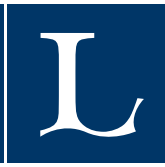
MASTER'S THESIS



Automatic Measurement of Source Code Complexity

Hassan Raza Bhatti

Master of Science
Computer Science and Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

ABSTRACT

The aim of this master thesis is to explore the area of software metrics and to identify software metrics related to the code complexity. In this thesis, thorough study is made to determine whether or not the automatic measurement of source code complexity is possible. A tool for automatic measurement of source code complexity is implemented during this thesis to prove the idea that the automatic measurement is achievable.

This report summaries the theory about software metrics, purpose and classification of the metrics, and the areas where metrics can be helpful to use. Detail description about some selected metrics (like Cyclomatic Complexity and Halstead metrics) is also a part of this report.

Three core requirements of this thesis are: 1) Measurement of code complexity for the code written in C. 2) Measurement should perform automatically on the code base and on a regular basis for new code releases. 3) Run on Solaris. Some of the existing complexity measurement tools (open-source and commercial) are evaluated in this thesis. QA-C is an existing commercially available tool for the code complexity of C code. The tool implemented in this thesis uses QA-C as a foundation for analyzing C code on Solaris. Web interfaces are designed to present the results of code complexity measurement.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

This thesis is the final milestone towards the completion of my master degree in Computer Science and Engineering from Luleå University of Technology, Sweden. My heartfelt praises are for my ALLAH, who is my creator, who is most merciful and compassionate and whose benevolence is the real reason for the successful completion of this thesis.

I am grateful to Tieto Sweden AB for providing me the opportunity to work in their esteem organization. The work environment and all kind of support provided by Tieto for this thesis are highly appreciated.

I owe my deepest gratitude to my supervisor/manager in Tieto, Maria Lockman. Her extraordinary cooperation and support made this work much easier and pleasant.

It's a pleasure to thank my technical supervisor Axelsson Bengt, who never felt tired to help in explaining all technical difficulties. His experience and knowledge made the harder problems simpler to understand and solve. In addition I would also like to thank Sara Berg as well as Lindblom Jan from Tieto. They were friendly and helpful.

I would like to express my humble gratitude to a Manager of the client company in Stockholm for initiating the idea of this research. His interest in this thesis and his appreciation for new ideas was the source of real motivation.

I am thankful to my examiner/supervisor, Jingsen Chen from Luleå University of Technology, for his guidance and encouragement.

My special thanks are for my family, particularly to my loving parents and brother. Living away from them and completing this huge task wouldn't have been possible without their prayers and moral support. I am also thankful to my friends who are living around me; they are the source for my happiness and to keep myself focused on my tasks.

Last but not least, bundle of thanks to my thesis partner Petter Berglund from Lund University of Technology. It was a delightful experience of working together with him on this thesis as he is not only technically competent but also very cooperative and friendly person.

Hassan Raza Bhatti

January 2010

Chapter 1

INTRODUCTION

This thesis is performed at the final stage of my master degree in Computer Science and Engineering from Luleå University of Technology, Sweden. In this thesis a software tool is developed for achieving the "Automatic Measurement of Source Code Complexity". All the work on this thesis is completed in the premises of Tieto Sweden AB in Luleå during the summer and autumn period of 2010.

1.1 Background

The idea of this thesis was originated by a telecom company in Sweden. The motive behind this idea was that the company wanted to investigate the possibility of automatic measurement of source code complexity of their entire system or subsystem. The company also wanted to investigate if complexity measurement can give any indication about the parts of code that have changed among different releases of same software.

As far as my personal motivation of choosing this thesis is concerned, I had certain preferences for my thesis. I wanted a thesis where I could explore something new for the enhancement of my skills. I didn't just want theoretical enhancement in my knowledge, but also wanted to practically implement what I learn from the theory. It was also my preference to work in some renowned organization during my thesis so that I could work with professionals from the IT industry. This thesis perfectly matched with the preferences about my thesis choice. Tieto offered this thesis on a demand of one of their largest client. Therefore it provided me the opportunity to work on a project which was essential for the client company.

1.2 Problem Area

The given problem for this thesis has two broader parts. The first part, research, is about the study and exploration of software metrics. The second part is to implement a solution for achieving automatic code complexity measurement using software metrics.

1.2.1 Research

A method for automatic measuring of source code complexity of a system is determined in this research. Software metrics related to code complexity are investigated and a suitable

set of metrics is identified for the given system. Existing solutions or tools for code complexity measurement are analyzed in this thesis.

The research also took into account how a code complexity metric correlates with other kind of metrics on development and maintenance work in order to find the impact of code complexity on for example maintainability of code base etc.

The specific requirements, for deploying the researched method using an existing complexity analysis tool, are also determined.

1.2.2 Implementation

After a suitable method was identified and proven suitable, that method is implemented as a prototype which can be integrated with existing automated build environment.

The code complexity measurement is performed using a commercially available code analysis tool QA-C and the functionality is extended to make automatic measurements possible and integrate-able with current build environment.

A database is used to store the results of gathered metrics for later retrieval and interpretation. Web interfaces are designed and implemented to review the results of gathered metrics and to visualize comparison among various releases of source code.

1.3 The Given System

The code measured in this thesis is a code written in C programming language. The code consists of many packages of software, and source code files are in thousands. All the code is maintained in Rational ClearCase on Solaris machines. The code analysis (for testing purpose) is performed on two sufficiently large packages of software.

1.4 The Method

The way adopted to perform this thesis can be divided into two main parts.

1.4.1 Pre-study

- Doing research on software metrics through appropriate literature to understand this area of software engineering.
- Identifying code metrics for complexity measurement.
- Presenting the pre-study conclusions to the client company.
- Investigating the existing methods/tools for complexity analysis.
- Understanding the automatic build environment and technologies being used by the company and determining the needs for automatic complexity measurement.

1.4.2 Implementation of tool

- Learning the languages and technologies in which the tool is decided to be developed.
- Designing web-interfaces for presenting the results of code analysis.
- Presenting non functional demo to the client company and deciding future tasks
- Implementing the tool and developing web-interfaces.
- Finalizing the prototype tool for automatic complexity measurement.

1.5 Delimitation

The analysis of results highlighted great contradiction between recommended maximum metric values by QA-C and database averages; however this problem was only found in Halstead metrics. This problem was identified during the analysis of result. At that time the thesis was at final stage of its completion, therefore the reason identified for this major contradiction couldn't be truly verified because of time constraints.

Due to the time bound evaluation copy of QA-C, only two software packages were analyzed. It could have been better to evaluate more packages of source code.

Comparison for various versions of packages, files and functions is implemented. However the comparison at releases/builds level isn’t implemented as it required little more time.

Chapter 2

SOFTWARE METRICS

A wide range of activities is associated with different phases of software development. Software metrics are techniques/formulas to measure some specific property or characteristics of software. In software engineering, the term 'software metrics' is directly related to the measurement. Software measurement has significant role in the software management. According to DeMarco; "You can't manage what you can't measure!" [1] Campbell also emphasized the importance of measurement in software management by stating, "If you ain't measurin,' you ain't managin' — you're only along for the ride (downhill)!" [2].

At this point it is worth to define 'measurement' itself. Norman Fenton defines measurement as the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules [3].

Software metrics are a quantitative guide to the performance of a certain piece of software in relation to the human interactions needed to make the software work. Metrics have been established under the idea that before something can be measured or quantified, it needs to be translated into numbers. There are several areas where software metrics are found to be of use. These areas include everything from software planning to the steps that are meant to improve the performance of certain software. Software cannot perform on its own without human interaction. Therefore, in a way, software metric is also a measure of a person's relation to the software that he or she is handling [4].

2.1 Uses of Software Metrics

Software metrics can apply on each measurable entities of software. These metrics are applicable throughout the software development life cycle. In start of a software project, software metrics can be developed, for example, for determining cost estimation and resource requirements. During the design phase metrics can be used to count function points of the software. Measurement of software size is another quantifiable aspect. Metrics gathered from source code can be used to measure the software size. An example of such metric is lines of code metric, which is the most basic metric for measuring software size. Similarly, metrics gathered from detailed design or from control structure can guide the software testing process. Code analysis can help in improving the software maintenance cost.

Software metrics enable software developers to analyze their code and make improvements if required. For example, if a certain part of their code is exceeding code complexity limits,

they can redesign it. Effective measurement can serve fundamental management objectives, such as cost and resource estimation, assess progress and future predictions. Software quality measurement is also possible through metrics.

Metrics can be developed for:

- Software size
- Cost estimations
- Effort estimations
- Software quality
- Maintainability
- Reliability models
- Complexity analysis
- Defect analysis
- Software testing

2.2 Classification of Software Metrics

In term of different measures during the different phases of software, metrics can be grouped as:

1. **Process metrics** - measures of software development process
2. **Product metrics** - measures of software product attributes
3. **Resource metrics** - measures of software resources

2.2.1 Process metrics

Software metrics relating the measurement of software process attributes are grouped as process metrics. Such measurements include, for example, duration estimate, cost assessment, effort required, process quality and effectiveness/efficiency of the development process. Depending upon the desired measures, these metrics can be calculated at any stage of the software development.

One example of process metrics is Source Line of Code (SLOC) metric. This metric simply count the lines of source code. By this count it can give an indication of effort made to develop that code.

2.2.2 Product metrics

Measurement of product doesn't only include the deliverable product to the customer, but it also includes all the activities made during the development process, for example documentation and prototypes.

A wide range of metrics exists for the measurement of software product. Product metrics are either the measure of product's external attributes or its internal attributes.

External attributes are the measure of product performance in the actual environment where the product has to run. These measures include software usability and re-usability, portability and efficiency.

Example of internal product attributes are: size of the software, correctness, complexity, bugs and testability.

2.2.3 Resource metrics

These metrics are more relevant to the managers for the estimation of resources needed for software project. These resources are man-powers (developers), physical resources, for example computers, material and methods. These metrics can also be classified under process metrics class.

2.3 Software Code Metrics

Software metrics that are directly countable from source code are referred as code metrics. Metrics of this category usually belong to the product metrics class. Majority of the known metrics at present belongs to the sub class of code metrics.

Code metrics provide developers better insight about the code which they are developing. By taking advantage of code metrics, developers can understand which types and/or methods should be reworked or more thoroughly tested. Development teams can identify potential risks, understand the current state of a project, and track progress during software development [11].

On the basis of different measures performed on code, these metrics can be further categorized into the following subgroups:

- Quantitative metrics
- Metrics of program flow control complexity
- Metrics of data control flow's complexity
- Metrics of control flow and data flow complexity
- Object-oriented metrics
- Safety metrics
- Hybrid metrics

2.3.1 Quantitative metrics

Metrics of this group take in account the quantitative characteristics of code. Quantitative characteristics of code can be the measure of size, count of total number of functions of a program, number of operations or operands in a program, number of comments and average lines per function.

Examples of quantitative metrics are: SLOC, Halstead Software Science, ABC metric and Jilb's metrics.

2.3.2 Metrics of program flow complexity

Quantitative metrics ignore the program flow structure. Analysis of program flow structure or control graph of a program gives the complexity of program flow. These metrics measure program complexity on the basis of analysis made on flow control structure of the program. These metrics can guide the number of required test cases.

First metric of this kind was introduced by McCabe in 1976, known as McCabe cyclomatic complexity. McCabe cyclomatic complexity is still one of the most commonly used metric in many commercial and non-commercial tools for code complexity measurement. More metrics based of this category are G. Mayers methodology, Hansen's method, Chen topological measure, Harrison's metrics and Mayjeal's metrics, Woodward's measure, Boundary value method and Shneidwind's metric [5].

2.3.3 Metrics of data control flow's complexity

This group of metrics is based on the analysis of program's data control flow. These metrics are suitable for data driven programs. An example of this type of metrics is Chepin's metric. This method examines the way variables are used in a separate program module and gives an estimate about the information strength of the module.

Weighting ratio is another metric from the same group which is used to measure the program complexity. Henry and Kafura information flow metric is also a measure of complexity based upon the information flow connections between a procedure and its environment [5].

2.3.4 Metrics of control flow and data flow complexity

Metrics of this group are used to measure the complexity on the basis of quantitative properties and analysis of control structures (program flow and data flow). This class of metrics and metrics of program flow are also categorized under the class of topological metrics [5]. Module cohesion, Testing M-measure, Kolofello's measure, McCloore's metric are some examples of this type of metrics.

2.3.5 Object-oriented metrics

The era of software metrics started before the concept of object-oriented programming. Difference in coding style of object-oriented programming compared to the structured programming endorsed the need of new class of metrics that can give more effective complexity measures for object-oriented languages. The most frequently used metrics of this group are set of Martin's metrics and Chidamber and Kamerer's metrics [5].

2.3.6 Safety metrics

These metrics have resemblance with quantitative metrics as these are quantitative measure of errors and defects in a program, for example the number of errors detected in code review or during testing. Average number of errors per thousand code lines is the most frequently used metric of this group.

2.3.7 Hybrid metrics

Combination of simpler metrics and their weighted total yields another class of metrics which is known as hybrid metrics. Depending upon the desired purpose to use software metrics, different metrics can be applied on a program and a weighting criteria can be developed that can help in achieving the purpose. The first metric introduced of this type is Cocol's metric. Another example of hybrid metrics is Zolnovskiy, Simmons and Tayer's metric, which is also a weighted total of different indices.

2.4 Code Complexity

The term “code complexity” is being frequently used in this report. It is also one of the emphasis of this thesis is the measurement of code complexity. Hence it is appropriate to understand the meaning of code complexity. K Magel highlights 6 different aspects related to the code complexity which implicitly defines the code complexity itself [19]:

1. The difficulty of understanding the program.
2. The difficulty of correcting the defects and maintaining the code.
3. The difficulty of explaining the code to other people.

4. The difficulty of updating the program according to some assigned rules.
5. The work load of writing programs according to the design.
6. The availability of necessary resources when programs are executing.

Code complexity often is the main factor which leads to defects, maintainability issues and portability of a software. Any software metric can be referred as complexity metric if it can measure one or more code complexity factors described earlier.

Chapter 3

SELECTION OF METRICS

The goal of this thesis (research) is to examine and implement a way in which automatic measurement of source code complexity is possible to perform. One part of this research is to find suitable metric or a set of metrics that can indicate the complexity of a system. Therefore, large number of software code metrics has been studied. Since the measurement of code complexity is to be performed on code written in C language, hence, object-oriented metrics aren't considered. Safety metrics are also excluded from the research as errors measurement is beyond the scope of this research. Code metrics from all other subgroups (discussed in previous chapter) have been studied to find suitable metric or combination of metrics that can give effective and useful complexity measurement.

This chapter describes the metrics which were found suitable for the needs of this project. Pros and cons of each of the selected metrics are explained and discussion is made about the motivation of selecting each metric. Section 3.8 discuses about using multiple metrics for a comprehensive code analysis. Section 3.8.1 proposes a set of metrics for a comprehensive code complexity measurement/analysis.

3.1 Source Line of Code (SLOC)

This metric is used to measure the quantitative characteristics of program source code. This metric is based on counting the lines of the source code. The original purpose of its development was to estimate man-hours for a project [5].

LOC is usually represented as:

- kLOC: thousand lines of code
- mLOC: million lines of code

3.1.1 Advantages

SLOC is an easy to compute metric. Line of code is a physical entity and manual counting effort can be eliminated by automatic counting process; thus it has a scope for automation of counting the lines of code.

3.1.2 Disadvantages

SLOC can serve some useful purposes but it has some limitations and drawbacks too. For example:

1. It is a poor productivity measure for individuals, because fewer lines of code for some complex function may require more intellectual efforts than a lengthy but simpler function.
2. Similarly two functionality wise identical programs written in two different languages cannot be compared using SLOC, because lines of code can vary a great deal depending upon the language syntax and style.
3. It is also not useful to compare hand written code and auto-generated code in terms of line of code.
4. SLOC has no counting standards, i.e. it is not clear whether commented lines should be considered as lines of code or not.
5. Dependent upon programming language style. Some languages allow multiple statements per line; in this case which line should be considered as line of code, either physical or logical line, these questions are not answered in the specification of SLOC.

