A Critical Re-examination of Cyclomatic Complexity Measures

B. Henderson-Sellers^a and D. Tegarden^b

^a School of Computing Sciences, University of Technology, Sydney, Australia

^b Department of Accounting, Virginia Tech., USA

Abstract

Cyclomatic complexity is a recognized measure of one aspect of structural complexity. Here two measures are re-examined theoretically for four cases: (a) modularization to eliminate duplicate code, (b) multiple entry, single exit modules, (c) single entry, multiple exit modules and (d) multiple entry, multiple exit modules.

Keyword Codes: D.2.8, D.2.5

Keywords: Metrics, Testing

1. INTRODUCTION

In this paper, we clarify the differences between the original, standard McCabe¹ metric, $V(G)$ and a new cyclomatic complexity metric, which we² denote by $V_{LI}(G)$. Both metrics will be discussed in terms of (a) decision structures, (b) number of linearly independent (LI) paths for the formation of a basis set (both of these initially for single-module programs and then for larger systems) and then considered with respect to (c) modularization, (d) modularization to remove duplicate code, (e) programs containing multiple entry modules and (f) programs containing multiple exit modules. Initially we outline the complexity framework within which these cyclomatic complexity metrics are placed — they characterize “internal attributes” which contribute to the overall structural complexity of the document (i.e. design document or code). We then introduce a detailed mathematical discussion of two forms of the cyclomatic complexity metric which are theoretically valid and immediately applicable to testing. Since the measures are conceptually well-validated, it is inappropriate, and indeed unnecessary, to seek statistical corroboration.

2. FRAMEWORK FOR COMPLEXITY METRICS

Software has several characteristics which can, for the purpose of this discussion, be divided into two groups: internal characteristics and external characteristics³. Internal characteristics include size and control flow complexity. These are items for which it is possible to derive measures/metrics which can be applied objectively — control flow complexity being the focus of this paper. External characteristics (e.g. maintainability, reliability) are much harder to measure since, as Fenton notes, they must involve human and/or environmental considerations. One aim of metrics research is to identify functional relationships between these external characteristics and the measurable internal characteristics. One component of this research agenda is to ensure the theoretical, and hence practical, validity of the measures of the internal characteristics. The validity of the internal measures is thus of paramount importance in ensuring that good prognostic models for external characteristics can be built in this way.

In the interim, however, one use of internal metrics which may be justifiable and pragmatically useful, even in the absence of any link to an external characteristic, is as “alarms”^{4,5}. An alarm would occur

Figure 1 Classification of complexity metrics

whenever the value of a specific internal metric exceeded some pre-determined threshold which would draw attention to that particular part of the code.

Complexity, itself an external characteristic or attribute, has three "flavours": computational, psychological and representational (Figure 1). The most important of these is psychological⁶ which encompasses programmer characteristics, structural complexity and problem complexity. Programmer characteristics are hard to measure objectively⁷ and little work has been done to date on measures of problem complexity. The prime focus of "complexity research" has thus been on structural complexity which is measured by module metrics and intermodule metrics⁵. Modularity metrics are focussed at the individual module level (subprogram) and may be measures for (i) the internals of the module (procedural complexity) such as size, data structure or logic structure (control flow complexity) or (ii) the external specification of the module (semantic complexity) — typical module cohesion as viewed externally to the module. Intermodule metrics (measuring systems design complexity), on the other hand, describe the connections *between* modules and thus characterize the system level complexity.

Here we focus on measures for control flow complexity and specifically on two variants of the cyclomatic complexity measure.

3. CYCLOMATIC COMPLEXITY

McCabe's¹ cyclomatic complexity, $V(G)$, is a measure in widespread use for assessing the control complexity (an internal characteristic) in a program. It aims to provide a basis set for constructing a testing programme (at the unit level) as well as identifying heuristically subprograms which might be regarded as "overly complex".

A basis set comprises a number of linearly independent (LI) paths through the program such that all other paths can be constructed from members of the basis set by using a vector-based approach. It has also been shown³ that for a program in which there are only *binary* decisions (viz. all nodes have outdegree ≤ 2), the magnitude of this basis set is equally given as (no of decisions + 1) or, by Euler's theorem, as the number of discrete regions of the 2-D flowgraph. However, it is important to stress that, despite numerous claims to the contrary⁸, the cyclomatic complexity *does not* measure external characteristics such as reliability, development effort, understandability etc. Neither we nor McCabe make any claim that these metrics are theoretically related to psychological complexity which itself must include a strong, cognitive component.

