Networking Software Studies with the Structured Testing Methodology

Vladimir V. Riabov
Rivier College, Department of Mathematics & Computer Science
420 South Main Street
Nashua, New Hampshire 03060, USA
E-mail: vriabov@rivier.edu

Abstract

The results of systematic software analyses with McCabe's and Halstead's metrics are presented for designing and testing three networking systems: the Carrier Internetworking switched routing solution, which allows managing the Internet-based virtual private networks over a multiservice asynchronous transfer mode infrastructure; Carrier Networks Support system that provides both services of conventional Layer-2 switches and the routing and control services of Layer-3 devices; and a system for providing different networking services (IP-VPNs, Firewalls, Network Address Translations, IP Quality-of-Service, and Web steering). The graph-based metrics (cyclomatic complexity, essential complexity, module design complexity, system design complexity, and system integration complexity) have been applied for studying the decision-structure complexity of code modules, code quality (unstructured logic), the amount of interaction between modules, and the estimated number of integration tests that are necessary to guard against errors. Nine protocol-based areas of the code (2,447 modules written in 149,094 lines of C-code) have been analyzed for BGP, Frame Relay, IGMP, IP, ISIS, OSPF, PPP, RIP, and SNMP networking protocols. It has been found that 511 modules (19.4% of the protocol-based code) are both unreliable and unmaintainable, including 27% of the BGP, IP, and OSPF code modules. Only the Frame Relay part of the code is well designed and programmed with a few possible errors. The number of unreliable code modules (29%) correlates well with the number of customer requests, error-fixing submits, and a number of possible errors (1,473) that have been estimated with the Halstead's metrics. Following the McCabe's approach of structured testing, 14,401 unit tests and 11,963 module integration tests have been developed to cover the protocol-based code areas. Comparing different Code Releases, it is shown that the reduction of the code complexity leads to significant reduction of the errors and maintainability efforts. The test and code coverage issues for embedded networking systems are also discussed.

1. Introduction

The McCabe's structured testing methodology [1] has become a widely used approach in the complexity code analysis, independent logical path testing, integration test planning, and test coverage efforts in different industries. Since 1996, this methodology becomes the standard, which was recommended by the National Institute of Standards and Technology [1]. The approach was developed by McCabe [2] 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 [3], McCabe suggests that the code modules approach zero defects when the cyclomatic complexity is less than 10.

Nowadays, the McCabe's QA tools [4-7] become available for software designers and test engineers [1]. Unfortunately, some companies (including some networking companies) are not familiar with the structured testing methodology and continue using "traditional" metrics (i.e., Reviewed-Lines-Of-Code [14, 15]) in their unit-and-integration testing practice. As a result, the quality of networking-service software and products is low, and testing, debugging, and sustaining efforts are tremendous.

In present study, the results of systematic analyses of networking-systems software with different metrics are presented for tree industrial projects A, B, and C. 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, 4]. 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 [1]. The methodology provides unique code coverage capacity [5]. Therefore, test and code coverage issues for embedded networking systems are considered as well.

2. Three Networking Code-Analysis Projects

The structured testing methodology [1], McCabe's IQ tools [4-8] and DISCOVER tool [9] have been used in the C-preprocessed code analyses of different internetworking systems. The first system (Project A) has been developed as a prototype of the Carrier Internetworking switched routing solution, which allows managing the Internet protocol (IP) virtual private networks (VPNs) environment over a multiservice asynchronous transfer mode infrastructure. More than 1.2 million lines of code (allocated in 1475 files) have been parsed and analyzed using the DISCOVER tool. The Cyclomatic Complexity and Nested Control Structures metrics have been applied for studying the complexity and quality of the code.

The second system (Project B) has been designed to support carrier networks. It provides both services of conventional Layer 2 switches [10] and the routing and control services of Layer 3 devices [11-13]. The McCabe IQ tool [4, 6, 7] has been used to study the Project-B 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 subtrees 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 third system (Project C) has been developed for providing different networking services (IP-VPNs, Firewalls, Network Address Translations (NAT), IP Quality-of-Service (QoS), Web steering, and others) [13]. 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 [4, 6, 7]. The code and test coverage procedures [5] have been developed and utilized in this project as well.