3.1.3 Motivation

The limitations of SLOC may give an impression of a less useful metric, but wise use of this metric can still be valuable. Variants of this metric are proposed by some scientists that can, for example, distinguish between physical and logical line of code.

Specifically to the requirements of this thesis, SLOC can be used to monitor 'change' in terms of lines of code between different build versions of the software.

This metric can be calculated for individual functions of a program. If a function is too large in comparison with the average length per function, it may indicate that the function is hard to maintain and hence complex in terms of maintainability.

The unit of this metric is a simple number which represents the number of lines in a source code file. This number can be used with other code metrics to formulate new complexity indicator that can be more comprehensive.

3.2 Halstead Software Science

A suite of metrics was introduced by Maurice Howard Halstead in 1977. This suite of metrics is known as Halstead software science or as Halstead metrics.

Most of the product metrics typically apply to only one particular aspect of a software product. In contrast, Halstead set of metrics applies to several aspects of a program, as well as to overall production effort [6].

Halstead metrics are based on the following indices:

- *n1* - distinct number of operators in a program
- *n2* - distinct number of operands in a program
- *N1* - total number of operators in a program
- *N2* - total number of operands in a program
- *n1'* - number of potential operators
- *n2'* - number of potential operands

Halstead refers to *n1'* and *n2'* as the minimum possible number of operators and operands for a module or a program respectively. This minimum number would occur in a programming language itself, in which the required operation already existed (for example,

in C language, any program must contain at least the definition of the function *main()*), possibly as a function or as a procedure; in such a case, *n1'=2*, since at least two operators must appear for any function or procedure: one for the name of the function and one to serve as an assignment or grouping symbol. Next, *n2'* represents the number of parameters, without repetition, which would need to be passed on to the function or the procedure [20].

3.2.1 Halstead formulas

On the basis of above mentioned indices (*n1, n2, N1, N2, n1', n2'*), Halstead derived more than a dozen formulas relating to the properties of code. These formulas can measure program vocabulary (*n)*, program Length (*N)*, program volume (*V)*, potential volume (*V')*, program Level (*L)*, total effort (*E)* and development time (*T)*. Halstead named his formulas as "Halstead's Software Science Metrics".

3.2.1.1 Program vocabulary

Halstead views a computer program as sequence of tokens where each token is either operator or operand. He defines program vocabulary as:

$$n = n_1 + n_2$$

where

n_1 = the number of distinct operators in the program,

n_2 = the number of distinct operands in the program,

and

n = the total number of distinct tokens or vocabulary of the program.

3.2.1.2 Program length

Halstead defined program length as the count of total number of operators and total number of operands. The formula is as under:

$$N = N_1 + N_2$$

where

N_1 = total number of operators in the program

N_2 = total number of operands in the program

and

N = program length or size of the program

3.2.1.3 Program volume

Halstead defines another measure of program size, the program volume. It is calculated as:

$$V = N \cdot log_2 n$$

where,

N = program length, calculated from Halstead formula for program length

n = program vocabulary, calculated from the formula of Halstead program vocabulary

and

V = volume of the program.

Volume can be interpreted as bits, hence V is the measure of storage volume required to represent the program [6].

Halstead observed that there is a relationship between code complexity and program volume. According to Halstead, *code complexity increases as volume increases*.

3.2.1.4 Potential volume

Potential volume is expressed as:

$$V' = N' \cdot log_2 n'$$

where

$N' = n_1 \cdot log_2(n_1) + n_2 \cdot log_2(n_2)$ is potential length of the program

$n' = n_1' + n_2'$, where n' is potential dictionary of the program

3.2.1.5 Program level

Halstead software science also introduced program quality level, which is obtained as:

$$L = V' / V$$

where V' is potential volume of the program and V is program volume.

The closer L is to 1, the better the program quality is. The ideal value for L is 1. Halstead also found a relationship between program level and code complexity and he claimed that, code complexity increases as program level decreases.

3.2.1.6 Total effort

Halstead metrics can be used as a simple theoretical model of the software development process. The effort required to develop software is given by the equation

$$E = V / L,$$

which can be approximated as:

$$E = \frac{n_1 n_2 [n_1 log_2 n_1 + n_2 log_2 n_2] log_2 n}{2n_2}$$

The units of E are elementary mental discriminations [6].

3.2.1.7 Development time

Development time can be obtained from the Halstead's formula:

$$T = E / S$$

where

E is total effort and S is Stroud number[1]. Value of S is usually taken as 18 for these calculations.

If only the value of length N is known, then the time is can be calculated by the following formula:

$$T = \frac{N^2 log_2 n}{4S}$$

where N can be obtained by using $n log_2 (n/2)$

3.2.1.8 More formulas

As stated earlier, Halstead derived more than a dozen formulas. Below is the short description of some of the Halstead formulas [5]. These formulas aren't related to code complexity.

- $L' = (2n_2)/(n_1 * N_2)$ - programming quality level based only on the parameters of a real program without taking into consideration theoretical parameters

[1] *Stroud number: Halstead used a number called the Stroud number (ranging from 5 to 20), which represents how many elementary discriminations the human brain can perform in a "moment." Halstead also postulated the human brain can handle up to five chunks of material simultaneously [21].*

- $E_c = V/(L')^2$ - the program's understanding complexity
- $D = 1/L'$ - labor intensiveness of the program's coding
- $Y' = V/D^2$ - expression language level
- $I = V/D$ - information content of the program; this characteristic allows you to estimate intellectual efforts on creation of the program
- $E_i = N' \cdot log_2(n/L)$ - estimate of necessary intellectual efforts when developing the program characterizing the number of necessary elementary solutions when writing the program

3.2.2 Problems with Halstead metrics

Although Halstead software science enable us to perform lot of useful measures on code but his work has also received criticism on different forums.

1. Halstead methodology for deriving some mathematical relationship has faults in view of some researchers [7].
2. Halstead used a number, called Stroud number (ranging from 5 to 20), which represents how many elementary discriminations a human brain can perform in a moment. This assumption of valuing Stroud number is maybe incorrect [7].
3. Another assumption of Halstead is that human brain can handle up to five chunks of material simultaneously. He used this assumption in the formulation of his equations. This assumption is may not be a precise assumption as well.
4. Halstead metrics are difficult to compute. For example it is not easy to define and count distinct number of operators and operands in a program. Therefore, gathering this metric itself is not quick if a program is using large number of operators and operands.
5. Halstead didn't specify that which value of program level makes the program complex. He just mentioned that code complexity increases as program level decreases.
6. Computation of Halstead metrics on "Bubble sort" suggests that the algorithm is very complex, but in the view of most programmers the algorithm is not complex [7].

3.2.3 Motivation

Halstead software science is a comprehensive set of metrics which enable us to measure different aspects of a program. Although there are issues/criticism on Halstead metrics but it's comprehensiveness motivates the use of it. Halstead metrics are part of many commercially used software for measuring code complexity, this also gives motivation to consider it.

It is good to have more than one metric for the quantitative measure of a program and not fully rely on one metric (e.g. SLOC).

Halstead measures are or can be used to compute other metrics, i.e. some of Halstead measures can serve as an input for the computation of some other metric. Example of such metric where Halstead effort metric is used is Maintainability Index metric.

3.3 ABC Metric

ABC metric is designed to measure the software size. This metric was introduced by Jerry Fitzpatrick in 1997. In this metric, Jerry tried to overcome the disadvantages of SLOC and other similar metrics.

Imperative languages, like C, have three fundamental operations: storage (variables), test (conditions), and branching. A raw ABC value is a vector that has three components [10]:

- **A**ssignment - an explicit transfer of data into a variable
- **B**ranch - an explicit forward program branch out of scope
- **C**ondition - a logical/boolean test

ABC values are written as ordered triplet of integers, e.g. ABC = <7, 12, 2> where 7 is the assignment count (variables), 12 is the branch count (represents function calls), and 2 is the condition count (for example if/else conditions in the program).

In order to obtain a single value to represent the size of a program, Jerry provided the following formula:

$$|ABC| = sqrt((A*A)+(B*B)+(C*C))$$

3.3.1 Advantages

1. It's an easy to compute metric.
2. Independent of a programmer's programming style.
3. It is possible to get ABC measurement for any module or piece of the code, whether it is a package, class, and subroutine and so on.
4. More insight view of the nature of a program is possible by retaining the ABC count in vector form.

3.3.2 Disadvantages

1. ABC metric somehow reflects the actual working size of a program, but not the actual length of the program. It can give the size value of 0 if a program is not performing any task but maybe there are few lines of code present in the program.
2. ABC metric is not widely adopted for size measurement.

3.3.3 Motivation

ABC metric provides an alternate method to measure the size and it is not hard to implement this metric. The use of this metric will provide a different opinion about the size and can be useful for the size comparison calculated by SLOC or Halstead metrics.

3.4 McCabe Cyclomatic Complexity

Thomas McCabe introduced a metric in 1976 based on the control flow structure of a program [15]. This metric is known as McCabe cyclomatic complexity and it has been famous code complexity metric throughout since it was first introduced.

The McCabe metric is based on measuring the linearly independent path through a program and gives cyclomatic complexity of the program which is represented by a single number.

McCabe noted that a program consists of code chunks that execute according to the decision and control statements, e.g. if/else and loop statements. McCabe metric ignores the size of individual code chunks when calculating the code complexity but counts the number of decision and control statements.

McCabe method maps a program to a directed, connected graph. The nodes of the graph represent decision or control statements. The edges indicate control paths that define the program flow [7]. Cyclomatic complexity is calculated as:

$$M = E - N + P$$

where

M is McCabe metric

E is the number of edges of the graph of a program

N is the number of nodes of the graph

P is the number of connected components

P can also be considered as the number of exits from the program logic.

McCabe compared his cyclomatic complexity with the frequency of errors and suggested the following relationship between Cyclomatic complexity and code complexity for a function. The recommended ranges are shown in Table 3.1

Cyclomatic Complexity	Code Complexity
1 – 10	A simple program, without much risk
11 – 20	More complex, moderate risk
21 – 50	Complex, high risk
50+	Untestable, very high risk

Table 3.1 – McCabe cyclomatic complexity ranges

3.4.1 Advantages

1. Cyclomatic complexity is easy to compute and apply.
2. It guides the testing process. It gives the minimum number of test cases required for a program.
3. It can be computed prior to coding, from the detail design of a program.
4. It can be used as a quality metric as it gives the relative complexity of various designs.
5. There are number of existing tools available to measure cyclomatic complexity for many different programming languages including C.

3.4.2 Disadvantages

1. Program's data is ignored in the cyclomatic complexity; it only provides the code complexity of program flow control.

2. Weighing scheme is too simple. For example same weight is placed for nested and non-nested loops. However nested conditional structures are harder to understand than the non-nested structures.
3. This metric only considers a program complex which has more number of decision and control statements. It ignores the size of the program or size of the code under each node (i.e. code chunk).
4. It may give a misleading complexity value with regard to a lot of simple comparisons and decision structures.

Note: The organization, for which this code complexity measurement is being performed, has previously used McCabe cyclomatic complexity, in 1987, for their software written in Plex. They got very high complexity values (above 400 mostly) for their code. They even got the cyclomatic complexity of around 1000 for some program. According to McCabe, program is non-testable and has very high risk if cyclomatic complexity is above 51. McCabe himself was surprised to see these high complexity values when he was contacted by a representative from the said organization.

3.4.3 Motivation

This metric provides the code complexity based on the analysis of program structure. In order to compare significant changes between different builds of the software, cyclomatic complexity can be used to measure the relative complexity of the builds. If there will be significant variation of cyclomatic complexity for a same module or component in latest and previous build version, it will indicate the part of software which has changed a great deal.

Quantitative metrics do not take into account the program's control flow while measuring the code complexity. A more decent measure of code complexity needs to measure the complexity of both quantitative and flow control properties of the code. Therefore McCabe metric is a potential choice along with quantitative metrics.

3.5 Henry and Kafura's Information Flow Metric

Henry and Kafura developed a metric based on the information flow connection between a procedure and its environment, called fan-in and fan-out.

fan-in: the number of local flows into a procedure plus the number of global data structures from which the procedure retrieves information.

fan-out: the number of local flows from a procedure plus the number of global data structures which are updates by the procedure.

To calculate fan-in and fan-out for a procedure, a set of relations is generated that reflects the flow of information through input parameters, global data structures and output parameters. From these relations, a flow structure is built that shows all possible program paths through which updates to each global data structure may propagate [8].

Complexity is defined as:

$$C_p = (fan-in_p \times fan-out_p)^2$$

Another definition of this metric is:

$$C_p = (ProcedureLength) \times (fan-in_p \times fan-out_p)^2$$ [6]

Procedure length can be obtained by using some quantitative metric, for example Halstead metric L or SLOC.

3.5.1 Advantages

1. It is a good metric for data-driven programs.
2. It can be derived prior to coding, during the design phase.
3. It can be used as maintainability metric, as high value of information flow complexity indicates lack of maintainability [9].

3.5.2 Disadvantages

1. It can give complexity value 0 if a procedure has no external interactions.

3.5.3 Motivation

This metric can indicate about the maintainability and measures the code complexity on the basis of information flow, therefore it is a useful metric and unique of its kind.

3.6 Maintainability Index (MI)

Software maintenance includes all post implementation changes made to a software entity [13]. IEEE defines software maintainability as "the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment."

Maintainability index (MI) is a single valued metric where index value between 0 - 100 is calculated to represent the ease of maintainability of the code or software product. Higher or closer to 100 value of MI means better maintainable code while low value will represent the code that will be hard to maintain. Visual Studio 2010 has this metric where it uses green, yellow and red color coding to represent different ranges of MI values as shown in the table 3.2.

MI range	Color code	Maintainability
0 - 9	Red	Low maintainability
10 - 19	Yellow	Moderate maintainability
20 - 100	Green	Good maintainability

Table 3.2 – Maintainability index

3.6.1 Construction of MI

The original construction of MI was based on the experiments on a suite of eight programs code obtained by HP. 40 different metrics were calculated on this code suite and the metrics were identified that could predict or can be used as a indicator of maintainability. Hence a set of these leading indicators was included in maintainability model.

From the resulting correlations and the principal component analyses it was apparent that [12]:

1. The Halstead metrics for program effort (E) and volume (V) were very strong predictors of maintainability.
2. The two cyclomatic complexity metrics, the McCabe cyclomatic complexity (V(g)) and the Myers extended cyclomatic complexity (V(g')), were equally significant predictors of maintainability.
3. The number of lines of code (LOC) and number of lines of comments (CMT) contributed significantly to the prediction of maintainability.
4. Subroutine averages, rather than total metric values, tend to be more stable and, therefore, more indicative of the true nature of the maintainability of a software suite.

