GRAPH THEORY APPLICATIONS IN DEVELOPING SOFTWARE TEST STRATEGIES FOR NETWORKING SYSTEMS

Vladimir V. Riabov*
Associate Professor, Department of Mathematics and Computer Science, Rivier College

Keywords: *cyclomatic complexity metrics, test and code coverage, embedded networking systems*

Abstract

The graph-based metrics (cyclomatic complexity, essential complexity, module design complexity, system design complexity, and system integration complexity) are reviewed and applied for studying the C-code complexity and estimating the number of integration and unit tests for two networking systems: Carrier Networks Support system with switches and routers, and Aggregation System for networking services. Comparing different code releases, it is found that the reduction of the code complexity leads to significant reduction of errors and maintainability efforts. The test and code coverage issues for embedded networking systems are also discussed.

1 Introduction

The issues of software quality are usually addressed in our Software Engineering course. Among other approaches [1, 2], students study in depth the McCabe's structured testing methodology [3] that became a widely used method in the complexity code analysis, independent logical path testing, integration test planning, and test coverage estimating in different industries. The approach was developed by McCabe [4] who applied the graph-theoretical complexity measuring techniques in studies of management and controlling the program (code) complexity. Based on the experimental results of Miller [5], McCabe suggests that the code modules approach zero defects when the module cyclomatic complexity is less than 10. Since 1996, this methodology becomes the standard, which was recommended by the National Institute of Standards and Technology [3]. Following specially designed computer-lab assignments, the students study how to apply the theory of graphs for the complexity code analysis, developing unit and integration test strategies, and predicting possible errors in the codes developed by themselves and industrial partners.

Nowadays, the McCabe's QA tools [6-9] become available for software designers and test engineers [3]. Unfortunately, many engineers (including those from some networking companies) are not familiar with the structured testing methodology and continue using "traditional" metrics (e.g., Reviewed-Lines-Of-Code [1, 2]) in their testing practice. As a result, the quality of networking-service software and products is low, and testing, debugging, and sustaining efforts are tremendous.

In the present study, we present an overview of the graph-based complexity metrics and the results of systematic metric analyses of networking-systems software for two industrial projects: Carrier Networks Support system with switches and routers (project A) and Aggregation System for networking services (project B). It has been found that the number of unreliable code functions correlates well with the number of customer requests, error-fixing submits, and the possible errors, which have been estimated with the McCabe's and Halstead's metrics [1-3, 6]. Also it is shown that the reduction of the code complexity leads to significant reduction of the errors and maintainability efforts. The unit and integration test strategies have been developed following the McCabe structured testing methodology [3]. The methodology provides unique code coverage capacity [7]. Therefore, test and code coverage issues for embedded networking systems [10] are considered as well.

2 Two Networking Code-Analysis Projects

The structured testing methodology [3] and McCabe's IQ tools [6-10] have been used in the C-preprocessed code analyses of different internetworking systems. The first system (Project A) has been designed to support carrier networks. It provides both services of conventional Layer 2 switches [11, 12] and the routing and control services of Layer 3 devices [13-15]. The McCabe IQ tool [6-9] has been used to study the Project-A C-code (about 300,000 lines) on the protocol basis. The Cyclomatic Complexity, Essential Complexity, Module Design Complexity, System Design Complexity, and System Integration Complexity metrics have been applied for studying the complexity of a code module's decision structure, the quality of the code (unstructured code constructs), a module's decision structure, the amount of interaction between modules in the program, and the estimation of the number of integration tests necessary to guard against errors. Nine protocol-based sub-trees of the code (3400 modules written in the C programming language for BGP, DVMRP, Frame Relay, ISIS, IP, MOSPF, OSPF2, PIM, and PPP protocols) have been analyzed.

The second system (Project B) has been developed for providing different networking services [Internet Protocol over virtual private networks (IP-VPNs), Firewalls, Network Address Translations (NAT), IP Quality-of-Service (QoS), Web steering, and others] [11]. The complexity code analysis of the C-preprocessed code and comparative analyses of the code releases have been made by estimating the Risk Factor, Cyclomatic Complexity, Essential Complexity, Module/Function Design Complexity, Number-of-Lines of Code, Estimated Number of Possible Errors, and Number of Unreliable & Unmaintainable Functions using the McCabe IQ tool [6-9]. The code and test coverage procedures [7] have been developed and utilized in this project as well.

Major areas of the code structures recommended for further reviewing have been identified in the detailed code analyses. Program revisions were used to find the code areas with potential errors and to change a code design practice of the code designers.