The detailed analyses of the codes allow to identify major areas of the code structures to be reviewed. The code revisions help 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 [1, 2] and McCabe QA tools [4-7] have been used to perform an analysis of codes for the projects A, B, and C, 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) [1, 2] 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 1, 3, 4-7.

Cyclomatic complexity, *v*, is a measure of the complexity of a module's decision structure [1, 2]. 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 [14]. The results of experiments by Miller [3] 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 [1]. Given a module whose flowgraph has *e* edges and *n* nodes, its cyclomatic complexity is $v = e - n + 2$.

Essential complexity, *ev*, is a measure of unstructuredness, the degree to which a module contains unstructured constructs [1, 4], 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.

Essential complexity is calculated by removing all structured constructs from a module's flowgraph and then measuring the cyclomatic complexity of the reduced flowgraph [1, 2]. 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 partly and completely unstructured modules should be recommended for redesigning.

Module design complexity, *iv*, is a measure of a module's decision structure as it relates to calls to other modules [1, 2, 4]. 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 individual 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 [1, 4]. 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, *S0*, measures the amount of interaction between modules in a program [1, 4]. 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 *S0* 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, *S1*, measures the number of integration tests necessary to guard against errors [1, 2, 4]. In other words, it is the number of linearly independent sub-trees in a program. A subtree 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 subtree tests in a basis set for a program.

The *S1* metric quantifies the integration testing effort and represents the complexity of the system design. It is calculated by using a simple formula, $S1 = S0 - N + 1$, where N is the number of modules in the program. Modules with no decision logic do not contribute to *S1*. This fact isolates system complexity from its total size.

The McCabe QA tool produces Halstead metrics [14, 15] for selected languages [4]. Supported by numerous industry studies [14], the B-metric of Halstead represents the estimated number of errors in the program.

3.3 Processing with the McCabe Tools

The procedures of the project processing with the McCabe tools are described in Refs. 4-7. In general, they can be divided into three groups at the Code Building level, Testing level, and Analysis level (see Fig. 1 for details).

4. Results of the Project-A Code Analysis

4.1 Project-A Code Review With DISCOVER Tool

The DISCOVER tool [9] has been used to perform an analysis of the Project-A code. It has been found that the code contains 6970 functions (defined in 781 files),

3410 variables, and 1652 classes/structures. This enormous code structure has been studied by 14 metrics [9] to understand the level of complexity (v-metric) and the maximum depth of nested control structures (DEPTH-metric).

4.2 Cyclomatic Complexity Metric Analysis of the Project-A code

In present study, the cyclomatic complexity v-metric has been evaluated by the DISCOVER v-Metrics queries for all 6970 functions of the Project-A code. Almost 14% of the code functions have the cyclomatic complexity more than 10 (including 282 functions with the cyclomatic complexity more than 20). All files, which contain functions with the cyclomatic complexity more than 50 (28 functions), should be reevaluated. They are concentrated in five subdirectories (SNMP protocol, Database Management, Network Interface Card, and other management utilities).

Unfortunately, the DISCOVER tool allows to estimate only the module/function complexity, not the system as a whole. The other software tools of studying the system complexity [4] can be recommended in this case. The v-metrics have to be calculated at earlier stages of the software development life cycle [14, 15]. The complexity code analysis would identify areas of possible error concentration and test strategies. Unfortunately, the Cyclomatic Complexity algorithms place the same weight on nested and non-nested loops. It is a well-known fact [1, 14, 15] that deeply nested conditional structures are harder to understand and modify than non-nested structures. Therefore, the Nested Control Structures (DEPTH) metrics has been applied for studying the scope of the code.