On the basis of these maintainability indicator metrics, MI is defined for one, four and five metric models as:

3.6.1.1 Single metric MI Model

$$MI = 125 - 10 \times log(aveE)$$

Where

MI is Maintainability Index

$aveE$ is average Halstead effort per module

3.6.1.2 Four metric MI Model

$$MI = 171 - 3.42 \times ln(aveE) - 0.23 \times aveVG2 - 16.2 \times ln(aveLOC) + 0.99 \times aveCMT$$

where

MI is Maintainability Index of the module

$aveE$ is average Halstead effort per module

$aveVG2$ is average extended cyclomatic complexity per module

$aveLOC$ is average lines of code per module

$aveCMT$ is average number of lines of comment per module

3.6.1.3 Five Metric MI Model

$$MI = 138 - 2.76 \times ln(aveE) - 0.33 \times aveVG2 - 12.2 \times ln(aveLOC) + 0.88 \times aveCMT + 1.04 \times EDOC$$

where

MI is Maintainability Index of the module

$aveE$ is average Halstead effort per module

$aveVG2$ is average extended cyclomatic complexity per module

$aveLOC$ is average lines of code per module

$aveCMT$ is average number of lines of comment per module

$EDOC$ is a 20 point subjective evaluation: 15 point for external documentation and 5 points for the ease of building the system

3.6.2 Advantages

1. This metric is not hard to compute if other metrics (the metrics on which this metric is based) are already being calculated by the tool.
2. It provides three alternate models for computing the MI.
3. It is a single-valued metric with a specified range.
4. Improved versions of the original methods are available for this metric.

3.6.2 Disadvantages

1. If maintainability is the only area of concern, or the metrics it requires for its computation aren't present in a solution, then this metric is hard to compute as it is based upon the calculations of other metrics.
2. The calculation of MI in all its models is based upon Halstead effort metric, but Halstead formulas have received lot of criticism from other scientists.
3. This metric is based upon the correct computation of other metrics; otherwise the prediction about maintainability could be misleading.

3.6.3 Motivation

Prediction about the code, whether it is hard or easy to maintain, is a very useful measurement especially for the large systems where modification or up-gradation is a continuous process. The set of metrics MI uses for its computation are commonly used metrics for most of the code complexity tools. Using the results of those metrics, this metric can easily be calculated. Therefore this metric is not costly to compute, nevertheless quite useful for any comprehensive complexity measurement tool.

3.7 Other Metrics

There are lots of other metrics which serve different purposes. Depending upon the goal of complexity measurement, more specific metrics serving to the goal should be studied. For example if the goal is "testability" then metrics related to the structural analysis should be measured.

3.7.1 Variants of Line of Code metric

It was discussed earlier that LOC metric doesn't specify which line should be considered as line of code. But variants of LOC overcome this problem by clearly defining the measurement goal.

Effective Lines of Code (eLOC) is one variant of LOC. It only counts the lines that are not commented, blank, standalone braces or parenthesis. In a way this metric presents the actual work performed.

Logical Line of Code (iLOC) is another variant of LOC. This metric shows the count of logical statements in a program, it only counts the statements which end at semi-colon. This definition of metric is only applicable for languages like C or Java, but for languages like Haskell this metric won't work.

Lines of Comment is another quantitative variant of LOC. It is obvious from its name that it counts the number of lines which are commented in source code. Some developers write code statement and comment on a same physical line. In such cases this metric can be further defined easily.

Comment to Code Ratio is a derived metric from eLOC and Line of Comment metric. This simple and easy to compute metric can provide hint for "understandability of the code". It can be obtained as follows:

$$CommentToCodeRatio = (LinesOfComments / eLOC) \times 100$$

There are more possible variants of LOC, for example counting blank lines or white spaces etc.

3.7.2 Function metrics

Metrics that measure some property of a source code function are belonging to this class of metrics. Most of the metrics we have seen so far are countable independently for functions. But there are some more simple metrics related to the functions.

Function Parameter Metric is one example of this kind. This metric only counts the input parameters of a function. On the basis of this count one can guess the functional complexity. For example total number of parameters for a function exceeding 6 can make that function complex, little hard to understand and test it.

Function Return Point Metric counts the number of exit points for a function. The more exit statements a function would have the more complex its functionality would be.

Function Interface Complexity is comprised of the number of function parameters and the number of function return points. Functions with greater than 6 input parameters can be very difficult to use on a routine basis and are problematic to parameter ordering. Functions with greater than one return point break the single entry/single exit design constraint. Functions of this type are difficult to debug when a runtime error occurs [14].

3.7.3 File/Project metrics

The metrics belong to the type of function metrics are either directly countable for files or the result of those metrics can be used to derive new metrics for files. However file has some unique countable properties as well. For example the number of functions in a file, number of operators and number of header files etc. Then averages of different function metrics can also make new file metrics, for example average cyclomatic complexity per function in a file. Therefore lot of metrics for measuring certain file properties are present or can be derived.

Similarly there are project metrics available too. Example is total number of files in a project, average line of code per file in a project, effort metric, and average cyclomatic complexity. Most of the project metrics are not unique, but are often derived from file/function metrics.

3.8 Combination of Metrics

As we have seen that there are large number of code metrics available and lot more metrics are possible to derive. In such situation it becomes difficult to pick up the reasonable number of metrics for a decent complexity analysis/measurement solution. If goal of the measurement is obvious and limited, then one can choose specific types of metrics specific to that purpose. The beneficiaries/users who will use these measurements are also an important factor when selecting a set of metrics. Depending upon user type/requirements, metrics selection would vary. There can be different scenarios in terms of user group and metrics selection.

If users of the system are software developers then probably large number of metrics from each metric group should be picked up for a solution. In this case the user will get in depth view of all different aspects of the code. User can take advantage of the tool for refactoring complex components of the code, take help in developing test case, analyze the quality of the code etc. The other type of users could be team leaders for the development teams; large/medium number of metrics would still be helpful for this type of users.

If primary users of the systems are managers then less but key metrics should be selected. Selection would also depend upon the goal/goals required from the complexity analysis. From management perspective probable goals are maintainability, portability, complexity analysis and quality etc. The metrics capable of measuring these goals can be selected.

A comprehensive code analysis solution that can be beneficial for developers as well as for the team leaders and managers should have small/medium number of metrics from each different group. The solution should be capable of measuring the code complexity at each level of the code (from function to project/package/build level). The solution should not just present the metric values but it should also guide/indicate the most complex part of the code. According to the definition of code complexity, the complexity comprises on multiple factors which include understandability and maintainability. Therefore a good solution should have metrics indicating these properties of the code. Approximately a combination of 15-20 different metrics would cover all important aspects of the code and would be suitable for most user needs.

3.8.1 Proposed metric suite

The following suite of metrics is proposed for a generic solution for code complexity measurement.

1. Lines of Code metric
2. Lines of Comments metric
3. Code to Comment Ratio metric
4. ABC size metric
5. McCabe cyclomatic complexity (or variant of this metric)
6. Function interface complexity metric (require 2 more metrics for its calculation)
7. Halstead software science metrics (6 unique metrics requires to be gathered and then dozen of metrics can be obtained from Halstead formulas)
8. Maintainability index metric (requires Halstead effort metric, McCabe cyclomatic complexity, Average line of comments per module)

Possible achievements from the proposed metric suite

1. More than 20 metrics can be obtained by the gathering of 12 unique metrics.
2. The solution will measure the size of the code using 3 different methods, i.e. Lines of Code, Halstead program length, ABC size measure.
3. The solution will measure the structural complexity of the code by calculating McCabe cyclomatic complexity.
4. McCabe cyclomatic complexity can be used as a guide for developing test cases.
5. Maintainability index will provide an indication about the maintainability of the code.
6. The solution will give a hint about code's understandability; this will be achieved through Comment to Code ratio metric.
7. Using the comparison between releases, it would be easy to monitor the change of code between the releases.
8. Halstead formulas will provide some useful measures like development time and total effort etc.
9. Quality of the code can be assessed.

Chapter 4

EXISTING TOOLS FOR CODE COMPLEXITY

There exist many tools for code analysis and measurement in the market, for example QA-C and Testwell CMT++. Some of them are freeware and open-source while others are proprietary tools. They differ a great deal in their features, support for languages, support for platforms, licensing prices and other aspects.

The suitable tool for this project should meet the following core requirements:

1. The tool must support C language as the source code being analyzed in this thesis is written in C.
2. The tool should be available for Solaris.

The preferred tool should offer more than just fulfilling the core requirements. The desired additional features are:

1. The tool should have integration possibility with ClearCase and makefile.
2. The tool should measure all or most of the metrics which were proposed in the previous chapter. Additional metrics are bonus.
3. Metrics should be calculated for function, file and project level.
4. Metrics should indicate the good or bad code in terms of complexity measurements. The maximum suggested values for the metrics should ideally be formulated for C language.
5. If the tool takes care of the assembly code mix with C, it would be a plus.
6. The generated report should be in XML or CSV formats or stored in database for later use.
7. Command line interface is preferred.
8. If the tool is proprietary, then the fully functional trial license should be provided for the thorough analysis of the tool.
9. The tool should support automatic measurements on regular basis.
10. The tool should provide comparisons between/among software releases.

On the basis of these core and additional requirements, many different tools were analyzed. The most suited tools are presented in this chapter.

4.1 Understand

Understand is a commercially available static analysis tool for maintaining, measuring and analyzing source code. This tool is developed by Scientific Toolworks Inc. (SciTools). This tool is thoroughly evaluated to see up to which extent this tool meets with the specific requirements for this project.

4.1.1 Tool evaluation

- **Language support** - This tool can analyze 14 languages including C/C++, Java, FORTRAN and some web programming languages like PHP.
- **OS support** - It is available for all major operating systems including Solaris.
- **Metrics** - It can gather large number of metrics including many basic metrics, some advance metrics and some custom metrics. During the evaluation of this tool it is found that some of the commonly used metrics aren't included in this tool, for example is it missing all Halstead metrics. It is also observed that all metrics aren't available for all the languages the tool can analyzed. However it can gather about 74 metrics and most of the metrics are available for C/C++.
- **Other features -** It is a GUI tool for all operating systems this tool supports. It outputs reports in graphical, textual and HTML format. It comes with a programming editor. It has a code check feature that checks the code for coding standards. The coding standards it has for code check are Effective C++ (3rd Edition), MISRA-C 2004 and MISRA-C++ 2008. A very nice feature of this tool is the plug-in support which allows users to define and add new metrics to the functionality. This tool is a proprietary of SciTools and the license cost for single user is $995 and floating license cost is $1995.
- **Critical analysis** - This tool is more useful for the developers than the management. It has features like code check and code editor that aren't generally needed for the management. Although this tool can gather large variety of metrics but it doesn't provide any recommended maximum and minimum values for the metrics. The feature of comparison among various releases of software is also not a part of this tool. Project is created manually and since it has the GUI interface it is maybe hard to automate it. Manually created projects with an in-built feature of code editing allow for unwanted changes to the code and lead to the problems of code security and safety. It has no in-built support for integration with any code repository; therefore code has to be available locally before the analysis can perform. During the analysis on actual code it produced some strange results. Value of 4576376.0000 is calculated for a metric "CountPath" when this tool was analyzing a package "Test". This value is way too strange and with no recommended maximum or minimum guidance, it even becomes more confusing.
- **Conclusion** - Keeping in view all the features of this tool, the problems it has and the specific requirements for this project, it was decided not to proceed with this tool.

4.2 Resource Standard Metrics (RSM)

Resource Standard Metrics or RSM is a source code metrics and quality analysis tool. RSM is a product of "M Squared Technologies" and it is commercially available tool for code analysis.

4.2.1 Tool evaluation

- **Language support** - RSM supports C/C++, C# and Java

- **OS support** - RSM can operate on Windows, Linux, Mac OS X and UNIX operating systems.
- **Metrics** - RSM claims that it can collect hundreds of source code metrics by function, class, file and project. The evaluation of RSM shows that it can gather around 30 unique metrics. Most of these unique metrics belong to quantitative group of metrics, i.e. the metrics reflects quantitative measures of the code; for example Line of Code, Effective Line of Code, Comments Lines, Blank Lines metrics etc. There is one metric for measuring structural complexity, that metric is Cyclomatic complexity. This tool has many metrics for quality analysis, but these metrics are language dependent. Some of the metrics are only for object-oriented languages and aren't available for C.
- **Other features** - RSM has integration feature with popular IDE's like Eclipse, Visual Studio and JBuilder. It works as a command line tool under UNIX and it has integration possibility with IBM Rational ClearCase. It can generate reports in HTML, XML, CSV and Text format. It can convert source code from DOS to UNIX format. It's a proprietary of M Square Technologies and its network floating license for UNIX costs $1,999.95 and company license for all operating systems is $9,999.95.
- **Critical analysis** - RSM is a nice tool for metric gathering. It can work on all major operating systems and its availability for UNIX makes it meet with one of the core requirements for this project. Its ClearCase integration is another plus point of this tool. Since RSM is a command line tool under UNIX, it increases its scope for the automation and integration of this tool with the current build environment of the company for which this project is performed. The default reporting format of this tool is little bit messy and hard to get an overview of the metrics summery quickly. Metrics values have no recommended upper and lower limits by the RSM. Like the previously analyzed tool "Understand", this tool can also be useful for the software developers but may not give much help to the management.
- **Conclusion** - This tool meets with the core requirements for this project and this tool can be used for metrics gathering.

4.3 Testwell CMT++

CMT++ is a complexity measurement tool, by Verifysoft Technology, for source code written in C/C++. Verifysoft Technology also provides complexity measurement tool for other languages but with different tool names, for example for complexity measurement tool for Java source code is named as CMT Java.

4.3.1 Tool evaluation

- **Language support** - C/C++
- **OS support** - Available for Windows and Unix only
- **Metrics** - CMT++ gathers limited set of metrics which includes McCabe cyclomatic complexity, Halstead software science metrics, Line of Code metrics and Maintainability Index.
- **Other features** - It is a command line tool for UNIX and GUI for Windows. Under Windows environment CMT++ can integrates with Visual Studio. It outputs reports in XML, HTML and CSV formats. It requires no compiler for C/C++ and it can calculate some metrics even if C code contains some assembly code in it.
- **Critical analysis** - CMT++ is a simple tool which keeps it focus on complexity measurements only and do not try to perform too many tasks, like Understand. This tool gathers limited number of metrics but the metrics gathered in this tool are pretty standard metrics. All the metrics in that set are useful and common metrics in many other tools. The other advantage of this tool is that it has recommended

maximum and minimum values for all its metrics. It lacks with ClearCase integration. This tool is a proprietary of Verifysoft Technology but its licensing cost is not specified.

- **Conclusion** - CMT++ is a simple tool by its functionality and meets with the core requirements for this project. But unfortunately it has no trial version available so it is very hard to predict if this tool is usable for this project or not. On request for time bound trial of this tool, no response was received by the Verifysoft, therefore the tool wasn't practically analyzed.

4.4 QA-C Source Code Analyzer

QA-C Source Code Analyzer is C code analyzing tool developed by "Programming Research". Its variants are available for other languages (C++ and Java). This is one of the oldest static code analysis tools available commercially in software industry as it has been available since 1986.

4.4.1 Tool evaluation

- **Language support** - C
- **OS support** - Available for Windows, Solaris (sparc) and RedHat Linux.
- **Metrics** - QA-C gathers a large number of metrics when it analyzed C code. Total number of metrics this tool can gather is above 60. The QA-C metrics are mostly advanced metrics and the numbers of metrics are also large. The metrics include most famous Halstead metrics and cyclomatic complexity metrics as well. QA-C also suggests recommended maximum values for its metrics. It also provides some detail description of each metrics. There are some unique metrics for functions, files and at project level.
- **Other features** - It can identify non compliance to MISRA coding standards. It also has bug detection and bug prevention feature that can be useful for the developers. Another feature that QA-C provides is the code comprehension. It IDE's integration facility as well. It also supports build system integration and makefile integration. QA-C generates graphical and textual reports.
- **Critical analysis** - QA-C is an advanced and comprehensive code analysis tool for C language. The metrics it uses are advance metrics and have recommendations about maximum value limits. Its makefile integration feature supports the need for this project. Its licensing cost is not specified but it is not a free tool.
- **Conclusion** - QA-C meets with all the core requirements of this project. Its metrics are more advance as compared to the metrics of RSM. Its makefile and build system integration features are additional advantages over RSM. Therefore this tool can be preferred over RSM and can be used for this project.