3 McCabe's Structured Testing Methodology

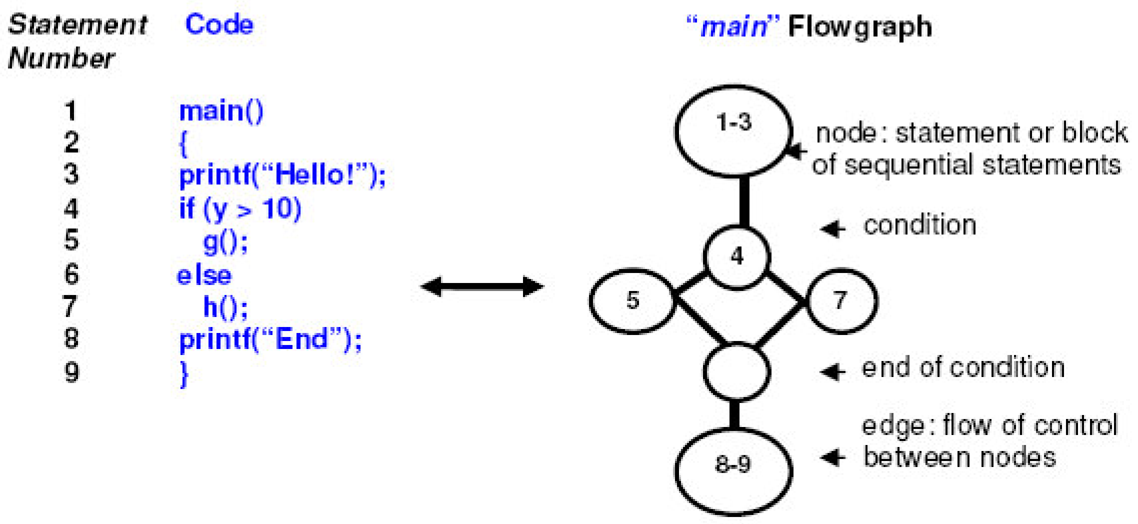
3.1 Methodology and McCabe QA Tools

The McCabe's methodology [3, 4] and McCabe QA tools [6-9] have been used to perform an analysis of codes for the projects A and B, which are described in the previous section. These enormous code structures can be effectively studied by the customized metrics [Cyclomatic Complexity (v), Essential Complexity (ev), Module Design Complexity (iv), System Design Complexity ($S0$), and System Integration Complexity ($S1$) metrics] [3, 4] to understand the level of complexity of a code module's decision structure, the quality of the code (unstructured code constructs), a module's design structure, the amount of interaction between modules in a program, and the estimation of the number of integration tests necessary to guard against errors.

3.2 Software Metrics Overview

The McCabe metrics are based on graph theory and mathematically rigorous analyses of the structure of software, which explicitly identify high-risk areas. The McCabe metrics are defined in Refs. 3, 4, 6-11.

For each module (a function or subroutine with a single entry point and a single exit point), an annotated source listing and flowgraph is generated as shown in Fig. 1. The flowgraph is an architectural diagram of a software module's logic.

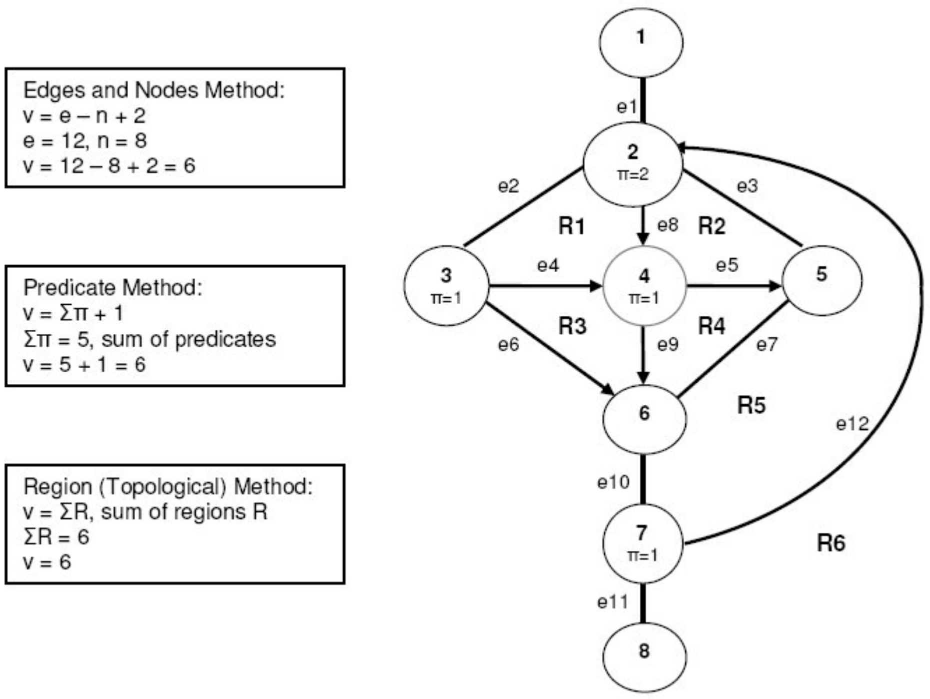


```
Statement   Code
Number

1           main()
2           {
3           printf("Hello!");
4           if (y > 10)
5             g();
6           else
7             h();
8           printf("End");
9           }
```

"main" Flowgraph
1-3
node: statement or block of sequential statements
condition
4
5
7
end of condition
edge: flow of control between nodes
8-9

Figure 1: The annotated source listing and the related flowgraph.



Edges and Nodes Method:
$v = e - n + 2$
$e = 12,\ n = 8$
$v = 12 - 8 + 2 = 6$

Predicate Method:
$v = \Sigma\pi + 1$
$\Sigma\pi = 5$, sum of predicates
$v = 5 + 1 = 6$

Region (Topological) Method:
$v = \Sigma R$, sum of regions R
$\Sigma R = 6$
$v = 6$

Figure 2: Three methods of evaluating the cyclomatic complexity of the graph.