Henderson-Sellers⁹ has introduced a revised equation for quantifying "cyclomatic complexity" which has properties upon code modularization not exhibited by the original¹ cyclomatic complexity — a concern raised initially by Shepperd⁸. Feghali and Watson¹⁰ have shown this to be equivalent to "integration complexity"¹¹ but with the module design complexity, $iv(G)$, replaced by $V(G)$. As such, this new formula, as pointed out¹⁰ represents the "integration complexity without design reduction" and "gives the cardinality of a basis set of paths through an entire program ...".

4. UNDERPINNING BY GRAPH THEORY

The cyclomatic complexity is based on graph theory¹² which says that for a *connected* graph, the graph cyclomatic number¹³, v , also called the first Betti number, the nullity or the cycle rank, is given as

$$v = e - n + 1 \quad (1)$$

and for a graph with p components

$$v = e - n + p \quad (2)$$

where e is the number of edges and n the number of nodes. A connected graph (Equation (1)) is a graph where *all* nodes are reachable from every other node. A disconnected graph is one with p components which are disjoint, but which could, with the addition of $(p - 1)$ additional edges, be turned into a connected graph (see figure 1 of ref 2). Since in a connected graph, a path from a to b is equivalently a path from b to a , there is no notion of "strongly connected" until the graph becomes directed, as in software representations. This means that, for a single program or program module, the connected, directed acyclic graph (DAG), does not satisfy the definition of connectedness until an edge is added from the stop node to the start node. The resultant graph is thus known as a strongly connected graph and has one more edge than its corresponding undirected graph (or the regular DAG equivalent). Thus the cyclomatic complexity, or cycle rank, is given for a single, strongly connected DAG by

$$V(G) = e - n + 2 \quad (3)$$

— a form frequently quoted in the literature¹⁴⁻¹⁶ as representing the standard McCabe metric. McCabe's¹ Theorem 1 says that in a strongly connected graph i.e. one in which p must have a value of unity, $V(G)$ is equal to the maximum number of linearly independent circuits. This is the basis set from which all other circuits (or for a DAG all other paths) can be created. This is seen as an important guideline and one we shall return to frequently below.

Equation (3) is useful for the cyclomatic complexity of a single component DAG. However, with respect to the extension of $V(G)$ to more than one component, there appear to be two options. Assuming that we are trying to describe a measurement effectively for a connected graph but expressed in terms of, say, three disjoint components, then the argument there is easily extended to give, inductively,

$$V_{LI}(G) = e - n + p + 1 \quad (4)$$

This is the form proposed by Henderson-Sellers⁹ and relabelled here with a subscript *LI* (linearly independent) as we shall later show that this metric (Equation (4)) is strongly related to the cycle rank of the graph.

Feghali and Watson¹⁰ point out that a different approach was taken by McCabe¹. He argued that *each* of the p components needs to be converted into a strongly connected graph, thus giving p graphs, themselves strongly connected, but not connected to each other. Here $V(G)$ is given by

$$V(G) = e - n + 2p \quad (5)$$

Thus each component is treated independently and the value given by Equation (5) is relevant to each component but not really to the integrated system. Indeed, Shepperd⁸ noted that this means that a program with several subroutines is "treated as unconnected components within the control graph" which has "the bizarre result of increasing overall complexity if a program is divided into more, presumably simpler, modules". Each additional subprogram isolated from the rest of the program increases the value of $V(G)$ by 1. Feghali and Watson¹⁰ interpret this additional complexity upon "brute-force modularization" as a test for the interface, rather than an addition to the integration test path basis, which is the focus of $V_{LI}(G)$ and the discussion here.