4.5 Other Tools

As mentioned earlier, there are number of other tools available. The most suitable tools for this project are presented in previous sections. However some of the other tools were considered and tested on limited level. Gathering same metrics from multiple tools helps in assessing the correctness of the metric values.

4.5.1 CCCC

CCCC is an open-source tool for the measurement of code metrics. It is a command line tool and is available for Solaris. This tool was developed as a master thesis and it was released as an open-source, but there is low development activity on this tool. This tool gathers very few code metrics. Testing of this tool on our code made the values of its metrics doubtful!

4.5.2 Metric Advisor

This tool is a proprietary of CDAC, India. It is capable of gathering 6 metrics, but the metrics are advanced and useful ones. It has GUI interface and it only support C language and it is available for Solaris. There is no trial version available for this tool and licensing cost is also unknown. This tool wasn't further assessed as CDAC didn't respond to the request for a trial version.

4.6 Tools Evaluation Summery

The tools evaluation proved that functionality wise the tools differ greatly. Some of the tools try to do too much, from complexity measurements to coding standard checks, from function level to project level, from bug detection to bug prevention and some provide code editing facility as well. On the other side, some tools keep their focus on complexity measurements only with either small or large number of metrics. Language support also varies among tools. Some tools support multiple languages at a same time while others are one language specific tools. Some tools work as standalone but some has integration features with IDE's. Visual Studio has its own complexity measurement feature in it. LOC or its variants and McCabe cyclomatic complexity are the most common metrics for every tool. The way of creating code analysis reports is also different in most tools.

Most of the tools are designed to help software developers, especially those tools that can be integrated with IDE's and gather large number of metrics.

Chapter 5

CODE COMPLEXITY MEASUREMENT

After identifying the code metrics needed for a comprehensive code analysis and evaluating various tools that can gather those metrics, the next primary task was to either implement or adopt a solution that can perform this analysis. Therefore a tool is developed to provide a solution to the given problem. Code Complexity Analysis (short name CCA) is the name given to this tool. CCA is developed in a way that it can run on regular basis and perform code analysis automatically. Metrics gathering from source code is part of the automatic code analysis and to display the results web-interfaces are designed.

The following section describes the detail about the possible ways of solving the problem and which approach is adopted for the solution.

5.1 Approaches for Metrics Gathering

Three alternative approaches for metrics gathering from source code were proposed:

1. Use an existing open-source or commercial proprietary tool which can accomplish all/most of the desired requirements.
2. Implement our own tool for complexity measurements that can gather all the proposed metrics.
3. Use an existing open-source or commercial proprietary tool as a foundation for gathering metrics and develop our own tool for to fulfill the requirements.

Each of these three approaches has its own benefits and drawbacks. Each approach is discussed briefly and then the adopted approach is presented with the arguments in favor of the selected method/approach.

5.1.1 Open-source or commercial tool

The use of an open-source or a commercially available tool for code complexity measurement could have been the most economical approach as far as time and financial cost is concerned. The thorough tool evaluation proved that none of the tool performs automatic measurement of code complexity. The tools that gather large number of metrics were more likely designed to assist the software developers. Lot of manual efforts, from project creations to comparing various releases and viewing results, are required if this approach is adopted.

5.1.2 Implementing a tool

Implementation of our own tool can overcome all the problems of an open-source or commercial tool. In this approach, the most wanted metrics can be gathered. Highest metrics values can be adopted according to the company's local coding style in regard of C language. The local developers can be interviewed when setting up the scoring criteria for the metrics values. This tool can well integrate with the company's automatic build environment and can be developed in a way that it would remain customizable for future needs. But the problem with this approach is that it will require a lot of time and therefore the duration, to implement it, will exceed from the time limits for a Master thesis.

5.1.3 Developing a tool using an open-source or commercial tool

The third approach is to use an existing open-source or commercial tool for metrics gathering and extends its functionality to fulfill the specific needs of this project. This project can be divided in two broader parts; first is the gathering of metrics and the other is to perform automatic code analysis. This approach will save the time for metrics gathering part and the focus would be set on automatic measurement. Hence this approach is found to be most suitable for this project.

5.1.4 Adopted approach

All three alternative approaches were examined carefully and discussion was held with the client company. Cost of implementing our own tool was discussed and the summery of existing tools evaluation was presented to the client company. The final decision favored the 3rd approach, which is to develop our own tool using an existing tool for metrics gathering part.

The selected tool for metrics gathering is QA-C by Programming Research.

5.2 Development Environment

The details about the development environment in which the automatic measurement procedure is developed are presented in this section. It includes all the technologies, tools and platform used for the development.

5.2.1 Programming languages

- Perl
- JavaScript
- CSS

Perl programming language is used for the development of the tool. The existing build environment in the client company is developed in Perl, therefore Perl is selected for complexity measurement tool as well.

JavaScript and CSS are used for featuring the web-interfaces for the tool.

5.2.2 Operating system platform

- Solaris (sparc)
- Windows XP

The tool is entirely developed on Solaris. However the designing part of web-interfaces is initially developed on Windows platform. Later on a demo for the tool was configured for Windows as well.

5.2.3 Tools and technologies

- QA-C
- Template toolkit
- Adobe Photoshop CS3
- Adobe Dreamweaver CS5
- IBM Rational ClearCase
- TortoiseGit
- Google Visualization API

QA-C is used as a tool for metrics gathering. Template toolkit is used for template based web-interfaces. Adobe products are used for initial web designing.

The C source code analyzed/measured in this thesis is using Rational ClearCase revision control system. The complexity analysis tool developed during this thesis interacts with ClearCase to get access to the code.

TortoiseGit is the revision control system used for handling the code of our code analysis tool. Google Visualization API is used to generate charts for showing comparison among different versions of software code.

5.2.4 Database

- MySQL

MySQL database is used for strong the metrics results and information about the processed code.

5.3 Automatic Metrics Gathering

An automated procedure for gather metrics, by using QA-C, is designed to be run on regular basis to perform code analysis. The short name given to this tool is CCA which stands for Code Complexity Analysis. It is entirely written in Perl and it has modular design which allows for extensions in its functionality.

5.3.1 CCA architecture

Figure 5.1 shows the architecture and explains how different modules interact with each other.

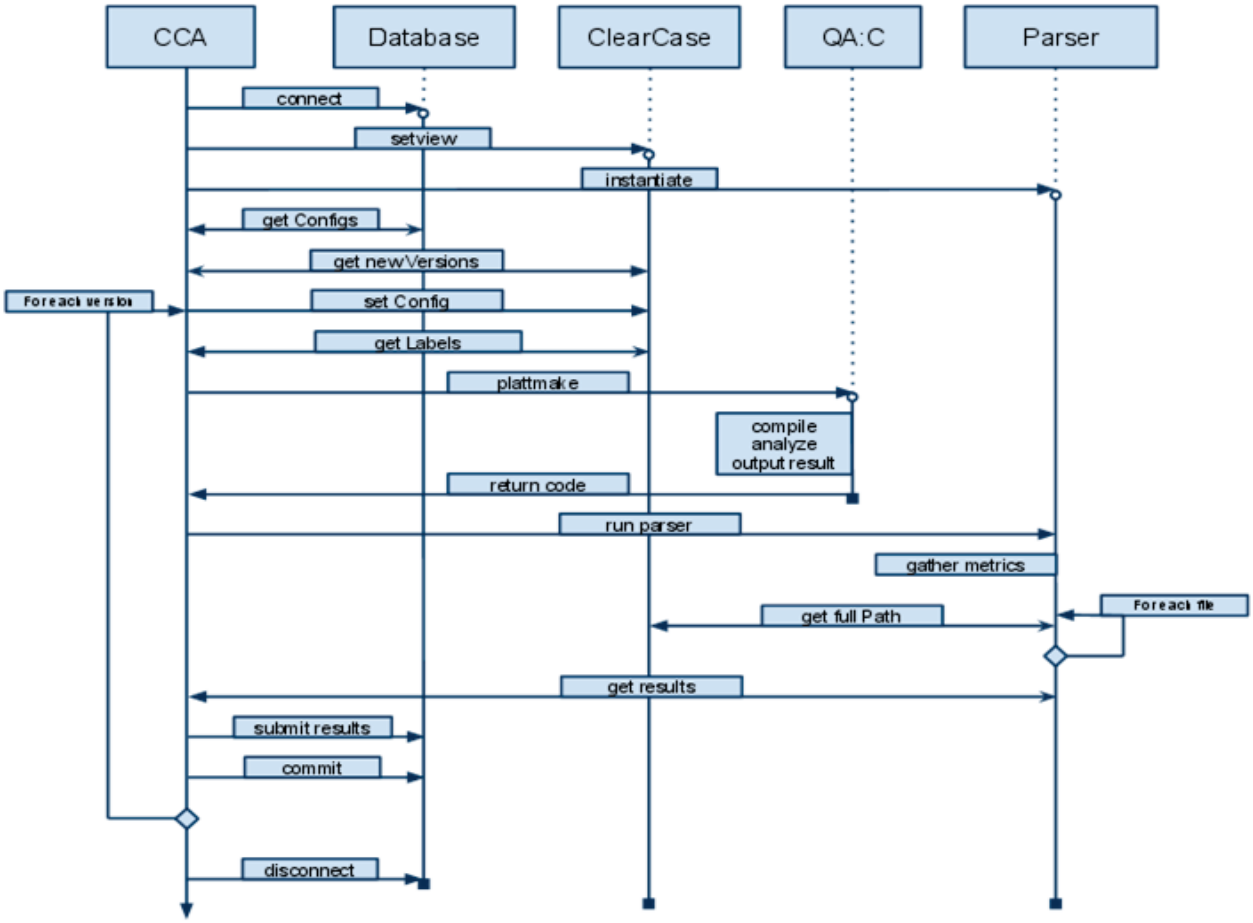


Figure 5.1 – Code Complexity Analysis architecture

CCA handles the interaction with ClearCase and file system. When this tool starts running, it connects with the Database and then it sets a view in ClearCase. Then it instantiates the parser module. Then the tool retrieves configuration specifications from Database. Then it finds the versions of source code from ClearCase against a configuration specification. The tool then interacts with QA-C to perform complexity analysis on the associated source code files. The Parser handles the results of the code analysis performed by the QA-C and sends the results to the CCA. The results are then stored in the Database for later retrieval. The procedure continues to run for each version of the configuration specification.

5.3.2 Database model

The database model is presented below. It shows how the database is designed to store the metric results and other information about builds, packages, sub-packages, files and functions. In order to avoid duplicate storing related to the source code, this tool interacts with another database that already existed in the system.

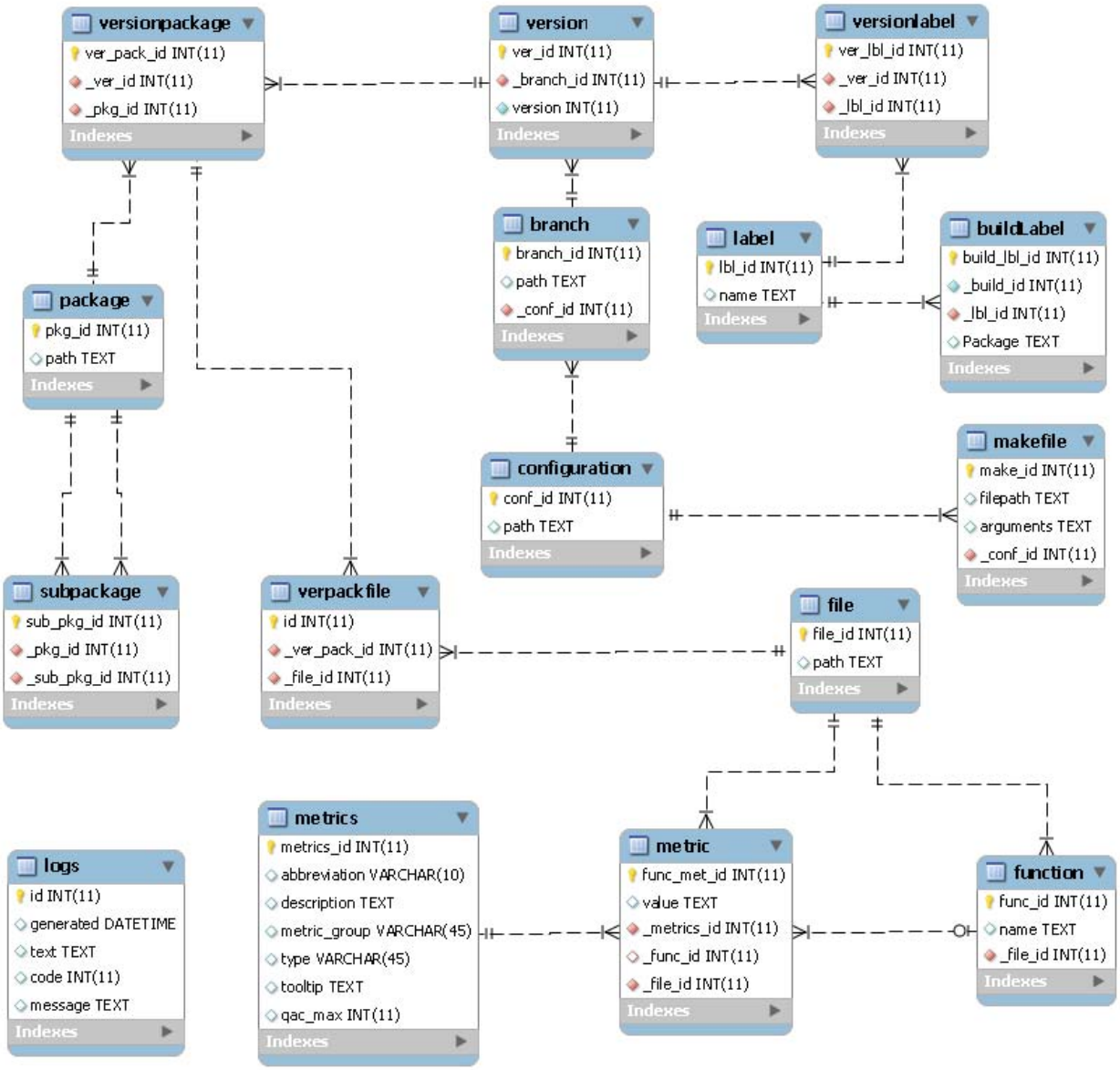


Figure 5.2 – Database model

QA-C gathered metrics have two categories, one is the file metrics and other is the function metrics. This database stores all the metrics results in one place. Functions metrics are directly obtained from metric table. Although there are separate file metrics but since all the functions belongs to some file, therefore file consists of file metrics + function metrics. Metrics for sub-packages or packages aren't stored in any separate placeholder, but are computed from the source code files associated with that package. The complexity metrics for a build/release are computed for all the packages belong to that build/release.

5.3.3 Configuration

Configuring the CCA tool is managed by a simple web interface. It facilitates users with simple management option without interacting directly with the database. Figure 5.3 is a

screen shot of the configuration interface.

