowing the IV&V Facility to more effectively apply the limited testing resources available to any project.

As the sole entity with the responsibility for IV&V of all NASA mission software, the IV&V Facility is in a unique position to create and maintain a master repository of software metrics. Under this charter, the IV&V Facility reviews requirements, code, and test results from NASA's most critical projects; hence, many of the required metrics are collected as a matter of course. No other organization has insight into such a broad range of NASA projects. This affords the IV&V Facility an unequalled opportunity to research not only the early life cycle indicators of software

Metric Type	Metric	Definiton
McCabe	v(G)	Cyclomatic Complexity
	ev(G)	Essential Complexity
	iv(G)	Design Complexity
	LOC	Lines of Code
Halstead	N	Length
	V	Volume
	L	Level
	D	Difficulty
	I	Intelligent Content
	E	Effort
	B	Error Estimate
	T	Programming Time
Line Count	LOCode	Lines of Code
	LOComment	Lines of Comment
	LOBlank	Lines of Blank
	LOCodeAndComment	Lines of Code and Comment
Operator/Operand	UniqOp	Unique Operators
	UniqOpnd	Unique Operands
	TotalOp	Total Operators
	TotalOpnd	Total Operands
Branch	BranchCount	Total Branch Count

Figure 1. Metric Groups.

quality, but other topics as well. Many large corporations have similar software metrics repositories; however, it is not always in their best interest to release data or results to the public. In the case of the IV&V Facility, the objective is to improve NASA's mission software regardless of the source. Sanitized data would be made available to NASA, industry, and academia to support software development and research by other organizations. This is consistent with the IV&V Facilities research vision of ”See more, learn more, tell more.”

3 Metrics

This section gives a brief overview of the software metrics which were collected in the course of this study. Inclusion in this section does not imply endorsement in any way by either NASA IV&V or WVU.

In order to facilitate the search for error-prone modules or functions, many tools have evolved over the past few years. One of the most popular ones (and the one being used extensively at NASA IV&V) is the McCabe IQ© package. This package can evaluate Ada, C and C++ source code, and provides many different types of software metrics. In particular, it can output all of the metrics found in Figure 1. These were the metrics used during this study.

3.1 McCabe

The McCabe metrics are a collection of four software metrics: essential complexity, cyclomatic complexity, design complexity and LOC [4, 5, 6]. Of these four, all but LOC are metrics which were developed by T. J. McCabe.

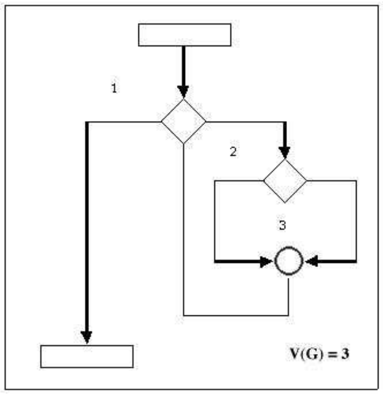

Figure 2. Example program flowgraph

McCabe & Associates claim that these complexity measurements provide insight into the reliability and maintainability of a module. For example, around NASA IV&V, a cyclomatic complexity of over 10 or an essential complexity of over 4 is flagged as a module that will be difficult to maintain and/or debug. This paper will not attempt to make any refutation to those claims and practices; however, these metrics are also commonly used as predictors for error-prone modules. As this paper will demonstrate, these complexity measurements do not *always* point the way towards modules with increased error density.

The following paragraphs present a short overview of the three complexity metrics mentioned previously.

Cyclomatic Complexity, or $v(G)$, measures the number of *linearly independent paths*[1] through a program's flowgraph[2]. $v(G)$ is calculated by:

$$v(G) = e - n + 2$$

where G is a program's flowgraph, e is the number of arcs in the flowgraph, and n is the number of nodes in the flowgraph [1]. For example, Figure 2 is a simple flowgraph; it's cyclomatic complexity is 3, since the graph has 6 arcs and 5 nodes ($v(G) = 6 - 5 + 2 = 3$).

Essential Complexity, or $ev(G)$, is the extent to which a flowgraph can be "reduced" by decomposing all the subflowgraphs of G that are D-structured primes [3]. $ev(G)$ is calculated by:

$$ev(G) = v(G) - m$$

[1]A set of paths is linearly independent if no path in the set is a linear combination of any other paths in the set.