4.3 Nested Control Structures (DEPTH) metrics

The maximum depth of nested control structures in all 6970 functions has been studied using the DISCOVER DEPTH metric. The study results show that maximum number (2619) of functions (38%) has normal depth (1) of nested control structures. At the same time, the code has 413 functions with the parameter of DEPTH bigger than 3 (including 25 Nested Control Structures with DEPTH bigger than 6).

5. Results of the Project-B Code Analysis

5.1 Study of Cyclomatic Complexity (v)

In present study, the cyclomatic complexity metrics have been found for all 3400 modules (C-preprocessed functions) related to nine protocols that has been reviewed in Project-B mentioned above. 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 $v \leq 10$.

5.2 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 $ev \leq 4$.

5.3 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.

5.4 Study of Module Design Complexity (iv)

The module design complexity metrics have been found for all 3400 modules related to nine protocols mentioned above. It is found that 1066 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 cases only 4 integration tests per module can be designed. BGP, MOSPF, and PIM have the worst characteristics (more than 42% of the modules require more than 7 integration tests per module).

5.5 Study of System Design Complexity (S0) and System Integration Complexity (S1)

The system design complexity metric has been found for all nine protocols mentioned above. The protocol-based part of the code is characterized by the parameter of the System Design Complexity (*S0*) 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 (*S1*) of 16026, which is a top estimation of the number of integration tests that are required to fully test the release program.

5.6 Halstead B-Metrics Study

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-B code potentially contains 2920 errors estimated by the Halstead metrics approach [4]. 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).

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

Based on the detailed analysis of the Project-B code, we selected 271 modules of the old Customer Release B.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 B.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 5.3). 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.

6. Results of the Project-C Code Analysis

The McCabe Structured Testing Methodology [1, 2] has been used in the complexity code analysis of all Code Releases in the Project C, 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-C code (RMC/CMC platform). Here we discuss the major findings of the comparative study of two Releases C-4 vs. C-3.

6.1 Review of the Project C-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. 1, 2).

The Project-C 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 a very 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-C 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 C-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 [4, 14, 15], 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.

6.2 Comparative Analysis of Two Releases (C-4 vs. C-3)

Changes have been made in 36 directories out of total 60 used directories (60%) of the Project C-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 C-3.

The changes in the Project-C 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 C-4. The latest fact indicates that no major code reconstruction efforts have been made at this stage of the Project-C.

Some functions become even more risky after modifications in the latest Release C-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 C-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 very low parameters of Cyclomatic Complexity and Essential Complexity have been added into the ReleaseC-4.

Based on this analysis, it has been recommended to the Project-C 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.

6.3 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*. 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 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-C code.

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 $v > 10$), error submits from Code Engineering Releases and Customer Error Reports (see Figs. 2 and 3 correspondingly).

7. Recommendations for Re-engineering Efforts

Based on reviewed information, several recommendations for teams of re-engineering network-services software (Projects A, B, and C) 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 [1, 4].

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.

8. Conclusion

The detailed analysis of the code identifies major areas of the code structure to be reviewed. 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 an identification of error-prone software, measuring the minimum testing effort and revealing areas of concentration for testing, 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, and testing, sustaining, and re-engineering practice in networking industry.