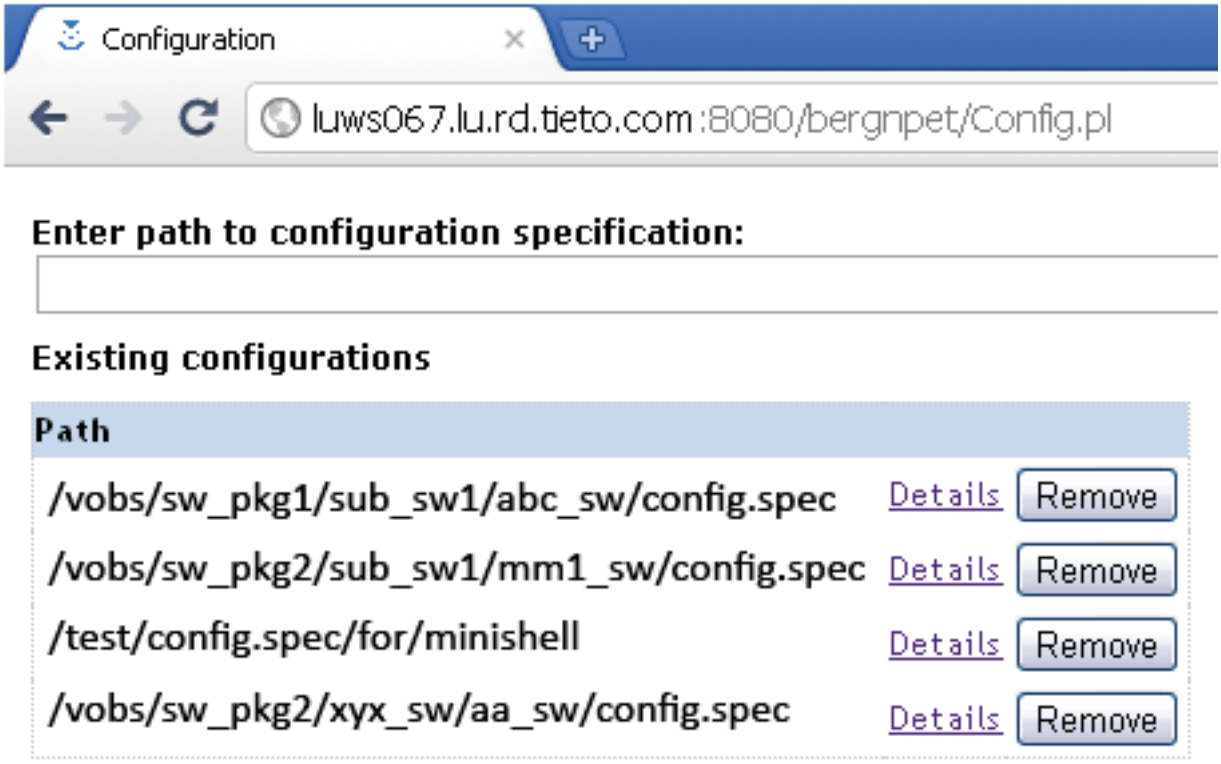


Figure 5.3 – Configuration interface first view

User can simply enter a path to a configuration specification and add it to the database. Already existing configuration specifications can be removed or details can be modified. Figure 5.4 explains the modification options when user clicks on Details.

Figure 5.4 – Configuration interface for editing configuration spec

5.3.4 Logger

The tool is designed to be fault tolerant. During the run it logs each step of the analysis. A simple web-interface is designed to present these logs. If any error occurs during the execution of the tool, for example some file is missing; it logs the error in the database but continues the analysis on other files. Figure 5.5 is a screen shot showing how the web interface displays these logs. It displays the date and time for each logged message. In the following columns, the interface shows the action performed and logged message. Error logs are displayed in red colored rows.

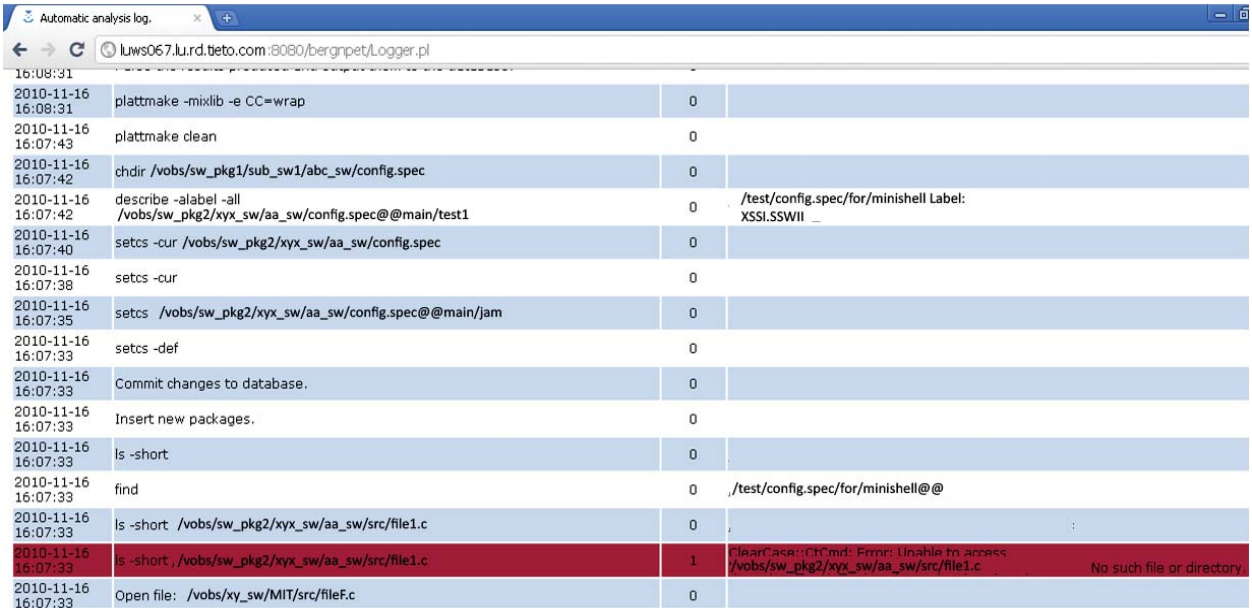


Figure 5.5 – Logger interface view

5.3.5 Web interfaces

Performing automated analysis on a large code base and storing large number of metrics for each file was a hard task. But after having so much information (metric results), it was also a complex task to present the result in a sophisticated manner. Therefore two web interfaces are designed for this purpose, one for reviewing the metrics and other for showing comparison among software releases.

The software architecture adopted for each web interfaces (including Configuration and Logger interface) is model-view-controller software architecture. This architecture provides clean separation between logic, code and design. All the web interfaces are designed to allow for simple alteration. This is achieved by relying on template based approach. Templates are created for all different features of the web interfaces and changing in any feature only require changing the appropriate template. The technologies used for developing these interfaces include HTML, JavaScript, Perl Script (using Template toolkit), CSS and Google Visualization API.

5.3.5 The complexity viewer

For presenting the detailed results of complexity analysis, a user friendly web interface is designed. It allows reviewing metrics results from Build level to the bottom most code function level.

Figure 5.6 shows the overall sketch of how the information is presented. In this figure, file and function metrics are presented for a selected build, and the packages included in that build are displaying at bottom of the figure.

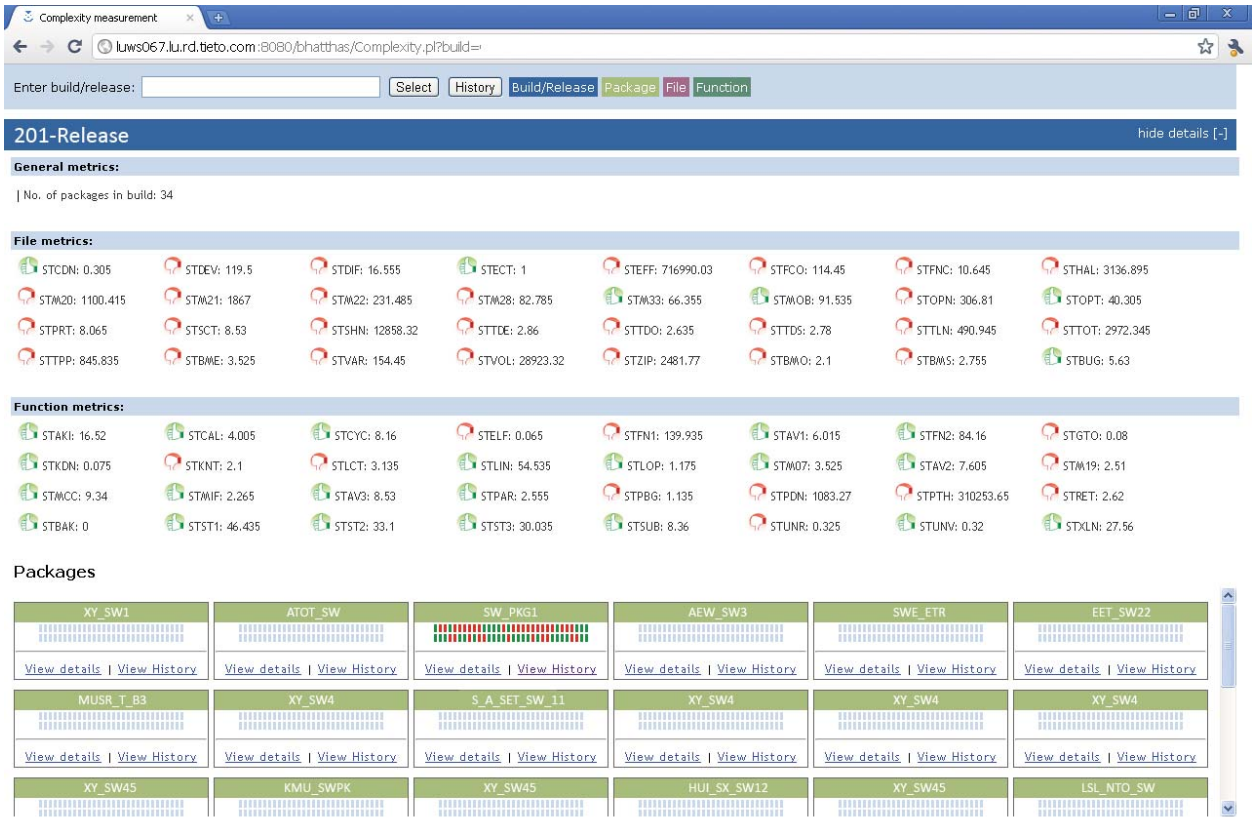


Figure 5.6 – Screen shot of complexity viewer web interface

The first view of the web interface asks for a build/release input. Auto-search-and-complete feature enable user to quickly find and entered the desired release. Figure 5.7 elaborates this functionality even further.

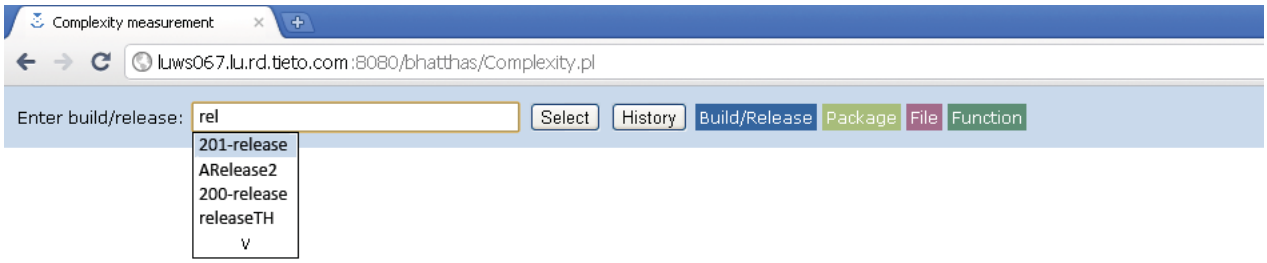


Figure 5.7 – Build/release selection

On pressing any part of the build/release name, the auto-search feature quickly turns on and displays the matching entries; and on pressing the enter makes the auto-completion of the highlighted build name.

In the next step the file and function metrics are presented in two separate groups. List of packages is displayed after that.

Metrics abbreviations used by QA-C are displayed and average database values for each metrics in that build/release. Green color coding and thumb up symbol is placed before all the metrics where the metrics values are within the QA-C maximum recommended limit. Red coloring and thumb down symbol is chosen for the metrics where values are exceeding the recommendations.

On mouse over on any metric displays a tool-tip containing the metric's full name, its recommended maximum value, its database average and some information how this metric is computed. Figure 5.8 explains this tool-tip feature.

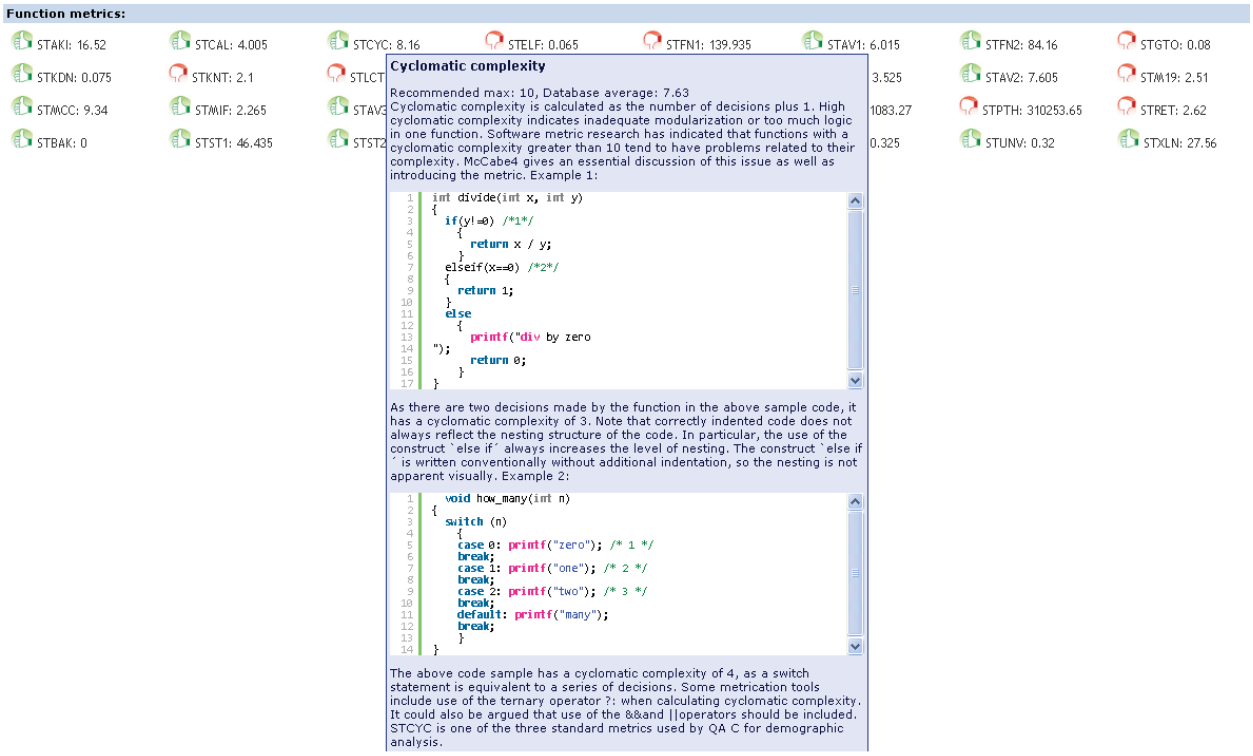


Figure 5.8 – Metric tool-tip

After presenting the metrics associated with the build, all the packages belong to that build are displayed. The metrics gathered for each packages displayed with red and green color coding in the respective box with package name. These color coding help the user to quickly guess the complexity of a package. If there are more green colors, it will explain the user that the package complexity is within the recommended range. If there are more reds then that package reflects the complex code and maybe in need of refactoring.

Figure 5.9 shows a package box and elaborate the color coding.

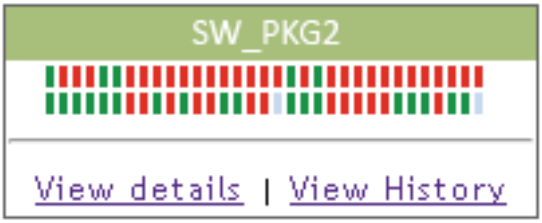


Figure 5.9 – Color coding represents metrics results

The summery of 60 different metrics is displayed in a small box with the package name on the top. The figure shows more reds, indicating more metrics are over the recommended complexity values. If user is interested in viewing the detail of this package, he will click on "View details" link and details of each metrics will be displayed in the same way as it was displayed for the build. If the package will have sub-packages then the color of the header color of each box will remain the same. If the package will have source code files then the header color will change to the specified color (purple) chosen for the files. Associated metrics with the will be displayed in the same way as they were displayed for the build. The same red and green coloring will present a quick view of the files or sub packages. In the similar fashion the complexities can be views till the function level.