[2]A flowgraph is a directed graph where each node corresponds to a program statement, and each arc indicates the flow of control from one statement to another.

[3]D-structured primes are also sometimes referred to as "proper one-entry one-exit subflowgraphs". For a more thorough discussion of D-primes, see [1].

where m is the number of subflowgraphs of G that are D-structured primes. [1]

Design Complexity, or $iv(G)$, is the cyclomatic complexity of a module's reduced flowgraph. The flowgraph, G, of a module is reduced to eliminate any complexity which does not influence the interrelationship between design modules. This complexity measurement reflects the modules calling patterns to its immediate subordinate modules [6].

3.2 Halstead

Another commonly used collection of software metrics are the Halstead Metrics [2]. They are named after their creator, Maurice H. Halstead. Halstead felt that software (or the writing of software) could be related to the themes which were being advanced at that time in the psychology literature. He created several metrics which are meant to encapsulate these properties; these metrics can be extracted by use of the McCabe IQ tool mentioned previously, and are discussed in detail below.

Halstead began by defining some basic measurements (these measurements are collected on a per module basis):

$$\begin{gathered}
\mu_1 = \text{number of unique operators} \\
\mu_2 = \text{number of unique operands} \\
N_1 = \text{total occurrences of operators} \\
N_2 = \text{total occurrences of operands} \\
\mu_1^* = \text{potential operator count} \\
\mu_2^* = \text{potential operand count}
\end{gathered}$$

These six metrics are self explanatory, with the possible exception of the potential operator/operand counts. Halstead defines μ_1^* and μ_2^* as the *minimum* possible number of operators and operands for a module. This minimum number would occur in a (potentially fictional) language in which the required operation already existed, possibly as a subroutine, function, or procedure. In such a case, $\mu_1^* = 2$, since at least two operators must appear for any function; one for the name of the function, and one to serve as an assignment or grouping symbol. μ_2^* represents the number of parameters, without repetition, which would need to be passed to the function or procedure.

Using these measurements, Halstead defined the *length* of a program P as:

$$N = N_1 + N_2$$

The vocabulary of P is:

$$\mu = \mu_1 + \mu_2$$

The *volume* of P, akin to the number of mental comparisons needed to write a program of length N, is:

$$V = N * log_2 \mu$$

V^* is the potential volume - the volume of the minimal size implementation of P.

$$V^* = (2 + \mu_2{}^*) log_2 (2 + \mu_2{}^*)$$

The *program level* of a program P with volume V is:

$$L = V^*/V$$

The inverse of level is *difficulty*:

$$D = 1/L$$

According to Halstead's theory, we can calculate an estimate $\hat{L}$ of L as:

$$\hat{L} = 1/D = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$$

The intelligence content of a program, I, is:

$$I = \hat{L} * V$$

The effort required to generate P is given by:

$$E = \frac{V}{\hat{L}} = \frac{\mu_1 N_2 N log_2 \mu}{2\mu_2}$$

where the unit of measurement E is elementary mental discriminations needed to understand P.

The required programming time T for a program of effort E is:

$$T = E/18 seconds$$

4 KC2 Project

The rest of this paper is dedicated to a case study on one NASA project, which will be referred to using the moniker "KC2". We will be addressing the relative merits of the above metrics when used as error-predictors for this project. KC2 is a C++ program which contains over 3000 modules.[4] 521 modules are of interest to us since these modules were

[4]A module, for the purposes of our tests, is the equivalent of a C function.

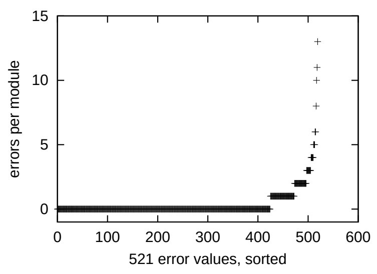

Figure 3. Distribution of errors: most modules have no errors.

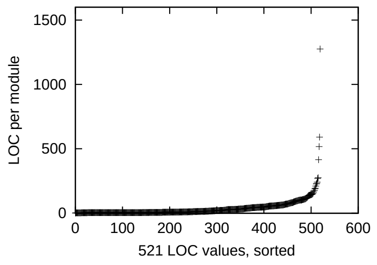

Figure 4. Distribution of LOC: most modules are very short.

built by NASA developers. The remaining 2500 or so modules are COTS[5] software.