References

1. 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, pp. 1-113.
2. McCabe, T. J., *A Complexity Measure*, IEEE Transactions on Software Engineering, Vol. 2, No. 4, Dec. 1976, pp. 308-320.
3. Miller, G., *The Magical Number of Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*, The Psychological Review, March 1956.
4. *Using McCabe QA,* User's Manual, Version 7.0, McCabe & Associates, Columbia, MD, 1999.
5. *Using McCabe Test,* User's Manual, Version 7.0, McCabe & Associates, Columbia, MD, 1999.
6. *Using McCabe C Parser*, User's Manual, Version 7.0, McCabe & Associates, Columbia, MD, 1999.
7. *Using McCabe IQ Add-Ons*, User's Guide, Version 7.0, McCabe & Associates, Columbia, MD, 1999.
8. *Testing Embedded Systems*, Report No. 1027, McCabe & Associates, Columbia, MD, 1999, pp. 1-2.
9. *DISCOVER User Guide*, Release 7.0 for SunOS, Solaris, HP-UX, and IRIX, Software Emancipation Technology, Inc., 1999.
10. Tanenbaum, A., *Computer Networks*, 4th edition, Prentice Hall, 2003.
11. Peterson, Larry, and Davie, Bruce, *Computer Networks: A Systems Approach*, 3d edition, Morgan Kaufmann Publishers, 2004.
12. Sheldon, T., *McGraw-Hill Encyclopedia of Networking & Telecommunications*, McGraw-Hill, 2001.
13. Coombs, C., Jr., and Coombs, C. A., *Communications Network Test & Measurement Handbook*, McGraw-Hill, 1998.
14. Pressman, Roger, *Software Engineering: A Practitioner's Approach*, 6th edition, McGraw-Hill, 2005.
15. Sommerville, Ian, *Software Engineering*, 7th edition, Addison-Wesley, 2004.
16. Heimann, D., *Complexity and Defects in Software – A Case Study*, Proceedings of the McCabe Users Group Conference, May 1994.