5.3.6 The comparison viewer

A separate web interface is designed to show the comparisons among various releases. Single interface with one input field is designed to handle the comparisons among packages, files or functions. The first view of the interface is shown in figure 5.10.

Figure 5.10 – Comparison viewer interface

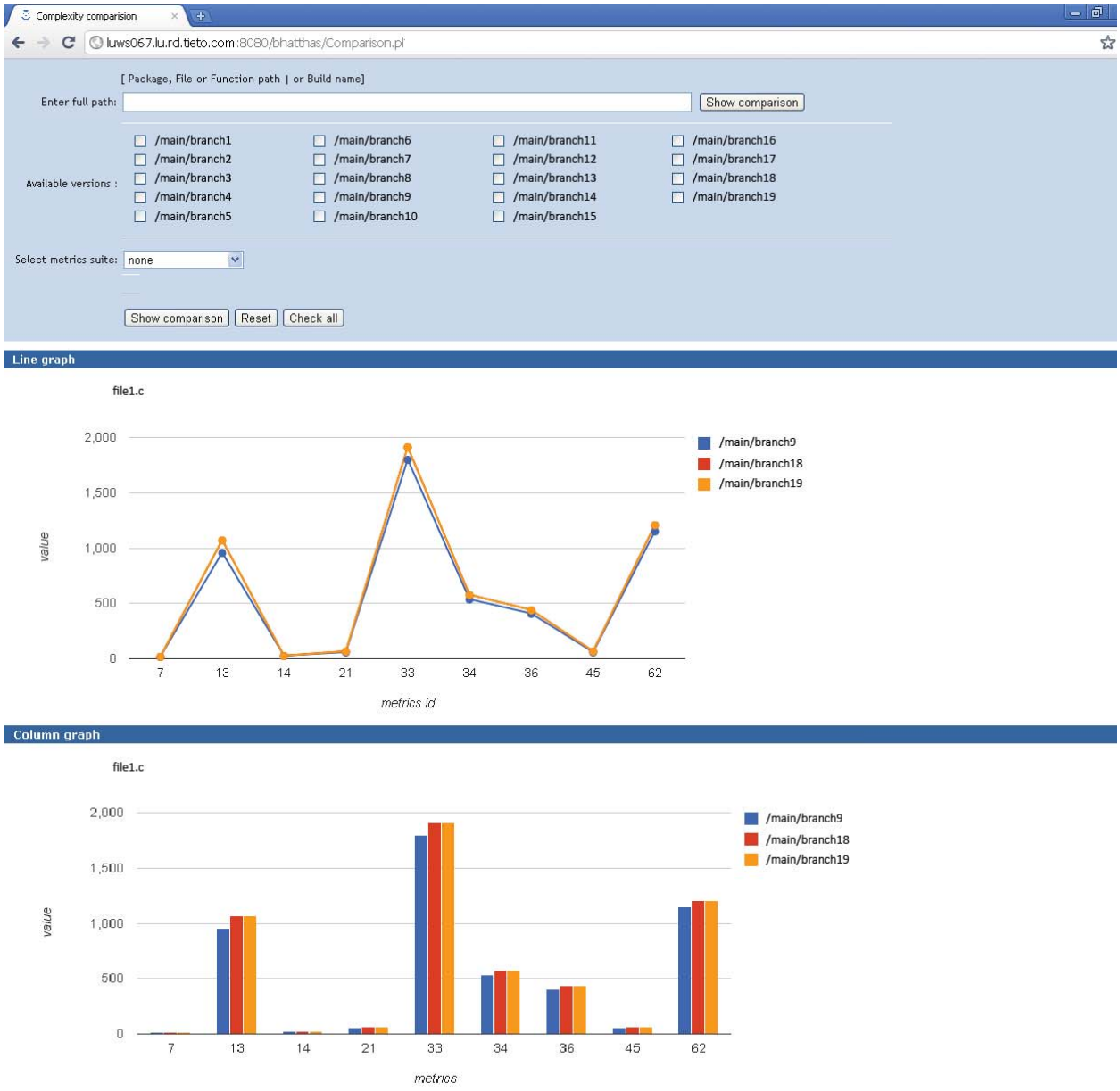
The input requires entering a full path with specified release/version where a source code file is present in the ClearCase. For a function, it requires full path of a file in which the function is present and then the name of the function. It would have been a hassle without the search and auto-complete feature. User can just type any part of either the path, or function name, or file name, or package name and the search will provide the user with the matching suggestions. This feature helps user to avoid type very long paths manually. It just requires couple of seconds to enter the desired path.

Figure 5.11 shows that a user only typed 3 middle letters of a source code file and the full path and version detail is provided in the suggestion. In the same way function or package names can be searched quickly. It also protects from incorrect input.

Figure 5.11 – File selection for comparison viewer

On the selection of file, the comparison of the selected version/release of the file with two of the latest releases is generated. If the selected file version is one from the two latest version of that file, then the comparison is generated between two latest versions.

Comparison is shown in Line graph, column graph and in tabular form. Figure 5.12 illustrates complete view for the file comparison.



Metrics Id	Metrics description	/main/branch9	/main/branch18	/main/branch19
7	Organic programmer months	15.23	16.35	16.37
13	Est. development (programmer-days)	957.7	1067.03	1069.28
14	Program difficulty	22.17	22.72	22.74
21	Number of functions in file	59	64	64
33	Number of statements	1796	1909	1910
34	Number of non-header comments	538	572	574
36	Number of internal comments	406	435	437
45	Est. porting (programmer-days)	58.12	62.19	62.25
62	Total number of variables	1148	1209	1209

Figure 5.12 – A complete view of a file comparison example

In the default comparison option, comparison among three versions (two latest and one selected) is generated. List of all available versions of a selected file is provided to the user

for his customized selection for more comparisons. A combo box contains metrics group helps user to select from a particular type of metrics. These features are explained more in figure 5.13.

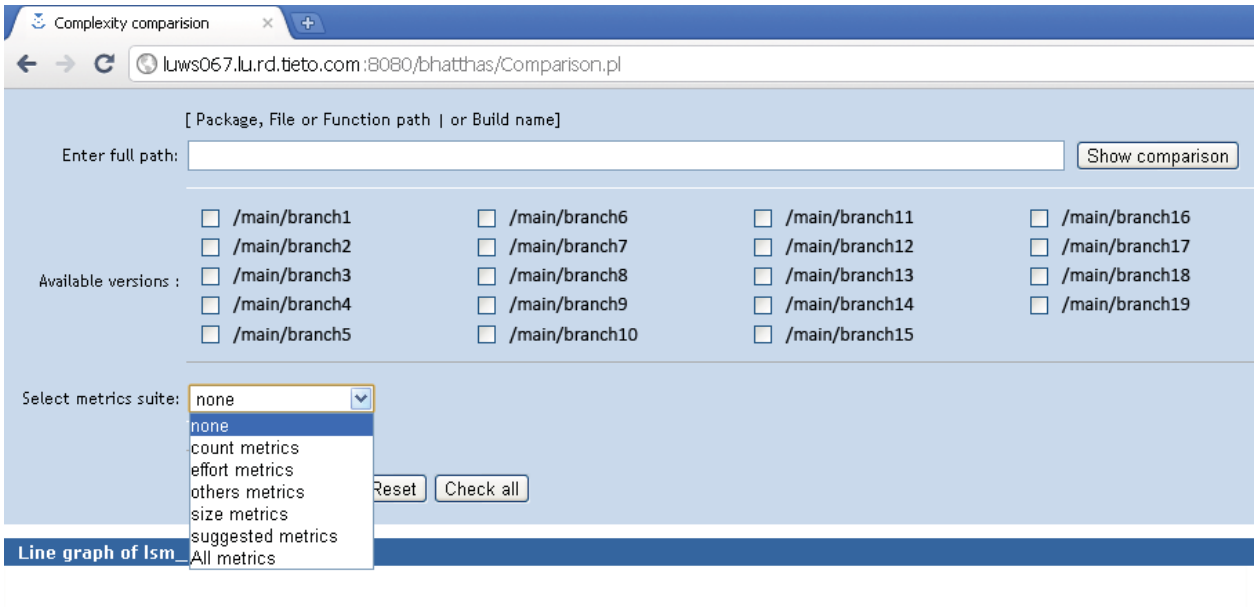


Figure 5.13 – Metric group selection

On the selection of a metric group, all metrics belong to that group appeared on the screen. The illustration is provided in figure 5.14.

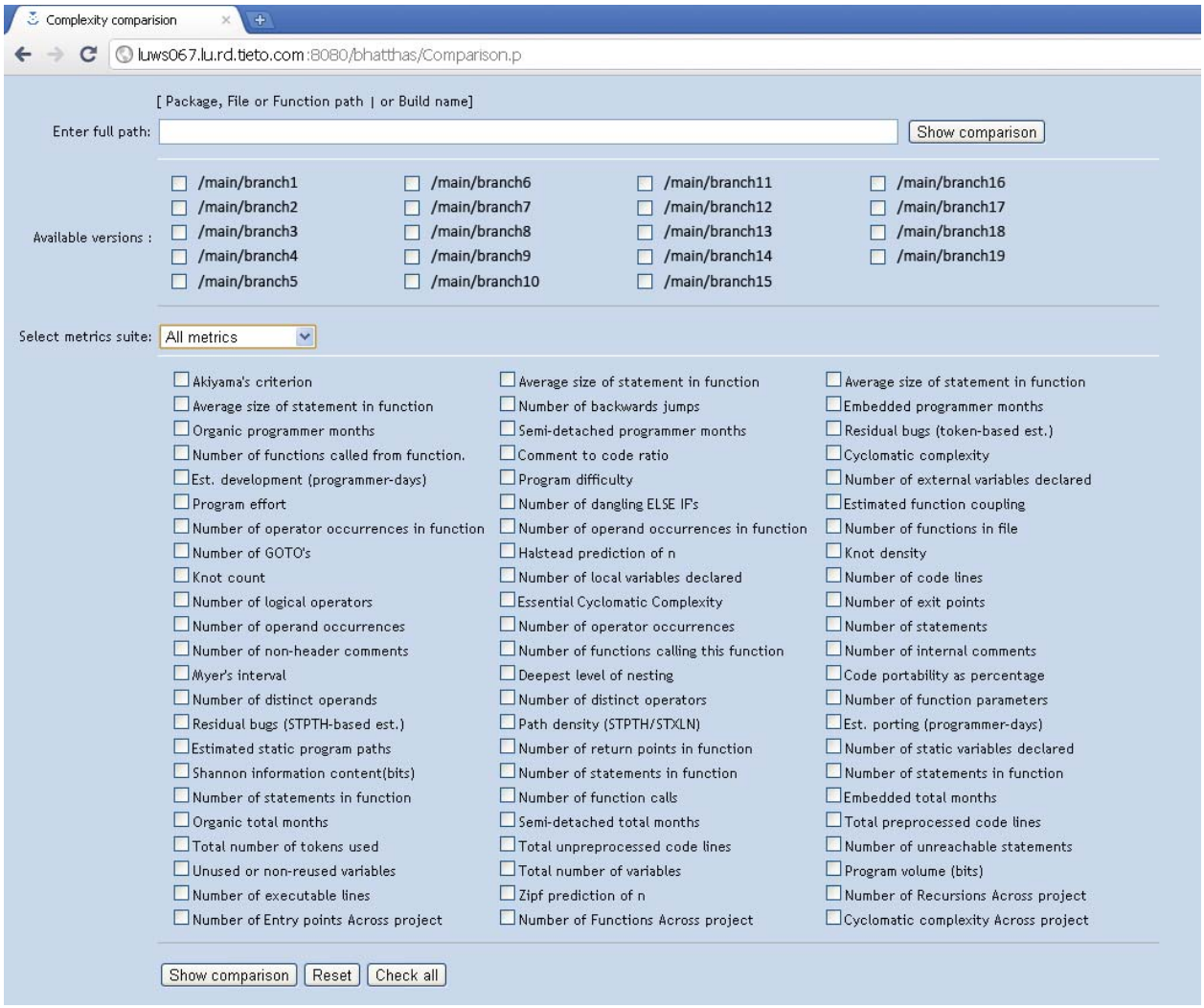


Figure 5.14 – Displaying all metric groups

In this example, the user has chosen the option to view all the metrics. User has the list of available versions for that file and list of all metrics. Depending upon the customized selection, the new comparison will be generated. There is a possibility to select all metrics but the graphs will look bad as there will be long horizontal scroll bar. But it is possible to select all metrics at once and perform a comparison.

If user only selects versions and don't select the metrics, the comparison will still be generated with the default metrics set. And if user only selects metrics and doesn’t change the version, the comparison will take the default version option and generate the comparison.

The name input field remains available, it facilitate user to enter a new file name or even package or function name! Exactly same options and visualizations exist for package and functions comparisons.

The charts of the same file1.c with customized versions and metrics are shown in the following three figures.