This concern led to the proposal⁹ of Equation (4) which has the properties that

- 1) modularization (prior to removal of repetitive code) has no effect on $V_{LI}(G)$.
- 2) the value of $V_{LI}(G)$ for the full program is equal to the total number of decisions, D , plus one.
- 3) the value of $V_{LI}(G)$ is unchanged when subroutines are merged back into the program either by nesting or sequence. This ties in with the argument that the integration testing procedures (for the present limited to single calls to single entry, single exit components) are unchanged by modularization.
- 4) the value of $V_{LI}(G)$ provides a basis set for integration testing or for unit testing of a single module.

A simple example will help here to illustrate the differences between $V(G)$ and $V_{LI}(G)$. In Figure 2, there is a main routine and two subroutines, A1 and A2. The values of $V(G)$ and $V_{LI}(G)$ are $V(G) =$

Figure 2 Main routine and two subroutines (A1 and A2) — depicted as three disconnected components.

Figure 3 Example of Figure 2 redrawn in terms of a single connected graph.

$19 - 18 + 2 \cdot 3 = 7$ and $V_{LI}(G) = 19 - 18 + 3 + 1 = 5$. In terms of testing paths, these graphs are equivalent to a single DAG, using the node splitting technique⁹, as shown in Figure 3, for which both $V(G)$ and $V_{LI}(G)$ have the same value, given by Equation (3) viz. $23 - 20 + 2 = 5$. In both figures, there are four decisions, whilst the basis set is more easily obtained from Figure 3. The (non-unique) basis set is

$$1\ 6; 2\ 4\ 3\ 7; 2\ 5\ 3\ 6; 2\ 5\ 3\ 8; 1\ 7$$

All other paths (1 8; 2 4 3 6; 2 4 3 8; 2 5 3 7) are expressible as linear combinations of elements of the basis set. For example, (1 8) = (1 6) + (2 5 3 8) - (2 5 3 6). Finally, both Figures 2 and 3 depict the infinite 2-D plane divided into five regions.

The relationship between the whole and the sum of the parts is also different. McCabe¹ shows that

$$V(G) = \sum V(G_i) \quad (6)$$

whereas it can be shown^{9,17} that, for subroutines merged, by nesting or sequence, into a main routine,

$$V_{LI}(G) = \sum V_{LI}(G_i) + 1 - p \quad (7)$$

Thus $V(G)$ (or more accurately $V_{LI}(G)$) is given by

$$V(G) \equiv V_{LI}(G) = \sum_{i=1}^p (e_i - n_i + 2) - p + 1 = e^* - n^* + 2p - p + 1 = e^* - n^* + p + 1 \quad (8)$$

where e^* and n^* are the total edges and nodes respectively. The difference here is that Fenton and Kaposi¹⁷ as well as Henderson-Sellers⁹ are considering that summing components means merging them (by sequence or iteration); whereas the McCabe approach (Equation (6)) sums *without* merging. Thus the focus is at the *module* level, where the most appropriate formula is Equation (3). $V_{LI}(G)$ retains an interpretation with respect to testing paths both at the unit and at the program level, whereas $V(G)$ treats modules in a program essentially independently.

For the example discussed above, Figure 2 has three components with component $V(G_i)$ values equal to 2, 2 and 3 respectively. Thus from Equation (7) we get $V_{LI}(G) = 2 + 2 + 3 + 1 - 3 = 5$ as derivable directly in Figure 3. On the other hand, Equation (6) gives $V(G) = 7$ which does not relate to criterion of the number of LI paths which is clearly 5.

Henderson-Sellers⁹ only considered modularization in the sense of dividing a program into modules and did not address the consequence of the second reason for modularization: the elimination of duplicate code chunks. Based on our Equations (4) and (7), we now extend the $V_{LI}(G)$ metric to cases beyond these single entry/single exit modules⁹ i.e. to (i) modularization in order to eliminate duplicate code, (ii) multiple entry single exit modules, (iii) single entry multiple exit modules and (iv) multiple entry multiple exit modules.

Figure 4 (a) Three modules (single IF THEN ELSE blocks) linked linearly ($p = 1$); (b) modularization into main routine and three subroutines ($p = 4$); (c) elimination of duplicate code ($p = 2$).