Of those 521 modules, 106 were found to have various numbers of errors, ranging from 1 to 13. A graphical representation of the errors per module and lines of code per module are presented in Figure 3 and Figure 4, respectively.

5 The Case Study

5.1 Regression

The first test that was run on the provided metrics was a simplistic regression technique, in which a curve fitting algorithm was applied to each metric. In this particular project, *none* of the attributes were correlated to error rates. This does not mean, however, that none of the attributes are good error predictors; it merely indicates that in this domain, simplistic regression is unrevealing.

5.2 Treatment Learning

Since regression was not particularly insightful in this domain, we turned to other techniques. In order to better

[5]COTS is an acronym for Commercial Off The Shelf

```
ORIGINAL:
```

outlook	*temp(^oF)*	*humidity*	*windy?*	*class*
sunny	*85*	*86*	*false*	*none*
sunny	*80*	*90*	*true*	*none*
sunny	*72*	*95*	*false*	*none*
rain	*65*	*70*	*true*	*none*
rain	*71*	*96*	*true*	*none*
rain	*70*	*96*	*false*	*some*
rain	*68*	*80*	*false*	*some*
rain	*75*	*80*	*false*	*some*
sunny	*69*	*70*	*false*	*lots*
sunny	*75*	*70*	*true*	*lots*
overcast	*83*	*88*	*false*	*lots*
overcast	*64*	*65*	*true*	*lots*
overcast	*72*	*90*	*true*	*lots*
overcast	*81*	*75*	*false*	*lots*

```
SELECT class FROM original         SELECT class FROM original
WHERE outlook = 'overcast'         WHERE humidity >= 90

lots                               none
lots                               none
lots                               none
lots                               some
                                   lots
```

Figure 5. Attributes that select for golf playing behavior.

understand the key factors that predict for more/less errors, we performed a coarse-grained sensitivity analysis. In this analysis, the 521 examples were divided into two groups: the 20% of modules with errors and the 80% of modules without errors. This division was searched for a strong *select statement* that most changed the ratio of modules with/without errors.

To understand the concept of a "strong select statement", consider the log of golf playing behavior seen in Figure 5. In that log, we only play *lots* of golf in $\frac{6}{5+3+6} = 43\%$ of the cases. To improve our game, we might search for conditions that increases our golfing frequency. Two such searches are shown in the bottom of Figure 5. In the case of `outlook=overcast`, we play *lots* of golf all the time. In the case of `humidity` $\geq$ `90`, we only play *lots* of golf in 20% of the cases. The net effect of these two select statements is shown in Figure 6.

The `WHERE` statements within a select statement can contain conjunctions of arbitrary size. Exploring all such conjunctions manually is a tedious task. TAR2 is an automatic tool for finding the strongest select statements[3, 7, 8, 9]; i.e., the statement that *most* selects for preferred behavior while *most* discouraging undesirable behavior. TAR2 calls this strongest select statement the "treatment" since it is a recommended action for improving the current situation. The algorithm is automatic and, as used in this study, searched the entire range of possible conditions. TAR2's configuration file lets an analyst search for the best select statement using conjunctions of size 1,2,3,4, etc. Since

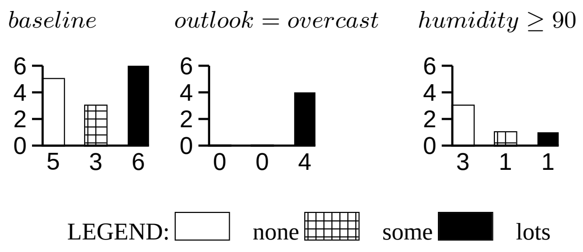

Figure 6. Changes to golf playing behavior from the baseline.

TAR2's search is elaborate, an analyst can automatically find the *best* and *worst* possible situation within a data set. For example, the select statements seen in Figure 6 were learnt by TAR2 and show the *best* and *worst* possible situation for playing *lots* of golf.

A common mistake that beginning treatment learners make is *over-training*. Over-training happens when you allow TAR2 to get *too* specific in its select statements, and your results end up too highly trained to be useful; i.e., your results, while extremely applicable to current data, are unlikely to apply to data seen in the future. One way of avoiding this pitfall is by assessing the learnt treatments against data not used during training. One method for doing so is N-way cross validation. In this process, a training set is divided into N buckets. For each bucket in turn, a select is learned on the other $N-1$ buckets, then tested on the bucket that was put aside. A select statement is deemed *stable* if it works in the majority of all N turns. TAR2 comes with an *N-way cross validation* tool that allows you to check the validity of a select statement.