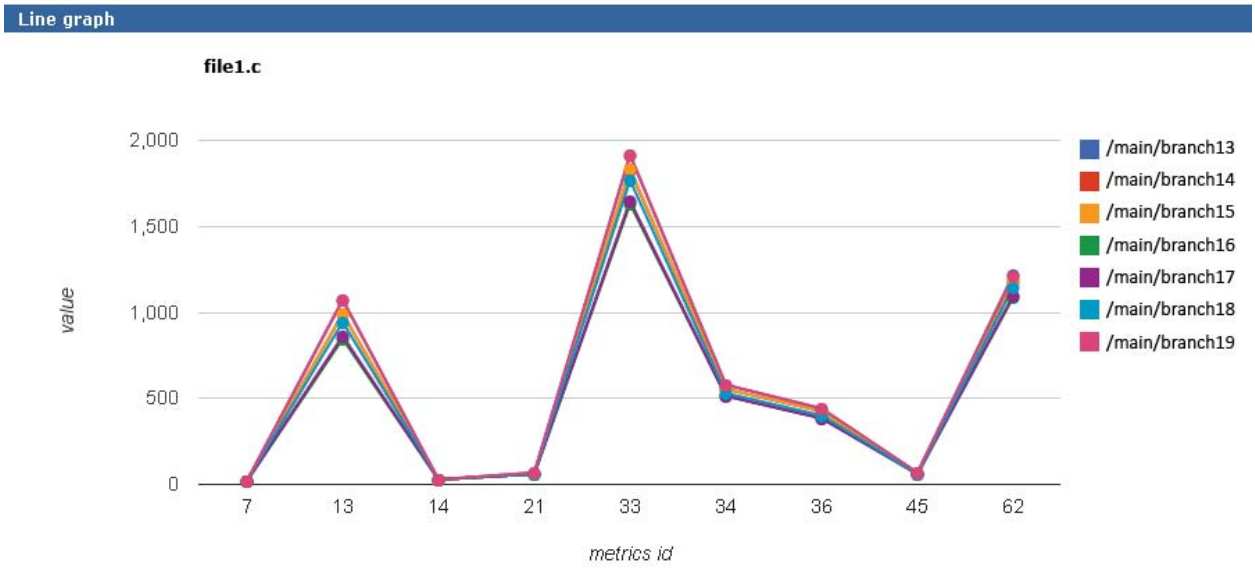


Figure 5.15 – Line graph of File1.c

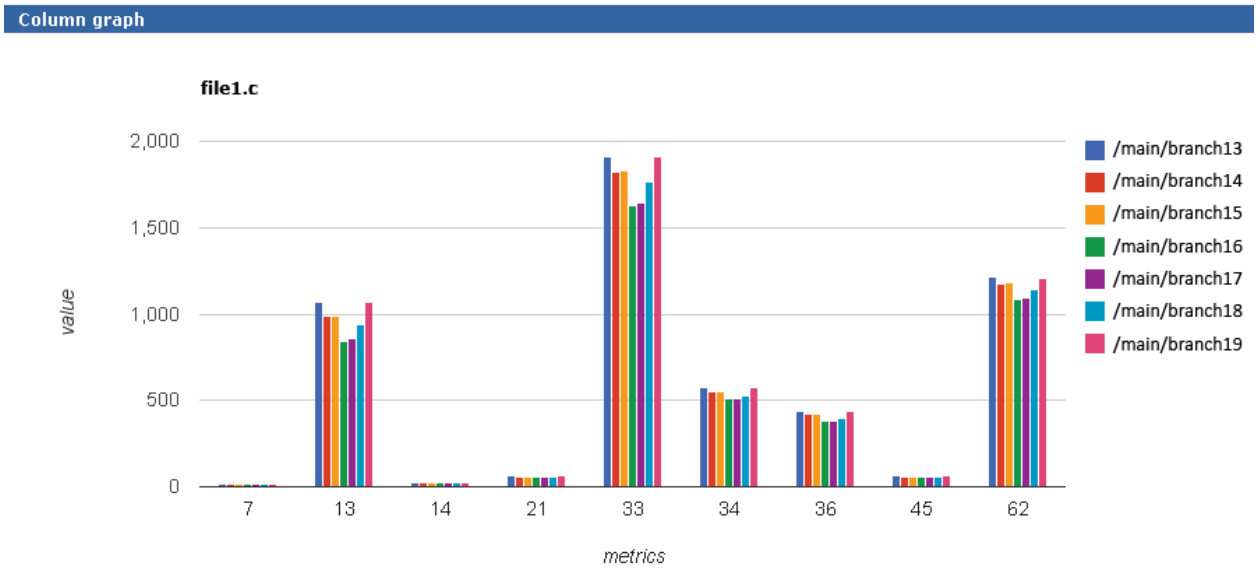


Figure 5.16 – Column graph of File1.c

Tabular data

Metrics Id	Metrics description	/main/branch13	/main/branch14	/main/branch15	/main/branch16	/main/branch17	/main/branch18	/main/branch19
7	Organic programmer months	16.36	15.58	15.63	14.15	14.23	14.99	16.37
13	Est. development (programmer-days)	1066.1	987.57	987.67	843.58	856.07	936.75	1069.28
14	Program difficulty	22.69	22.18	22.14	21.45	21.55	21.96	22.74
21	Number of functions in file	64	60	60	57	57	59	64
33	Number of statements	1911	1824	1827	1627	1644	1769	1910
34	Number of non-header comments	572	551	554	512	513	530	574
36	Number of internal comments	435	419	421	382	383	398	437
45	Est. porting (programmer-days)	62.22	59.39	59.55	54.17	54.46	57.23	62.25
62	Total number of variables	1211	1178	1182	1088	1093	1145	1209

Figure 5.17 – Line graph of File1.c

These charts show the history of the selected file development. The variations in charts represent that file1.c is continuously modified for these versions.

The comparison viewer interface is reachable from complexity viewer interface. Recall figure 5.9, the "View history" link under the package name will open the comparison viewer interface in new window, which will show the comparison of that package with default parameters as mentioned earlier. The same facility is available for files and functions too.

Chapter 6

EVALUATION & RESULTS

The outcome of the developed tool shows that

1. The tool is capable of performing code analysis automatically on regular basis. It proves that the automatic measurement of source code complexity is possible to implement.
2. This tool can be helpful for developers to view the quality of their code in terms of code metrics. Potentially fault-prone code can easily be identified which can suggest developers about the code that requires refactoring.
3. Comparison between releases helps the managers to observe the change of code between the releases. It is also possible to identify what parts of code have changed and how much they are changed.
4. The change of code may also help the testers to focus their testing efforts on those parts of code that are changed. If there are new source code files added, the McCabe cyclomatic complexity metric can still be useful for the testers to know how much test cases they need to develop.
5. The automatic logging feature allow for real-time monitoring of the tool.
6. The configuration interface allows for simple addition/alteration of configuration specification.

6.1 The Analyzed Code

A time bound fully functional evaluation copy of QA-C is used to analyze the code. Because of the license time restrictions, two software packages were selected for performing code analysis, SW_PKG1 and SW_PKG2. The selection of the source code packages were made after consulting with senior developers. The developer's opinion about SW_PKG1 was that this package is nicely designed and structured, and probably should contain less complex code. The other package SW_PKG2 was supposed to be more complex package from developer's point of view.

6.1.1 Package complexities analysis

The results of analysis performed on these packages are presented in the following figures.

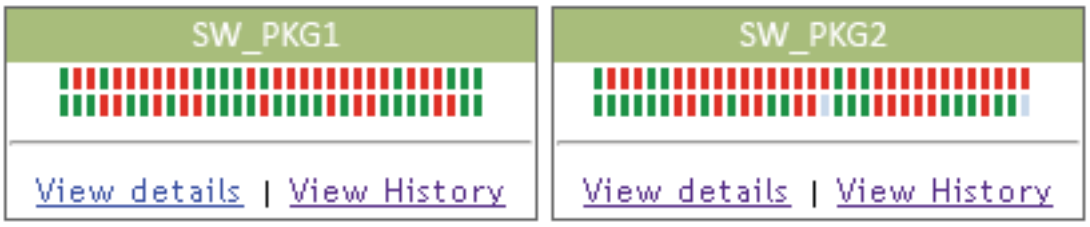


Figure 6.1 – Packages analysis

The color coding technique used in this tool for representing metrics results for a fast assessment suggests that SW_PKG1 has fewer reds than the SW_PKG2. It relates with the developer's opinion about these two packages that the SW_PKG2 is comparatively more complex.

SW_PKG1 results

Figure 6.2 – Metric results for SW_PKG1

Files and function metrics are shown separately and "thumb ups" are more in number,

which implies that the source code files of this package are mostly within the recommended complexity limits.

Note: Metrics short names/abbreviations defined by QA-C are used to present the metrics, for example STCYC is the short name for cyclomatic complexity metric. The complete names associated with these abbreviations can be found in APPENDIX D. It also includes the detail about how each metric is calculated / derived.

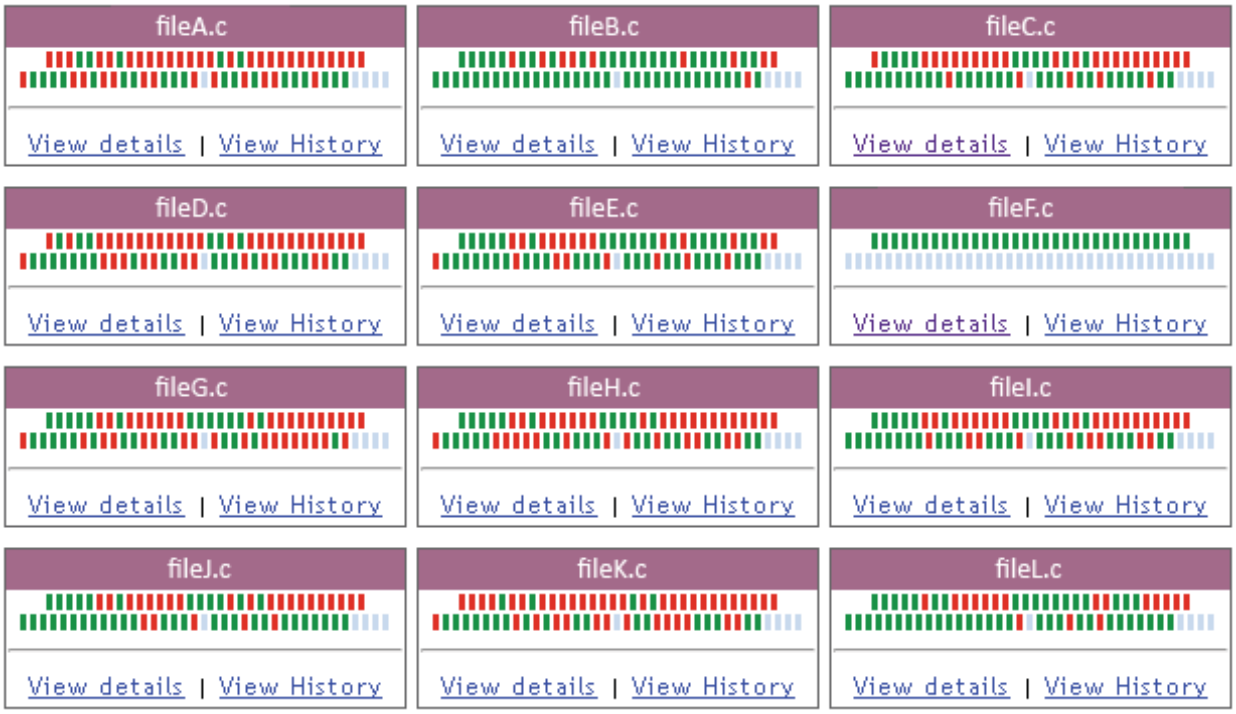


Figure 6.3 – Files of SW_PKG1

This figure shows the list of files included in SW_PKG1 and their respective color coding. It is possible to view further detail of any of these files and easy to identify the functions that are complex, and perhaps in need of refactoring.

SW_PKG2 results

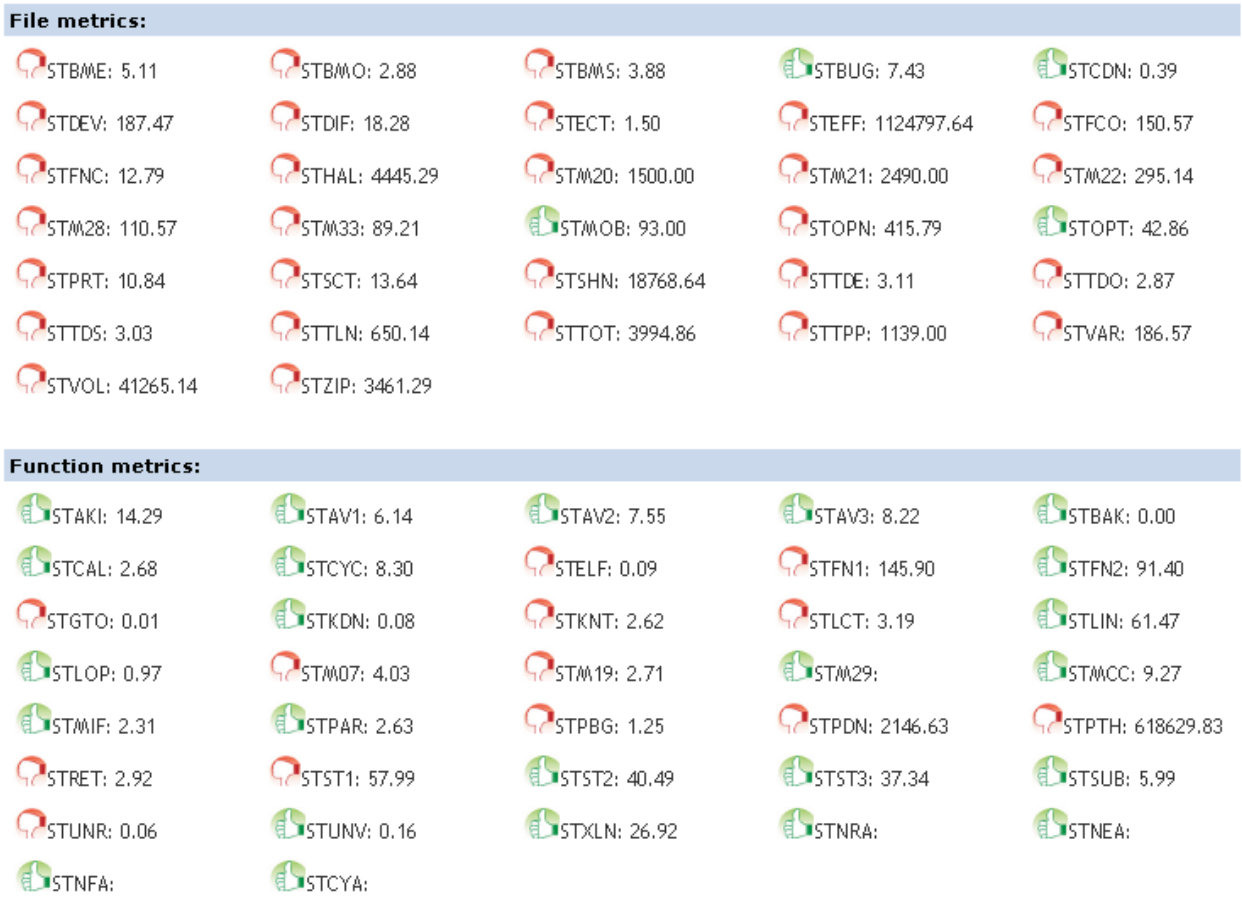


Figure 6.4 – Metric results for SW_PKG2

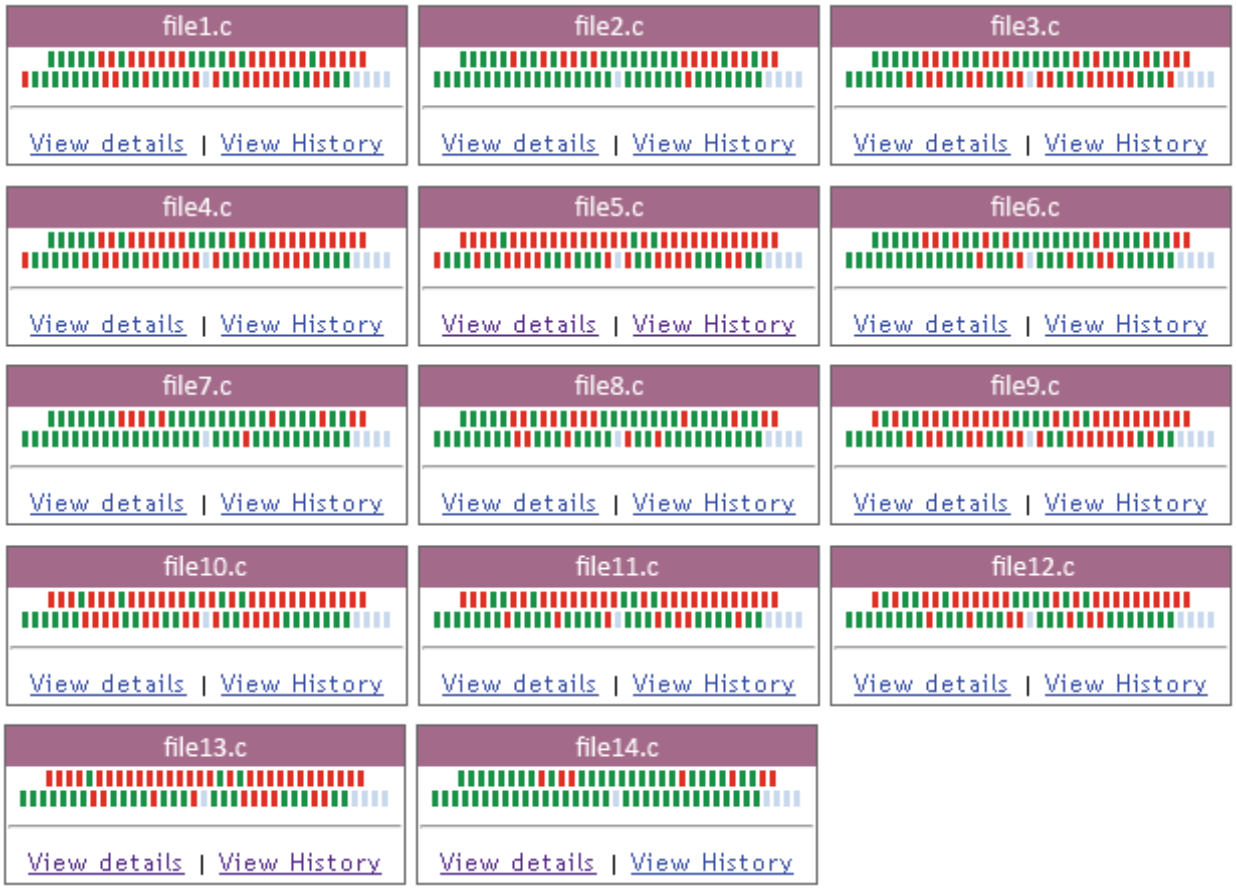


Figure 6.2 – Files of SW_PKG2

6.2 Strange Metrics

It is observed that the results of some metrics, where the values are exceeding the QA-C recommended maximum, are extremely strange and non-realistic. Examples of these strange metrics are shown in table 6.1.