Figure 5 Multiple entry, single exit modules — example DAGs

4.1 Modularization to Eliminate Duplicate Code

Consider a single program which contains u blocks of duplicate code, each with e' edges and n' nodes embedded within it. Let us consider the simplest case: that these u blocks are connected by simple edges (Figure 4(a)). Since there are u "chunks" of duplicate code connected together linearly, there are $u - 1$ edges of connection. The total complexity of this single module ($p = 1$) is thus given as

$$V_{LI}(G) = (ue' + u - 1) - un' + 2 = u(e' - n') + u + 1 \quad (9)$$

We now consider the modularization of this program. Since there are u duplicate chunks, the modularized program has one main routine which has u nodes and $u - 1$ edges and u other components (viz. $p = u + 1$) (Figure 4(b)). The value of $V_{LI}(G)$ is unchanged. The concept that these $p - 1$ components are duplicates and can be replaced by a single module (Figure 4(c)) is introduced. Now there is the main routine plus only one other module: that of the duplicated chunk. This has e' edges and n' nodes. For the main module, $V_{LI}(G_1) = (u - 1) - u + 2 = 1$ and for the submodule, $V_{LI}(G_2) = e' - n' + 2$. Hence, from Equation (7)

$$V_{LI}(G) = V_{LI}(G_1) + V_{LI}(G_2) + 1 - 2 = 1 + e' - n' + 2 + 1 - 2 = (e' - n') + 2 \quad (10)$$

Since the value of the unmodularized code is given by Equation (9) above, it can be seen that the modularization to deal with each chunk of repeated code gives a reduction in the value of the cyclomatic complexity of $(ue' - un' + u + 1) - (e' - n' + 2) = (u - 1)(e' - n' + 1)$. In other words, as noted in ref 10 yet consistently with ref 9, the cyclomatic complexity of a program containing duplicate chunks explicitly embedded in the main program (Figure 4(a)) is greater than the modularized version where

the duplication is removed from the graph depicting the program (Figure 4(c)). Additional CALLs to the same subprogram engender no extra *structural* complexity since no additional test paths are created. Feghali and Watson's¹⁰ comment on the dramatic increase of "complexity" in this situation is relevant to cognitive complexity but not structural complexity or the testing paths approach.

As a corollary, since the complexity of the repeated module can be expressed as $v = e' - n' + 2$, we can substitute this expression for v and calculate a decrease in complexity which gives a result identical to that given in ref 8. Summed over all k repeated modules leads to a decrease of $\sum_{i=1}^k (u_i - 1) \times (v_i - 1)$ where v is the complexity of the module being extracted. This reduction also reflects the smaller cyclic basis so that the test strategy avoids duplicate testing of the duplicate code chunks.

In contrast, the identical analysis using $V(G)$, rather than $V_{LI}(G)$, gives

- (a) as one module $V(G) = u(e' - n') + u + 1$ (cf. Equation (9) for $V_{LI}(G)$)
- (b) as $u + 1$ modules $V(G) = u(e' - n' + 2) + 1$ (cf. Equation (9) for $V_{LI}(G)$)
- (c) as two modules $V(G) = e' - n' + 3$ (cf. Equation (10) for $V_{LI}(G)$)
- (d) reduction (between (c) and (a)) $\Delta V(G) = (u - 1)(e' - n') + u - 2 = (u - 1)(v - 1) - 1$
- (e) reduction (between (c) and (b)) $\Delta V(G) = (u - 1)(e' - n' + 2) = (u - 1)v$

In terms of calculating $V_{LI}(G)$ for a program with several CALLs to the same subroutine, then additional CALLs in excess of one do not add to the value of $V_{LI}(G)$ since they introduce no further entries in the cyclic basis. Thus Equation (7) holds for subroutines with multiple CALLs. However, it should be noted that this conclusion should not be extrapolated to other classes of complexity measures, such as psychological complexity or understandability — for example, intuitively multiple CALLs may make a program harder to comprehend and debug (a topic which cannot be addressed by the structural cyclomatic complexity measures discussed here).

4.2 Multiple Entry, Single Exit Modules

Whilst multiple entry modules are generally not frequently encountered, and indeed discouraged by some authors, it is important that any validated metric be equally applicable to multiple entry modules as to single entry modules. In the context of cyclomatic complexity metrics, the results of the previous section apply directly. Multiple entries are simply a method of code reuse. Thus the arguments for duplicate code expressed above remain valid and $V_{LI}(G)$ is calculated from Equation (7).