Cyclomatic complexity, *v*, is a measure of the complexity of a module's decision structure [3, 4]. It is the number of linearly independent paths and, therefore, the minimum number of paths that should be tested to reasonably guard against errors. A high cyclomatic complexity indicates that the code may be of low quality and difficult to test and maintain. In addition, empirical studies have established a correlation between high cyclomatic complexity and error-prone software [1]. The results of experiments by Miller [5] suggest that modules approach zero defects when the McCabe's Cyclomatic Complexity is within 7 ± 2. Therefore, the threshold of *v*-metric is chosen as 10.

A node is the smallest unit of code in a program. Edges on a flowgraph represent the transfer of control from one node to another [3]. Given a module whose flowgraph has *e* edges and *n* nodes, its cyclomatic complexity is $v = e - n + 2$. This complexity parameter equals the number of topologically independent regions of the graph and correlates with the total number of logical predicates in the module [3, 4] (see Fig. 2).

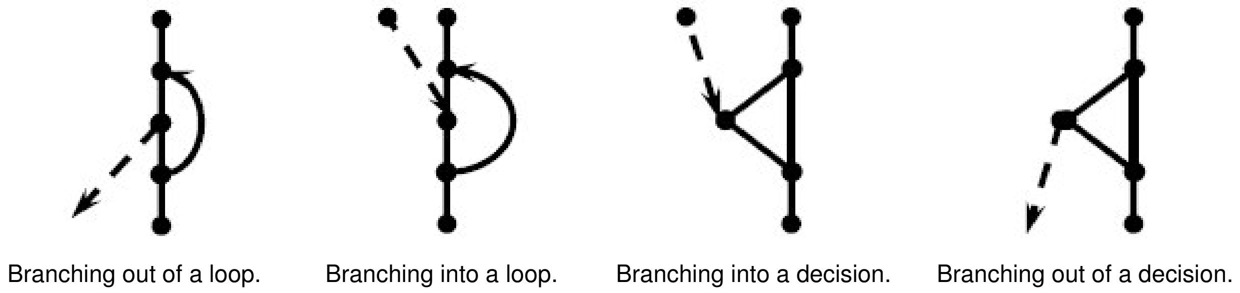


Figure 3: Examples of the unstructured logical constructs.

Essential complexity, *ev*, is a measure of unstructuredness, the degree to which a module contains unstructured constructs [3, 6] (see Fig. 3), which decrease the quality of the code and increase the effort required to maintain the code and break it into separate modules. When a number of unstructured constructs is high (essential complexity is high), modularization and maintenance is difficult. In fact, during maintenance, fixing a bug in one section often introduces an error elsewhere in the code [1, 4].

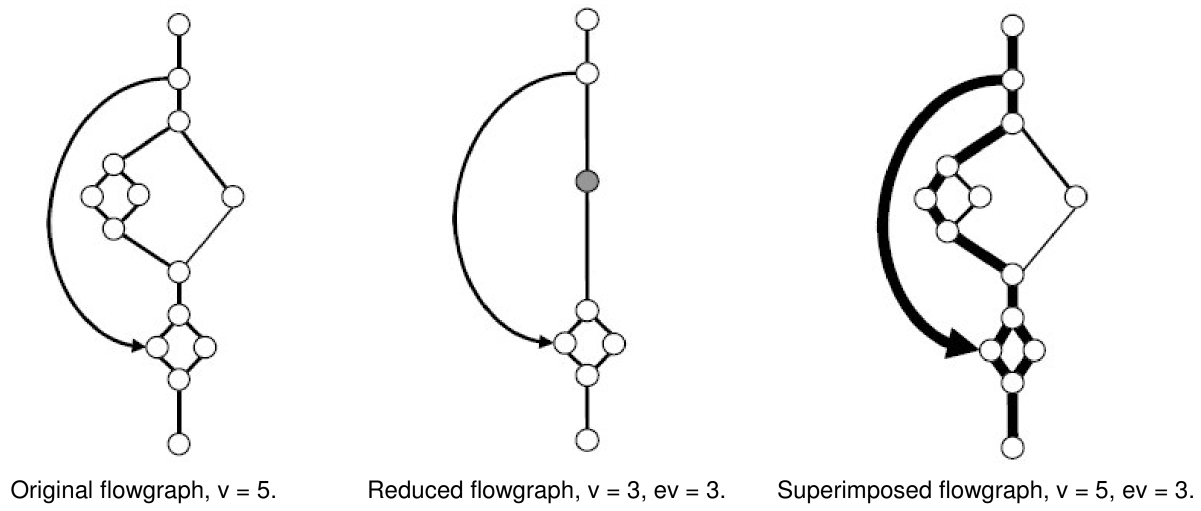

Original flowgraph, v = 5.
Reduced flowgraph, v = 3, ev = 3.
Superimposed flowgraph, v = 5, ev = 3.

Figure 4: Evaluation of the essential complexity of the flowgraph.

Essential complexity is calculated by removing all structured constructs from a module's flowgraph and then measuring the cyclomatic complexity of the reduced flowgraph [3, 4] shown in Fig. 4. The reduced flowgraph gives you a clear view of unstructured code. When essential complexity is 1, the module is fully structured. When essential complexity is greater than 1, but less than the cyclomatic complexity, the module is partly structured. When essential complexity equals cyclomatic complexity, the module is completely unstructured. The unstructured modules should be redesigned.