After performing treatment learning and 10-way cross validation on our data set, the best(least errors) stable treatments were:

$$\boxed{L > 0.35} \text{ and } \boxed{T < 4.9}$$

TAR2 can also be used to find the *worst* action in the current situation merely by reversing the internal scoring mechanism. The worst(most errors) stable treatments found by TAR2 for this data set were:

$$\boxed{LOC > 118} \text{ and } \boxed{ev(G) > 7}$$

The effects of these select statements is shown in Figure 7, along with the results from the customary $v(G) > 10$ and $ev(G) > 4$ select statements. Note that although there are other select statements which will significantly alter the baseline distribution, none are as good as those listed above. TAR2 automatically searches through the entire space of all possible conjunctions of a specified size, and therefore assures that you can find the *best* possible select statement available for a data set.

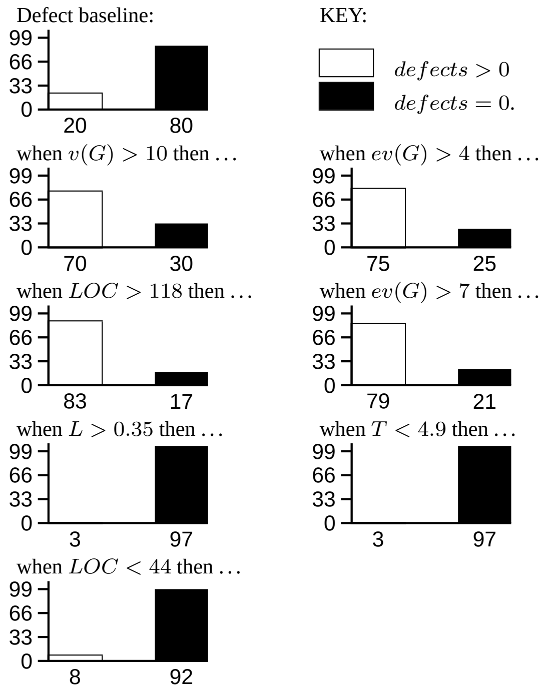

Figure 7. Results

Each of the distributions in Figure 7 should be compared to the baseline distribution, which can be found at the top of Figure 7. Clearly, $v(G) > 10$ and $ev(G) > 4$ are *both* good select statements, in that they significantly alter the distribution from the baseline. In addition, TAR2 was able to determine that $ev(G) > 7$ is actually a better select statement than either of the two customary McCabe metrics. It is interesting to note that $ev(G)$ actually performs better for KC2 than the more widely used $v(G)$. However, neither $ev(G)$ nor $v(G)$ are the *strongest* select statements to be using (when predicting for errors), since LOC has a better distribution.

In addition, the McCabe metrics are not useful at all when attempting to predict for error-free code. With a distribution of 97% error-free modules to 3% error-prone, $L > 0.35$ and $T < 4.9$ are actually *very* strong select statements in this domain. In addition, the simplistic LOC metric *again* proves it's usefulness, as it can be used to predict for error-free code as well, as the bottom of Figure 7 shows.

6 Discussion

This case study indicates that although McCabe complexity metrics are *OK* as error predictors, other metrics may prove to be better. It is also clear that simplistic regression techniques often shed little light on the error predicting capabilities of various metrics; only through more thorough

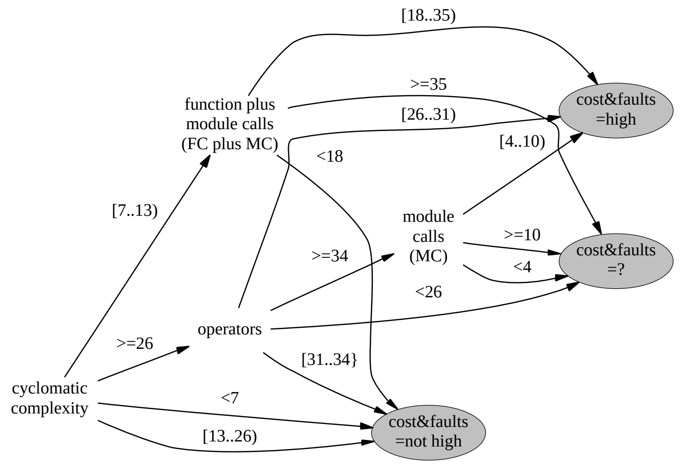

Figure 8. Excerpt from *Complexity measure evaluation and selection* **[11].**

testing can candidate attributes be detected. This study does *not* attempt to suggest which metrics should be used for any particular project; it only demonstrates that blind faith in one metric is certainly not justified for multiple domains.