For example, Figure 5(a) depicts two simple program modules each with a cyclomatic number of two (one decision per module). There are clearly three vectors in the cycle basis. We suppose that the subprogram module, G_2 , has the normal entry point at the beginning of the module, but that it also has an entry point at node E_2 and E_3 as indicated. The value of $V_{LI}(G)$ from Equation (7) is found to be 3 ($= 2 + 2 + 1 - 2$). This has 2 decisions so again we expect $V_{LI}(G) = 1 + 2 = 3$. There are also three planar regions as anticipated from Euler's formula. This is shown as a single connected graph in Figure 5(b).

4.3 Single Entry, Multiple Exit Modules

Multiple exit points are frequently used in structured programming. Generally these occur in branches of IF structures where an immediate return to the CALLing module occurs (Figure 6(a)). In this case, the addition of a second RETURN at X_1 (in addition to the normal RETURN at the end of the module at X_2) can be represented in one of three ways (Figures 6(b-d)). Either an additional (fictitious) edge can be drawn to the exit node of the subroutine (Figure 6(b)), in which case one is added to the value of $V_{LI}(G)$ for every such RETURN; or, using node splitting to change the disconnected components of Figure 6(b) to the connected graph of Figure 6(c), an additional edge is drawn direct to the main module; and finally (Figure 6(d)), the subroutine can be fully embedded. In all cases, we can easily capture this additional complexity explicitly by adding one to $V_{LI}(G)$ for every additional exit point. Thus Equation (4) becomes

$$V_{LI}(G) = e - n + p + 1 + \sum_{j=1}^{p-1} (r_j - 1) \quad (11)$$

where e is the number of edges, n the number of nodes, p the number of connected components (modules), and r_j the number of exit points in the DAG representation of the j -th module (where there are $p - 1$

Figure 6 Single entry, multiple exit modules — example DAGs

such modules e.g. subroutines). This can be interpreted as an increase in the complexity of the program (cf. one not using multiple exits) by

$$\Delta V_{LI}(G) = \sum_{j=1}^{p-1} (r_j - 1) \quad (12)$$

— an approach also used in ref 15. In this example, the value of $V_{LI}(G)$ is found to be 3 ($=18 - 17 + 2$: Figure 6(c) or $16 - 15 + 2$: Figure 6(d)). Alternatively, Equation (11) can be applied directly to the disjoint graphs of Figure 6(a) to give $V_{LI}(G) = 15 - 16 + 3 + 1 = 3$. There are clearly 2 decisions and 3 regions.

4.4 Multiple Entry, Multiple Exit Modules

Multiple entry multiple exit (MEME) point modules are seldom encountered in structured programming. However, for completeness sake, we note briefly that since multiple entries do not change $V_{LI}(G)$, MEME modules can be treated identically to multiple exit modules to which Equation (11) applies.

5. DISCUSSION

The focus of both the measures $V(G)$ and $V_{LI}(G)$ is that of structural complexity. They can be related (with varying degrees of success) to the number of decisions and the provision of a basis set of testing paths (unit and system level, as discussed above). One difference relates to whether $V(G) = \sum V(G_i)$ which is true for McCabe's $V(G)$ but not for $V_{LI}(G_i)$; although the latter retains consistency with the notion of a basis set. At the system level, $V_{LI}(G)$ maintains a description of the basis set with regard to a testing methodology and complements the notion of integration complexity¹¹, $iv(G)$. The connection, as Feghali and Watson¹⁰ correctly note, is that $V_{LI}(G)$ provides a test-paths view equivalent to this integration complexity without design reduction, whereby decision nodes not involved in subprogram CALLs are ignored. Thus the formula for integration complexity¹¹, S_1 , given by

$$S_1 = \sum iv(G) + 1 - p \quad (13)$$

is closely paralleled by Equation (7) for $V_{LI}(G)$.