Module design complexity, *iv*, is a measure of a module's decision structure as it relates to calls to other modules [3, 4, 6]. This quantifies the testing effort of a module with respect to integration with subordinate modules. Software with high module design complexity tends to have a high degree of control coupling, which makes it difficult to isolate, maintain, and reuse software components.

To calculate the *iv*-metric, all decisions and loops that do not contain calls to subordinate modules are removed from the module's flowgraph [3, 6]. The module design complexity is the cyclomatic complexity of this reduced flowgraph and, therefore, of the module structure as it relates to those calls. Module design complexity can be no greater than the cyclomatic complexity of the original flowgraph and typically is much less. All decisions and loops that do not contain calls to subordinate modules should be removed. The original flow-graph is superimposed over the design-reduced flowgraph to show the decisions and loops that were removed.

System design complexity, S_0, measures the amount of interaction between modules in a program [3, 6]. It provides a summary of the module design complexity of the system components and measures the effort required for bottom-up integration testing. This metric also provides an overall measure of the size and complexity of a program's design, without reflecting the internal calculations of individual modules. Systems with high design complexity often have complex interactions between components and tend to be difficult to maintain.

The S_0 metric is calculated as the sum of the module design complexities of all modules in a program. It reveals the complexity of the module calls in a program.

Integration complexity, S_1, measures the number of integration tests necessary to guard against errors [3, 4, 6]. In other words, it is the number of linearly independent sub-trees in a program. A sub-tree is a sequence of calls and returns from a module to its descendant modules. Just as the cyclomatic complexity of a module defines the number of test paths in the required basis set for that module, integration complexity defines the number of linearly independent sub-tree tests in a basis set for a program or subsystem.

The S_1 metric quantifies the integration testing effort and represents the complexity of the system design. It is calculated by using a simple formula, $S_1 = S_0 - N + 1$, where N is the number of modules in the program. Modules with no decision logic do not contribute to S_1. This fact isolates system complexity from its total size.

The McCabe QA tool produces Halstead metrics [1, 2] for selected languages [6]. Supported by numerous industry studies [1], the Halstead's B-metric represents the estimated number of code errors.

3.3 Processing with the McCabe Tools

The procedures of the project processing with the McCabe tools are described in Refs. 6-9. In general, they can be divided into three groups at the Code Building level, Testing level, and Analysis level, as shown in Fig. 5.

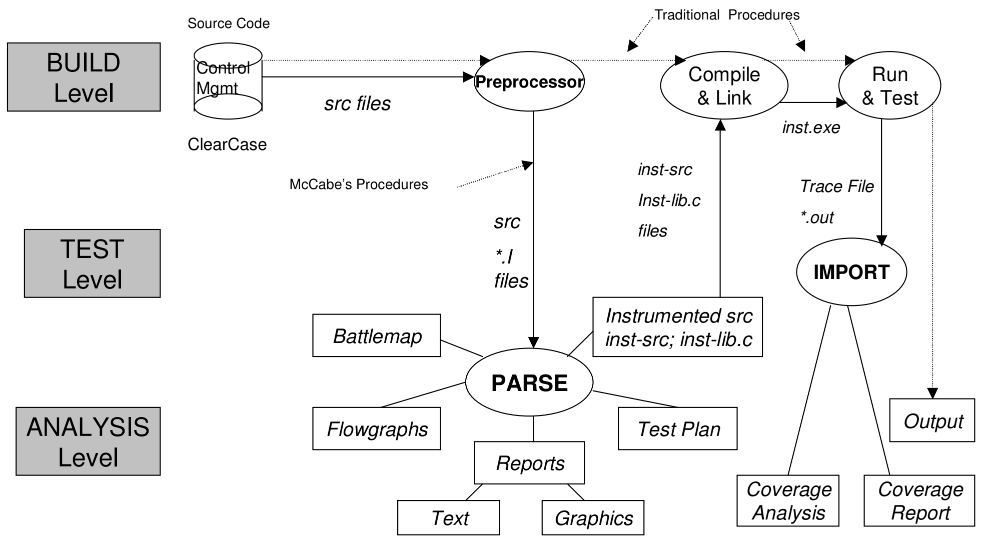


Figure 5: Procedures of the project code processing with the McCabe tools.

4 Results of the Project-A Code Analysis

Table 1: *v*-Cyclomatic and *ev*-Essential Cyclomatic Complexity Metrics for Project-A Nine-Protocol Code

Range	**BGP**	**DVMRP**	**FR**	**ISIS**	**IP**	**MOSPF**	**OSPF2**	**PIM**	**PPP**	**Total**
v = [1,10]	148	149	176	229	609	18	314	150	328	2121
[11,20]	62	54	38	42	205	9	101	66	110	687
[21,30]	34	20	8	24	58	2	45	35	35	261
[31,40]	11	12	4	4	32	1	21	15	13	113
[41,50]	11	9	4	3	16	-	16	6	9	74
[51,60]	6	6	1	-	7	-	7	6	7	40
[61,70]	5	4	1	-	7	-	3	7	4	31
[71,80]	5	2	-	1	3	1	2	3	2	19
[81,90]	2	2	1	-	1	-	2	4	3	15
[91,100]	2	-	-	-	1	-	1	1	1	6
[101,200]	5	2	-	-	6	-	2	6	4	25
[201,600]	1	2	-	-	1	-	-	-	4	8
Modules	292	262	233	303	946	31	514	299	520	3400
ev = [1,4]	137	111	158	197	470	13	273	167	255	1781
[5,10]	65	66	55	77	241	10	128	66	140	848
[11,20]	48	52	16	19	162	5	71	37	76	486
[21,30]	22	13	2	9	37	2	26	11	18	140
[31,40]	6	7	-	1	14	-	8	11	12	59
[41,50]	6	5	1	-	10	1	3	5	3	34
[51,60]	1	2	1	-	5	-	2	2	2	15
[61,70]	3	1	-	-	1	-	3	-	4	12
[71,80]	1	-	-	-	-	-	-	-	2	3
[81,90]	-	2	-	-	3	-	-	-	-	5
[91,100]	1	1	-	-	2	-	-	-	1	5
[101,200]	1	2	-	-	1	-	-	-	4	8
[201,600]	1	-	-	-	-	-	-	-	3	4
Unreliable&Unmaint.	129	112	50	70	292	13	190	109	182	1147
Estimated Errors	399	309	167	181	685	32	396	336	415	2920