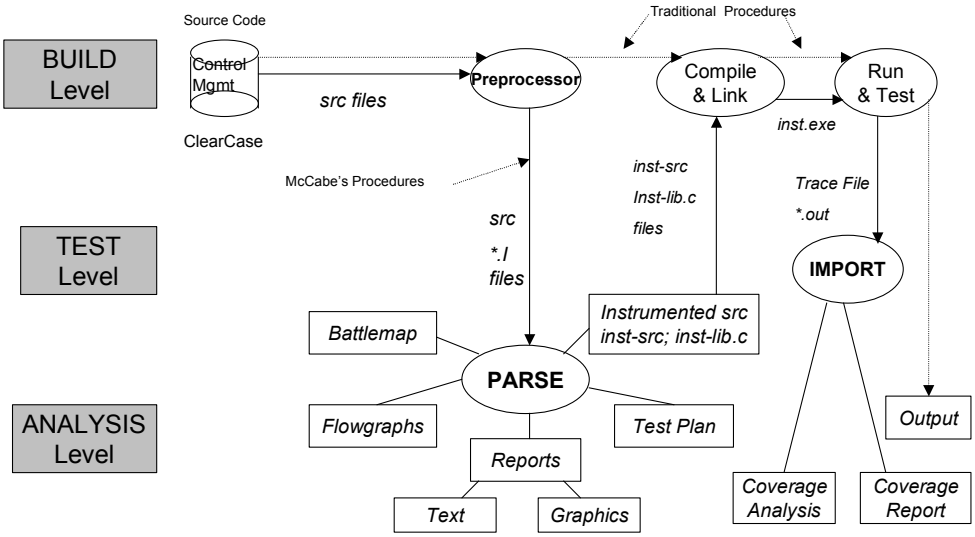

Figure 1 *Procedures of the project code processing with the McCabe tools*

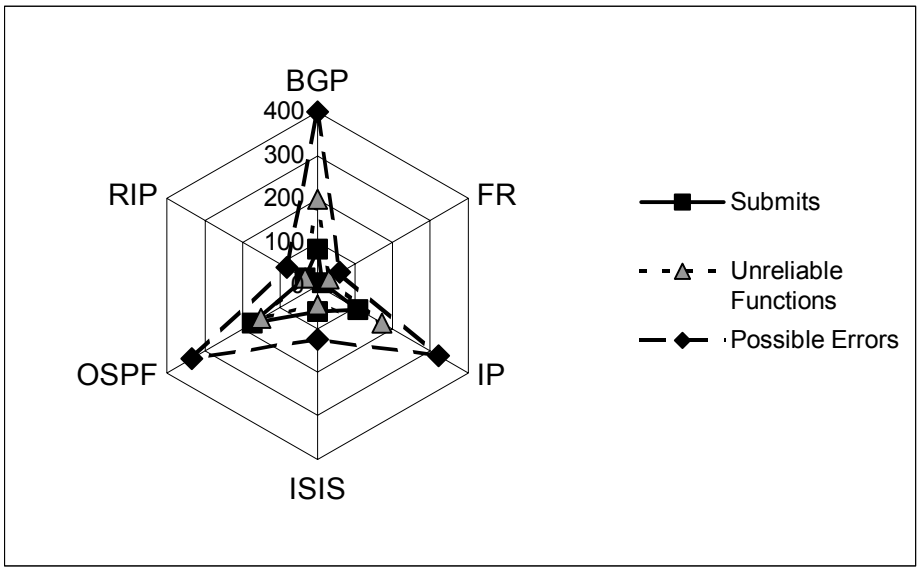

Figure 2 *Correlation between the Number of Error Submits, Number of Unreliable Functions (v > 10), and the Number of Possible Errors for Six Protocols*

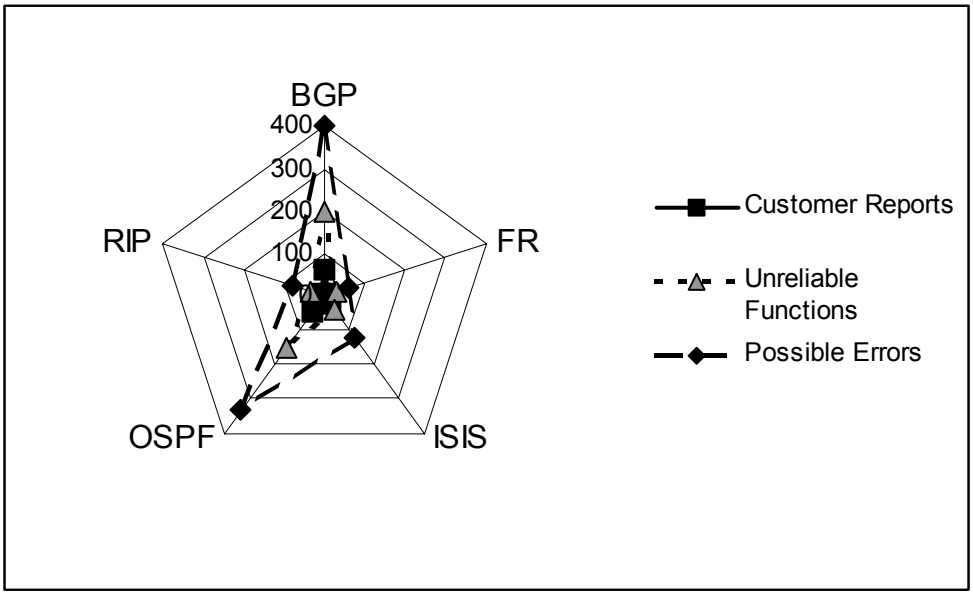

Figure 3 *Correlation between the Number of Customer Error Reports, the Number of Unreliable Functions (v > 10), and the Number of Possible Errors for Five Protocols*

Table 1 *v-Cyclomatic Complexity and ev-Essential Cyclomatic Complexity Metrics for Project-B 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
[201,300]	1	2	-	-	1	-	-	-	3	7
[301,600]	-	-	-	-	-	-	-	-	1	1
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
[201,300]	1	-	-	-	-	-	-	-	2	3
[301,600]	-	-	-	-	-	-	-	-	1	1
Unreliable &Unmaint	129	112	50	70	292	13	190	109	182	1147
Estimated Errors	399	309	167	181	685	32	396	336	415	2920

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 (orig.)	3	4	4	0	0	0	0	11
YELLOW-risk (orig.)	3	0	6	2	0	0	0	11
RED-risk (original)	3	3	12	11	2	5	2	38
Total 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(modif.)	1	1	2	0	0	0	0	4 (36**%**)
YELLOW-risk(modif.)	1	0	4	0	0	0	0	5 (45**%**)
RED-risk (modified)	3	2	9	8	2	2	1	27 (71**%**)