In closing, we propose that a modification to the $V_{LI}(G)$ approach also may be worth exploring. Rather than Equation (4), it is worth evaluating the measure

$$C(G) = e - n + p \quad (14)$$

This has the advantage that it represents, both at module and program level, the number of decisions rather than the number of decisions plus one; that is measurable on a ratio scale since it has an absolute zero such that transformations of the type $g(x) = ax$ are possible⁶; that the sum of the components is equal to complexity of the total program (thus satisfying Weyuker's¹⁸ 5th axiom). Conversely, the basis set is given by $C(G)+1$. At the same time, $C(G)$ does not change upon modularization and is unchanged when subprograms are nested or sequenced back into the main program. When a program is modularized to remove duplicate code, $C(G)$ is reduced by an amount $(u-1)v$ (cf. $(u-1) \times (v-1)$ for $V_{LI}(G)$). Finally, both $C(G)$ and $V_{LI}(G)$ provide the dimension of the cycle rank for developing integration testing methodologies.

Consequently, $C(G)$ may be a possible candidate for inclusion in compound psychological complexity measures; although it should be stressed that neither analytical nor empirical work yet undertaken seems convincing⁸.

6. ACKNOWLEDGEMENTS

We wish to thank David Monarchi for constructive discussions during the preparation of this paper. We wish to thank Ross Jeffery, June Verner and Yagna Pant for comments on an earlier draft of this paper.

REFERENCES

1. McCABE, T., 'A complexity measure', *IEEE Trans. Software Eng.*, 1976, Vol. SE-2, (4), pp. 308-320
2. HENDERSON-SELLERS, B. and TEGARDEN, D., Response to 'Clarification concerning modularization and McCabe's cyclomatic complexity', *Comm. ACM*, 1994, Vol. 37, (4), pp. 92-94
3. FENTON, N.E., 'Software Metrics: A Rigorous Approach', (Chapman and Hall, London, 1991)
5. SZENTESI, J. and GRAS, J., 'Some practical views of software complexity metrics and a universal measurement tool', First Australian Software Engineering Conference (ASWEC), Canberra, 14-16 May, 1986
6. KITCHENHAM, B.A. and LINKMAN, S.J., 'Design metrics in practice', *Inf. Soft. Technol.*, 1990, Vol. 32, (4), pp. 304-310
7. ZUSE, H., 'Software Complexity: Measures and Methods', (Walter de Gruyter, Berlin, 1990)
9. KEARNEY, J.K., SEDLMAYER, R.L., THOMPSON, W.B., GRAY, M.A. and ADLER, M.A., 'Software complexity measurement', *Comm. ACM*, 1986, Vol. 29, (11), pp. 1044-1050
10. SHEPPERD, M., 'A critique of cyclomatic complexity as a software metric', *Software Engineering Journal*, 1988, Vol. 3, pp. 30-36
11. HENDERSON-SELLERS, B., 'Modularization and McCabe's cyclomatic complexity', *Comm. ACM*, 1992, Vol. 35, (12), pp. 17-19
12. FEGHALI, I. and WATSON, A.H., 'Clarification concerning modularization and McCabe's cyclomatic complexity', *Comm. ACM*, 1994, Vol. 37, (4), pp. 91-92
13. McCABE, T.J. and BUTLER, C.W., 'Design complexity measurement and testing', *Comm. ACM*, 1989, Vol. 32, (12), pp. 1415-1425
14. BOLLOBÁS, B., 'Graph Theory. An Introductory Course', (Springer-Verlag, New York, 1979)
15. BERGE, C., 'Graphs', 2nd ed., (North-Holland, Amsterdam, 1985)
20. BAKER, A.L. and ZWEBEN, S.H., 'A comparison of measures of control flow complexity', *IEEE Trans. Software Eng.*, 1980, Vol. SE-6, (6), pp. 506-512
21. HARRISON, W.A., 'Applying McCabe's complexity measure to multiple-exit programs', *Software — Practice and Experience*, 1984, Vol. 14, (10), pp. 1004-1007
22. CONTE, S.D., DUNSMORE, H.E. and SHEN, V.Y., 'Software Engineering Metrics and Models', (Benjamin/Cummings, Menlo Park, CA, 1986)
24. FENTON, N.E. and KAPOSI, A.A., 'Metrics and software structure', *Inf. Software Technol.*, 1987, Vol. 29, (6), pp. 301-320
25. WEYUKER, E.J., 'Evaluating software complexity measures', *IEEE Trans. Software Eng.*, 1988, Vol. 14, (9), pp. 1357-1365