4.1 Study of Cyclomatic Complexity (*v*)

In present study, the cyclomatic complexity metrics have been found for all 3400 modules (C-code preprocessed functions) related to nine protocols that have been reviewed in Project-A mentioned above in section 2. The results are shown in Table 1. It has been found that 38% of the code modules have the Cyclomatic Complexity more than 10 (including 592 functions (out of 3400) with the Cyclomatic Complexity more than 20). Only two protocol-based parts of the code (FR and ISIS) have relatively low *v*-metrics, namely, at least 76% of the code with $v \leq 10$.

4.2 Study of Essential Cyclomatic Complexity (*ev*)

The essential cyclomatic complexity metrics have been found for all 3400 modules related to nine protocols mentioned above. The results are shown in Table 1. It has been found that 48% of the code modules have the Essential Cyclomatic Complexity more than 4 (including 771 functions (out of 3400) with the Essential Cyclomatic Complexity more than 10). Only two protocol-based parts of the code (FR and ISIS) have relatively low *ev*-metrics, namely, at least 65% of the code with $ev \leq 4$.

4.3 Unreliable and Unmaintainable Code Modules Study

Using both metrics, Cyclomatic Complexity (*v*) and Essential Cyclomatic Complexity (*ev*), the code areas of reliability ($v \leq 10$) and maintainability ($ev \leq 4$) have been found. The areas have been identified from the scatter plots for each of nine protocols. The most unreliable and unmaintainable areas (at $v > 10$ and $ev > 4$) are shown in Table 1.

Totally 1147 modules (functions) are unreliable and unmaintainable, which represent 34% of the code. Following the definitions, when essential complexity is 1, the module is fully structured. When essential complexity is greater than 1, but less than the cyclomatic complexity, the module is partly structured. When essential complexity equals cyclomatic complexity, the module is completely unstructured. Among 3400 modules considered, 1447 modules (42%) are fully structured, 1453 modules (43%) are partly structured, and 500 modules (15%) are completely unstructured.

4.4 Study of Module Design Complexity (*iv*)

The module design complexity metrics were found for all 3400 modules related to nine protocols mentioned above. It has been found that 1066 code modules (functions) (31%) have the Module Design Complexity more than 5 [including 143 functions (out of 3400) with the Module Design Complexity more than 20]. Only four protocol-based parts of the code (FR, ISIS, IP, and PPP) have relatively low *iv*-metrics, namely, at least 71% of the code with $iv \leq 5$. In these four cases only 4 integration tests per module can be designed. BGP, MOSPF, and PIM have the worst characteristics (42% of the code modules require more than 7 integration tests per module).

4.5 Study of System Design Complexity (S_0) and System Integration Complexity (S_1)

The system design complexity metric was found for all nine protocols mentioned above. The protocol-based part of the code is characterized by the parameter of the System Design Complexity (S_0) of 19417, which is a top estimation of the number of unit tests that are required to fully test the release program. Also the code is characterized by the parameter of the System Integration Complexity (S_1) of 16026, which is a top estimation of the number of integration tests that are required to fully test the program.

4.6 Halstead B-Metric Studies

The Halstead B-metrics (possible errors) have been found for all 3400 modules related to nine protocols mentioned above. The results are shown in Table 1. It has been found that the Project-A code potentially contains 2920 errors estimated by the Halstead metrics approach [1]. Significant parts of the code (203 code modules, 6%) have the Number-of-Error B-Metric more than 3. Only five protocol-based parts of the code (FR, ISIS, IP, OSPF2, and PPP) have relatively low (significantly less than average error level of 0.86 per module) B-error metrics. In other four cases (BGP, DVMRP, MOSPF, and PIM), the error level is the highest one (more than one error per module).

4.7 Comparison of Two Customer Releases of Project-B: Redesign Efforts

Based on the detailed analysis of the Project-A code, we selected 271 modules of the old Customer Release A.1.2 and recommended them for redesigning by the software development team. After the re-engineering efforts, 16 old modules have been deleted and 7 new modules have been added for issuing the new Customer Release A.1.3. Analyzing the deleted modules, we found that 7 deleted modules were unreliable ($v > 10$) and 6 deleted modules were unmaintainable ($ev > 4$). Also, 19% of the deleted code was both unreliable and unmaintainable. These facts correlate well with our previous findings (see section 4.3). More, all seven new modules have been reliable and maintainable.