It is not our purpose to defame or otherwise injure the reputation of $v(G)$ and $ev(G)$; certainly those metrics are useful and viable options. For instance, in a similar study conducted by J. Tian and M. Zelkowitz [11], it was found that cyclomatic complexity *was* the most useful attribute for error-prediction; Figure 8 is an excerpt from that article which demonstrates this fact. It is interesting to note, however, that other metrics can perform better; in fact, in this case study, the cheap and easy to collect LOC metric actually performs exceptionally well as *both* a selector for error-prone and error-free modules. It has been suggested by other researchers that in fact LOC may be a better metric to use when evaluating for error-prone code; the most notable example of this is Martin Shepperd's research [10]. Shepperd claimed that

> ...[Cyclomatic Complexity] is based upon poor theoretical foundations and an inadequate model of software development. The argument that the metric provides the developer with a useful engineering approximation is not borne out by the empirical evidence. Furthermore, it would appear that for a large class of software it is no more than a proxy for, and in many cases outperformed by, lines of code.

While we do not necessarily agree with Shepped's scathing views on cyclomatic complexity, we do acknowledge that he is correct, at least for this domain, in his opinion that $v(G)$ is often outperformed by LOC.

Based on these results, it is obvious that what is a good predictor for one project might not be at all useful on another. Certainly, a blind faith in any one attribute is a dangerous proposition. To sum the problem up succinctly, the pressing research question is: *Given* that good error predictors are project specific, *how early* in the development cycle can a developer find the important predictor(s) for their project?

7 Conclusion

In summary, we have shown two things which hold true in this particular domain:

- McCabe complexity metrics are *not* bad error-predictors, but others are better.
- LOC, a relatively cheap and easy-to-collect metric, is one of the best all-around error predictors.

Our future work addresses the pressing research question described above. More specifically, in order to mitigate the problem of finding good predictors for specific projects, we believe that projects must be studied from inception to completion; *i.e.*, given data sets with date stamps, how early can data from inception to present predict for errors in modules written one month ago? The results of such a study should at least give a general idea of the time frame before a good predictor can be found. Perhaps such a study might reveal that good predictors change during a project; for example, early in a project's life cycle, LOC may be a good predictor, while later on $v(G)$ might be better.

We are anticipating being able to perform just such a study on various projects at NASA. The results of these studies could greatly aid in error predicting and project dependency. Being able to predict for errors quickly and accurately will reduce the time spent debugging and testing, and should help to get software completed as quickly and efficiently as possible.

Acknowledgements

This research was conducted at West Virginia University under NASA contract NCC2-0979. The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not constitute nor imply its endorsement by the United States Government.

References

[1] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press, 1995.

[2] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[3] Y. Hu. Better treatment learning, 2003. Masters Thesis, Department of Electrical Engineering, University of British Columbia, in preperation.

[4] McCabe and Associates. Software metrics: Mccabe metrics. Available from `http://www.mccabe.com/metrics.php`.

[5] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[6] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12), December 1989.

[7] T. Menzies and Y. Hu. Reusing models for requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://tim.menzies.com/pdf/01reusere.pdf`.

[8] T. Menzies and Y. Hu. Agents in a wild world. In C. Rouff, editor, *Formal Approaches to Agent-Based Systems, book chapter*, 2002. Available from `http://tim.menzies.com/pdf/01agents.pdf`.

[9] T. Menzies and Y. Hu. Just enough learning (of association rules). In *KDD'02 (submitted)*, 2002. Available from `http://tim.menzies.com/pdf/02tar2.pdf`.

[10] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, pages 30–36, March 1988.

[11] J. Tian and M. Zelkowitz. Complexity measure evaluation and selection. *IEEE Transaction on Software Engineering*, 21(8):641–649, Aug. 1995.