After redesigning, code changes resulted in the reduction of the cyclomatic code complexity by 115 units. 70 old modules (41% of the code) were improved, and only 12 modules (about 7% of the code) become worse. This analysis demonstrates a robustness of the structured testing methodology and mutual successful efforts of design and test engineers, which allow improving the quality of the Customer Releases.

5 Results of the Project-B Code Analysis

The McCabe's Structured Testing Methodology [3, 4] has been used in the complexity code analysis of all Code Releases in the Project B, as well as in the comparative study of the Releases. The data contains parameters of Risk Factor, Cyclomatic Complexity, Essential Complexity, Module/Function Design Complexity, Number of Lines-of-Code, Estimated Number of Possible Errors, and Number of Unreliable & Unmaintainable Functions for all 60 directories of the Project-B code (RMC/CMC platform). Here we discuss the major findings of the comparative study of two Releases B-4 vs. B-3.

5.1 Review of the Project B-4

The code directories have been divided into 7 groups (Embedded Management, OS/Tools, Platform, Protocols, Routing, Services, and Wireless). The distribution of the directories by the group membership is given in Table 2.

The analysis of Releases indicates that all directories can be ranged by the key evaluating parameter of the Risk Factor, which is based on average parameters of the Cyclomatic Complexity, Essential Complexity, Module/Function Design Complexity, Estimated Number of Possible Errors, and Number of Unreliable & Unmaintainable Functions (see Refs. 3, 4).

Table 2: The Numbers of the Modified Directories by Types of Functionality and Risk Factor Values

Type of Directory	Embedded Management	OS/Tools	Platform	Protocols	Routing	Services	Wireless	Total
GREEN-risk (original)	3	4	4	0	0	0	0	11
YELLOW-risk (original)	3	0	6	2	0	0	0	11
RED-risk (original)	3	3	12	11	2	5	2	38
Number of Directories	9	7	22	13	2	5	2	60
Number of Modified Directories	5	3	15	8	2	2	1	36
%% Modified Directories	56%	43%	68%	62%	100%	40%	50%	60%
GREEN-risk (modified	1	1	2	0	0	0	0	4 (36%)
YELLOW-risk (modified)	1	0	4	0	0	0	0	5 (45%)
RED-risk (modified)	3	2	9	8	2	2	1	27 (71%)

The Project-B code has a high level of the risk factor (RF = 1.843). The most part of the code (38 out of 60, or 63%) has the "RED" values of a risk factor ($RF > 1.5$). The "YELLOW" zone ($1.5 > RF > 1.0$) includes 18.5% of the used code, and the "GREEN" low-risk area ($RF < 1.0$) includes the rest 18.5% of the used code. It is an important fact that the directories related to Routing, Services, and Wireless functionality (15% of the total used code) have totally a high level of a risk factor (RED). The most part of the Protocol-functionality code (85%) is also characterized by a high level of a risk factor (RED). Only 18% of the Platform-functional code has a low level of a risk factor (GREEN), in contrast to 57% of the OS/Tools-functional software allocated in the same GREEN risk-factor zone.

The study covers 16,275 functions of the C-preprocessed code (860K lines) allocated in 979 files. The average parameters (per function) of the v-Cyclomatic Complexity Metric and the ev-Essential Complexity Metric are very high (v_{aver} = 10.54, ev_{aver} = 4.165), which indicates the inappropriate quality of the Project-B software system design. As a result, 4810 functions (30%) are unreliable ($v > 10$), and 3381 functions (21%) are both unmaintainable and unreliable ($ev > 4$ & $v > 10$).

The latest version of the code (Release B-4) contains 8,613 possible errors (1 error per 100 lines of the code, or 1 error per 2 functions at average). This estimation is based on the Halstead's methodology [1, 2], and represents the upper level of errors in badly designed logic-and-operator structures.

A large volume of unit-test and integration-test efforts should be provided and proper managed in this case. The upper-level estimation of the test efforts indicates that 96,721 independent logical paths should be analyzed in the unit testing, and 80,526 integration cases should be planned for testing.

5.2 Comparative Analysis of Two Releases (B-4 vs. B-3)

Changes have been made in 36 directories out of total 60 used directories (60%) of the Project B-4 code. The modified directories by types of functionality are shown in Table 2. The modification efforts are 50 % higher than in the previous Release B-3.

The changes in the Project-B code affected 36 directories (60%) mostly allocated in the RED highest risk zone (75% of all changes). Only 36% functions in the GREEN risk zone and 45% functions in the YELLOW risk zone have been modified. The details of this analysis are given in the Table 2.

A significant reduction of the risk factor has been achieved for functions from the RED zone in the Interprocess-Communication (Platform) directory by 5%, in the Address-Manager (Protocols) directory by 2%, in the Remote-Procedure-Call (Platform) directory by 3%, and in the Card-Manager (Platform) directory by 2%. Unfortunately, the risk factor of the whole code remains at the same level of 1.84 (RED) in the latest Release B-4. The latest fact indicates that no major code reconstruction efforts have been made at this stage of the Project-B.

Some functions become even more risky after modifications in the latest Release B-4. For example, the risk factor increased in the following directories: Interface-Manager (Platform) by 4%; ROUTING (Routing) by 3%; Card3-Driver (Platform) by 2%; Layer-2-Tunneling (Protocols) by 2%; Configuration-Manager (Platform) by 4%; Portal-Server (Services) by 3%; and Database Manager (OS/Tools) by 9% in the RED zone, and Interconnection-Service-Node (Platform) in the GREEN zone.

The Project B-4 code has been expanded by 8388 lines of C-preprocessed code. As a result of this code expansion, the Estimated Number of Possible Errors was increased by 115 errors, the Cyclomatic Complexity was increased by 1943 independent logical paths, the Essential Complexity (Unstructured Logic) was increased by 714, the Module Design Complexity was increased by 1082, which indicates the number of additional unit tests (1082) and integration tests (932).

The quality of changes is at the high level of confidence, which can be characterized by low increased Number of Unreliable & Unmaintainable Functions (63). Totally 116 new functions at low parameters of Cyclomatic Complexity and Essential Complexity have been added into the Release B-4.

Based on this analysis, it has been recommended to the Project-B software development team to concentrate their efforts on the code logical restructuring and reducing the Risk Factors of the modified code areas in the vital performance areas, which are valuable to the customers.

5.3 Protocol Based Analysis

Nine protocol-based areas of the code (2,447 modules written in 149,094 lines of code) have been analyzed, namely *BGP, FR, IGMP, IP, ISIS, OSPF, PPP, RIP,* and *SNMP* [12-15]. It has been found that 29% of the code modules have the cyclomatic complexity more than 10 (including 320 functions with $v > 20$). Only the Frame Relay (FR) part is well designed and programmed with few possible errors. Also, 39% of BGP, 31% of PPP and 30% of IP, OSPF, and RIP code areas are unreliable with $v > 10$. We found that 511 modules (19.4% of the protocol-based code) are both unreliable and unmaintainable ($v > 10$ and $ev > 4$), including 27% of the BGP, IP, and OSPF unreliable-and-unmaintainable code areas. The estimated number of possible errors in the protocol-based code is 1,473. Following the McCabe's approach of structured testing, 14,401 unit tests and 11,963 module integration tests have been developed to cover nine protocol-based selected areas of the Project-B code.

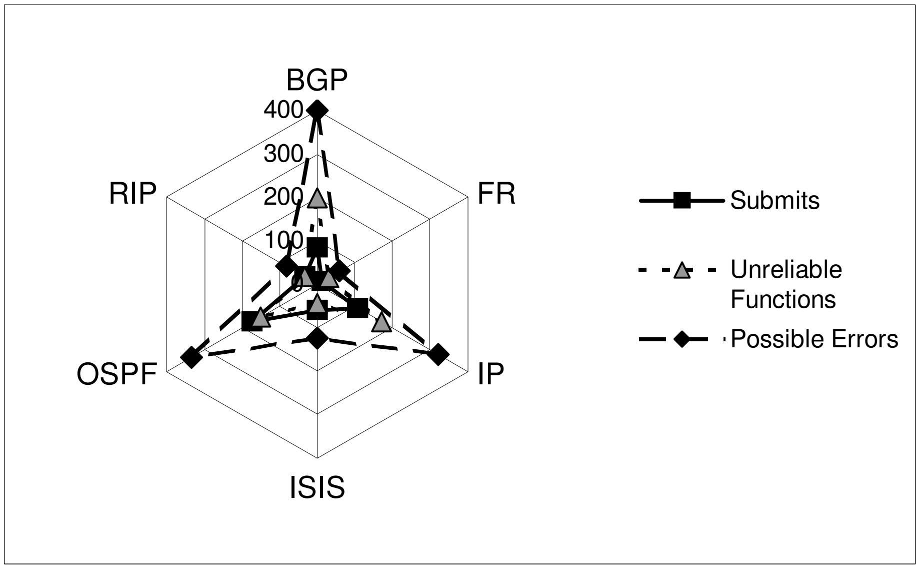


Figure 6: Correlation between the Number of Error Submits, Number of Unreliable Functions (v > 10), and the Number of Possible Errors for Six Protocols.

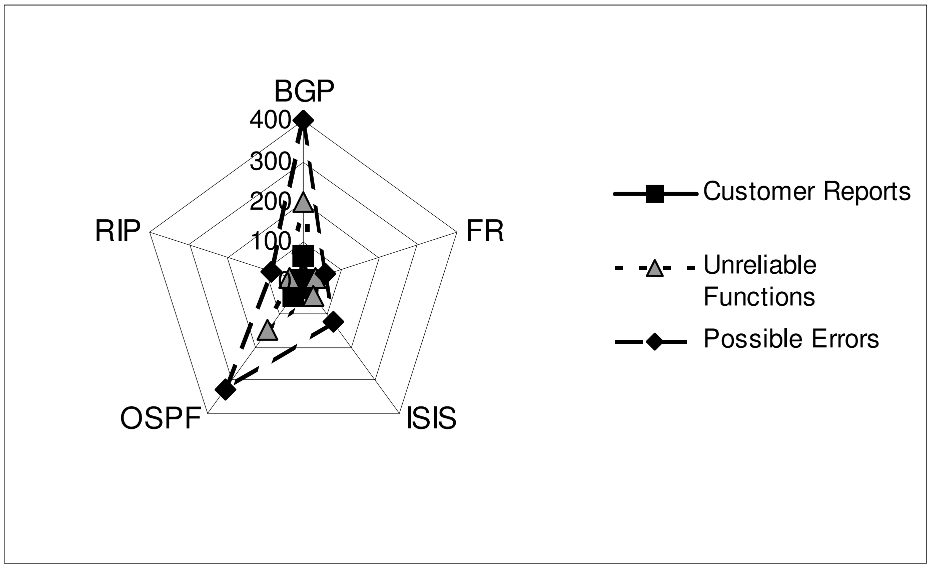


Figure 7: Correlation between the Number of Customer Error Reports, the Number of Unreliable Functions (v > 10), and the Number of Possible Errors for Five Protocols.

Studying the relationship between software defect corrections and cyclomatic complexity [16], we have found a great correlation between the numbers of possible errors, unreliable functions (with $v >$ 10), the error submits from Code Engineering Releases, and the Customer Error Reports (see Figs. 6 and 7 correspondingly).

6 Recommendations for Re-engineering Efforts

Based on reviewed information, several recommendations for teams of re-engineering network-services software (Projects A and B) have been developed:

- Reliable-and-maintainable modules ($v < 10$ and $ev < 4$) are the best candidates for re-using in the new versions of the Projects' products;
- Unreliable-and-unmaintainable modules ($v > 10$ and $ev > 4$) should be redesigned;
- Reliable-and-unmaintainable modules and unreliable-and-maintainable modules should be reviewed and tested;
- Future Unit & Integration Test plans can be developed using the McCabe's Independent Path techniques and Test & Code Coverage methodology [3, 6].

These efforts would allow improving the quality of the network-services software, significantly reducing a number of "bugs" and maintenance efforts, attract new customers, and, finally, increase company-marketing shares.

7 Conclusions

The knowledge of the graph theory and its applications in software engineering is beneficial for students with computer-science majors. The detailed analysis of the code complexity reveals areas of the code structure that should be revised. The code revision would allow to find the code areas with potential errors and to improve a code design and testing practice. Particularly, the provided analysis can be used in identifying error-prone software, measuring the optimum testing efforts, predicting the effort required to maintain the code and break it into separate modules, allocating possibly redundant code, indicating all inter-module control, and providing a fundamental basis for integration testing. The complexity code analysis and structured testing methodology should become a necessary attribute of software design, implementation, testing, sustaining, and re-engineering practice and training.

References

[1] Pressman, R., Software Engineering: A Practitioner's Approach, 6th ed., McGraw-Hill, 2005.

[2] Sommerville, I., Software Engineering, 8th ed., Addison-Wesley, 2006.

[3] Watson, A. H., and McCabe, T. J., Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication, No. 500-235, National Institute of Standards and Technology, Gaithersburg, MD, 1996.

[4] McCabe, T. J., A Complexity Measure, In: IEEE Transactions on Software Engineering, **2** (4), 308-320, 1976.

[5] Miller, G., The Magical Number of Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. In: Psychological Review, **31** (3), 128-134, 1956.

[6] Using McCabe QA, User's Manual, Version 7.0, Columbia, MD: McCabe & Associates, 1999.

[7] Using McCabe Test, User's Manual, Version 7.0, Columbia, MD: McCabe & Associates, 1999.

[8] Using McCabe C Parser, User's Manual, Version 7.0, Columbia, MD: McCabe & Associates, 1999.

[9] Using McCabe IQ Add-Ons, User's Guide, Version 7.0, Columbia, MD: McCabe & Associates, 1999.

[10] Testing Embedded Systems, Report No. 1027, Columbia, MD: McCabe & Associates, 1999.

[11] Riabov, V. V. Networking Software Studies with the Structured Testing Methodology. In: Computer Science and Information Systems, edited by P. Petratos and D. Michalopoulos. Athens Institute for Education and Research, Greece, 2005, pp. 261-276.

[12] Tanenbaum, A., Computer Networks, 4th ed., Prentice Hall, 2003.

[13] Peterson, L., and Davie, B., Computer Networks: A Systems Approach, 3rd ed., Morgan Kaufmann Publishers, 2004.

[14] Sheldon, T., McGraw-Hill Encyclopedia of Networking & Telecommunications, McGraw-Hill, 2001.

[15] Coombs, C., Jr., and Coombs, C. A., Communications Network Test & Measurement Handbook, McGraw-Hill, 1998.

[16] Heimann, D. Complexity and Defects in Software – A Case Study. In: Proceedings of the McCabe Users Group Conference, May 1994.

---

* **Dr. VLADIMIR V. RIABOV**, Associate Professor of Computer Science at Rivier College, teaches Computing Concepts & Tools, Software Engineering, Object-Oriented System Design, Networking Technologies, and System Simulation and Modeling. He received a Ph.D. in Applied Mathematics and Physics from Moscow Institute of Physics and Technology and M.S. in Computer Information Systems from Southern New Hampshire University. Vladimir published about 105 articles in encyclopedias, handbooks, journals, and international and national conference proceedings, including *the Journal of Spacecraft and Rockets*, *Journal of Aircraft*, *Journal of Thermophysics and Heat Transfer*, *The Internet Encyclopedia*, *The Handbook of Information Technologies*, *Proceedings of International Congress of Aeronautical Sciences*, *International Symposia on Rarefied Gas Dynamics*, *International Conference on Computer Science and Information Systems*, *Conferences of American Institute of Aeronautics and Astronautics*, and others. He is a member of ACM, AIAA, IEEE, and MAA.