Metric abbreviation	DB average value	QAC-recommended max
STEFF (Program effort)	1321976.09	2
STVOL (Program volume)	47449.36	100
STZIP (Zipg prediction of n)	3911.02	1
STSHN (Shannon information)	20974.21	50

Table 6.1 – Strange metrics

The difference between DB avg. values and the recommended values is unbelievably huge and clearly seems non-realistic. A question arises whether QA-C is gathering these metrics incorrectly or there is problem with the metric's formulas.

An idea, to verify if QA-C is gathering these metrics incorrectly, is to gather these metrics on same source code using two/three other complexity analysis tools. If results are similar then probably the fault is with the metrics formulas.

A master thesis was performed by a student, Armin Kuro, of KTH University in 2002/2003 on complexity analysis [18]. The source code analyzed in that thesis was from Scania AB, Sweden. QA-C tool was also used for code analysis and these metrics were problematic in that research too. Since QA-C was used in that research also, therefore it is still hard to identify the real cause of these huge variations.

It is important to note that all these problematic metrics are part of Halstead software science metrics. It is mentioned in chapter 3 of this report that Halstead formulas have received criticism from many scientists. Therefore, on the basis of these strange metrics values calculated in this thesis and a thesis held in Scania AB, and the scientists' opinion about Halstead metrics, it won't be unfair to say that some of Halstead formulas are problematic and misleading. However reliability of QA-C's calculations is still a subject to verification.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusions

- Software metrics can only serve as recommendations, one can't fully rely on them.
- Specifying a purpose for gathering metrics is vital to ensure relevant metrics are gathered. Large number of metrics becomes confusing if they aren't required.
- One metric might be meaningless itself, but in conjunction with another metric(s) might produce something useful.
- New custom metrics can be developed using some of the existing ones. Developers can be interviewed in order to take their opinion that how they rate the complexity of certain piece of source code, say 1 to 10. Metrics like McCabe cylomatic complexity, line of code and developer's rating can be used together to formulate a new metric.
- Metrics maximum/minimum limits should be defined depending upon the programming language, programmer’s opinion, and the specific programming requirements for the code to be analyzed.
- A successful implementation of complexity analysis for one project/organization might be unsuccessful for another project/organization.

7.2 Future Work

- The gathered metrics should more thoroughly be evaluated to find possible correlations between certain metrics.
- The results of the gathered metrics should thoroughly be verified using other metrics gathering sources/tools.
- The intermediate tool used for metrics gathering can be omitted and tool can be fully developed within the company, and not to rely on a third party tool where the functionality of the tool can change for its newer versions.
- The graphs used for showing the comparisons are generated using Google Visualization API. It requires internet connection to be able to generate the visualizations. It can be security risk for the sensitive data. An alternate approach should be adopted where visualizations are possible even without the internet connection.
- The various tasks software(s) can perform are named as 'functions' (real functions). These functions aren't the functions of a source code file. The company for which this research is performed uses this terminology. In the solution provided in this research, it is possible to see the complexity of a complete build, its packages and sub-packages, source code files and functions. But it is not possible to see which

source code files or packages relate to which 'function'. A method to map source code files to the real functions need to be determined and maybe it requires another thesis. But if this method is defined then this tool should extend its functionality to show the complexities of real functions and comparison with previous releases.

- Metrics scoring criteria and maximum/minimum limits should be redefined for the problematic metrics, like Halstead software science metrics.

REFERENCES

[1] DeMarco, Tom, "Controlling Software Projects", Yourdon Press, New York, 1986

[2] Campbell, Luke and Brian Koster, "Software Metrics: Adding Engineering Rigor to a Currently Ephemeral Process," briefing presented to the McGrummwell F/A-24 CDR course, 1995.

[3] Fenton, N., and Pfleeger, S. L. "Software Metrics - A Rigorous and Practical Approach", 2 ed. International Thomson Computer Press, London, 1996.

[4] Sam Miller, "Areas of Use for Software Metrics", 2008 [Online: accessed on 2010-06-16 from http://www.articlesbase.com/management-articles/areas-of-use-for-software-metrics-306127.html]

[5] Anton Milutin, "Software code metrics", 2009 [Online: accessed on 2010-06-21 from http://www.viva64.com/content/articles/code-analyzers/?f=Metrics.html&lang=en&content=code-analyzers]

[6] Everald E. Mills, "Software Metrics", Software Engineering Institute, 1988.

[7] Lou Marco, "Measuring software complexity", Enterprise Systems Journal, 1997.

[8] S. Henry, D. Kafura, K. Mayo, A. Yerneni, S. Wake, "A Reliability Model Incorporating Software Quality Factors", Department of Computer Science, Virginia Tech, Blacksburg.

[9] Jeremy Singer, Christos Trortjis, and Martin Ward, "Using Software Metrics to Evaluate Static Single Assignment Form in GCC", University of Manchester - UK, University of Ioannina - Greece, University of Western Macedonia - Greece.

[10] Jerry Fitzpatrick, "Applying the ABC Metric to C, C++, and Java", C++ Report, June 1997.

[11] MSDN, "Code Metrics Values", [Online: accessed on 2010-06-21 from http://msdn.microsoft.com/en-us/library/bb385914.aspx]

[12] Kurt D. Welker, Paul W. Oman, Gerald G. Atkinson, "Software Maintenance: Research and Practice Vol 9", John Wiley & Sons, Ltd, 1997.

[13] Young Lee, "Automated Source Code Measurement Environment For Software Quality", Auburn University Alabama, December 2007.

[14] M Squared Technologies, "Source Code Size Metrics", [Online: accessed on 2010-12-10 from http://msquaredtechnologies.com]

[15] T. J. McCabe, "A Complexity Measure," ICSE '76: Proceedings of the 2nd international conference on Software engineering, 1976.

[16] Complexity Metric Control, [Online: accessed on 2010-20-05 from http://gcc.gnu.org/onlinedocs/gnat_ugn_unw/Complexity-Metrics-Control.html]

[17] S. C. Johnson, "Portable C Compiler", [Online: accessed on 2010-20-05 from http://pcc.ludd.ltu.se/]

[18] Armin Krusko, "Complexity Analysis of Real Time Software", Royal Institute of Technology Sweden, 2002.

[19] Harrison, W., K. Magel, R. Kluczny, and A. DeKock, "Applying Software Complexity Metrics to Program Maintenance." Computer, 1982.

[20] Rafa E. AL QUTAISH, Alain ABRAN, "An Analysis of the Design and Definitions of Halstead's Metrics", École de Technologie Supérieure, Canada.

[21] Hamer, P. G. and Frewin, G. D., "M. H. Halstead's Software Science - A Critical Examination", in the Proceedings of the 6th International Conference on Software Engineering, Tokyo, Japan, 1982.

APPENDIX D

The Calculation of Metrics

QA C calculates metrics in three groups. Function metrics are generated for all functions with a full definition. File metrics are generated for each file analysed. Project metrics are calculated once per complete project, and are generated from cross-module analysis.

All metrics, except STLIN and STTPP, ignore code that is disabled on evaluation of pre-processor directives. That is, both the code in the source file and the `#include`'d files inside conditional pre-processor directives are ignored for all metrics, except STLIN and STTPP. See Appendix B (-ppm) for an example.

Function-Based Metrics

These metrics are all calculated for the source code of an individual function. Some metrics are used in the definition of others, but there is no obvious sequence. They are described here in alphabetical order, for ease of reference.

The complete list is as follows:

STAKI	Akiyama's Criterion
STAV1	Average Size of Function Statements (variant 1)
STAV2	Average Size of Function Statements (variant 2)
STAV3	Average Size of Function Statements (variant 3)
STBAK	Number of Backward Jumps
STCAL	Number of Distinct Function Calls
STCYC	Cyclomatic Complexity
STELF	Number of Dangling Else-If's
STFN1	Number of Function Operator Occurrences
STFN2	Number of Function Operand Occurrences
STGTO	Number of Goto's
STKDN	Knot Density
STKNT	Knot Count
STLCT	Number of Local Variables Declared
STLIN	Number of Maintainable Code Lines
STLOP	Number of Logical Operators
STM07	Essential Cyclomatic Complexity

STM19	Number of Exit Points
STM29	Number of Functions Calling this Function
STMCC	Myer's Interval
STMIF	Maximum Nesting of Control Structures
STPAR	Number of Function Parameters
STPBG	Path-Based Residual Bug Estimate
STPDN	Path Density
STPTH	Estimated Static Path Count
STRET	Number of Function Return Points
STSUB	Number of Function Calls
STST1	Number of Statements in Function (variant 1)
STST2	Number of Statements in Function (variant 2)
STST3	Number of Statements in Function (variant 3)
STUNR	Number of Unreachable Statements
STUNV	Number of Unused and Non-Reused Variables
STXLN	Number of Executable Lines

STAKI Akiyama's Criterion

This metric is the sum of the cyclomatic complexity (STCYC) and the number of function calls (STSUB). Although this is not an independent metric, it is included on account of its use in documented case histories. See Akiyama[2] and Shooman[3] for more details.

The metric is calculated as:

STAKI = STCYC + STSUB

STAVx Average Size of Function Statements

These metrics (STAV1, STAV2, and STAV3) measure the **average** number of operands and operators per statement in the body of the function.

They are calculated as follows:

STAVx = (N1 + N2) / number of statements in the function

[2] Akiyama, F. (1971) *An Example of Software System Debugging*, Proc. IFIP Congress 1971, Lbujlana, Yugoslavia, American Federation of information Processing Societies, Montvale, New Jersey.

[3] Shooman, M.L. (1983) *Software Engineering*, McGraw-Hill, Singapore.

where

N1 is Halstead's number of operator occurrences
N2 is Halstead's number of operand occurrences

The STAVx metrics are computed using STST1, STST2 and STST3 to represent the number of statements in a function. Hence there are three variants: STAV1, STAV2 and STAV3, relating to the respective statement count metrics.

This metric is used to detect components with long statements. Statements comprising a large number of textual elements (operators and operands) require more effort by the reader in order to understand them. This metric is a good indicator of the program's readability.

Metric values are computed as follows:

STAV1 = (STFN1 + STFN2) / STST1
STAV2 = (STFN1 + STFN2) / STST2
STAV3 = (STFN1 + STFN2) / STST3

STBAK Number of Backward Jumps

Jumps are never recommended and backward jumps are particularly undesirable. If possible, the code should be redesigned to use structured control constructs such as `while` or `for` instead of `goto`.

```
main()
{
    Backward:
    switch(n)
    {
    case 0:  printf(stdout, "zero\n");
        break;
    case 1:  printf(stdout, "one\n");
        goto Backward;                  /* 1 */
        break;
    case 2:  printf(stdout, "two\n");
        break;
    default: printf(stdout, "many\n");
        break;
    }
}
```

The above code sample has a STBAK value of 1.

STCAL Number of Distinct Function Calls

STCAL counts the number of function calls in a function.

It differs from the metric STSUB, in that only distinct functions are counted (multiple instances of calls to a particular function are counted as one call), and also that functions called via pointers are not counted.

STCYC Cyclomatic Complexity

Cyclomatic complexity is calculated as the number of decisions plus 1.

High cyclomatic complexity indicates inadequate modularization or too much logic in one function. Software metric research has indicated that functions with a cyclomatic complexity greater than 10 tend to have problems related to their complexity.

McCabe[4] gives an essential discussion of this issue as well as introducing the metric.

Example 1:

```
int divide(int x, int y)
{
    if (y != 0)                                 /* 1 */
    {
        return x / y;
    }
    else if (x == 0)                            /* 2 */
    {
        return 1;
    }
    else
    {
        printf("div by zero\n");
        return 0;
    }
}
```

As there are two decisions made by the function in the above sample code, it has a cyclomatic complexity of 3. Note that *correctly indented* code does not always reflect the *nesting structure* of the code. In particular, the use of the construct '`else if`' always increases the level of nesting. The construct '`else if`' is written conventionally without additional indentation, so the nesting is not apparent visually.

Example 2:

```
void how_many(int n)
```

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

```
{
    switch (n)
    {
    case 0: printf("zero");           /* 1 */
        break;
    case 1: printf("one");            /* 2 */
        break;
    case 2: printf("two");            /* 3 */
        break;
    default: printf("many");
        break;
    }
}
```

The above code sample has a cyclomatic complexity of 4, as a switch statement is equivalent to a series of decisions.

Some metrication tools include use of the ternary operator `? :` when calculating cyclomatic complexity. It could also be argued that use of the `&&` and `||` operators should be included.

STCYC is one of the three standard metrics used by QA C for demographic analysis.

STELF Number of Dangling Else-Ifs

This is the number of `if-else-if` constructs that do not end in an "`else`" clause. This metric is calculated by counting all "`if`" statements that do not have a corresponding "`else`". The warning 2004 message is issued for each instance. STELF provides a quick reference allowing monitoring of these warnings.

The code sample below has an STELF value of 1.

```
int divide(int x, int y)
{
    if (y != 0)
    {
        return x/y;
    }
    else if (x == 0)
    {
        return 1;
    }
}
```

STFN1 Number of Function Operator Occurrences

This metric is Halstead's operator count on a function basis (N1).

STFN1 is related to STOPT and STM21: all of these metrics count 'operators', the difference is summarized as:

STFN1	Counts ALL operators in the function body
STM21	Counts ALL operators in the file
STOPT	Counts DISTINCT operators in the file

See STOPT for the definition of an operator.

STFN2 Number of Function Operand Occurrences

This metric is Halstead's operand count on a function basis (N2). STFN2 is related to STOPN and STM20: all of these metrics count 'operands', the difference is summarized as:

STFN2	Counts ALL operands in the function body
STM20	Counts ALL operands in the file
STOPN	Counts DISTINCT operands in the file

See STOPN for the definition of an operand.

STGTO Number of Gotos

Some occurrences of "`goto`" simplify error handling. However, they should be avoided whenever possible. The Plum Hall Guidelines say that `goto` should not be used.

STKDN Knot Density

This is the number of knots per executable line of code. The metric is calculated as:

STKDN = STKNT / STXLN

The value is computed as zero when STXLN is zero. QA C issues warnings if the knot density exceeds certain values.

STKNT Knot Count

This is the number of knots in a function. A knot is a crossing of control structures, caused by an explicit jump out of a control structure. Knots can be produced by misuse of "`break`", "`continue`", "`goto`", or "`return`".

STKNT measures knots, not by counting control structure crossings, but by counting the following keywords:

- "got" statements
- "continue" statements within "loop" statements
- "break" statements within loop or switch statements, except those at top switch level
- all "return" statements except those at top function level

STKNT is undefined for functions with unreachable code.

The function below has an STKNT value of 1.

```
void fn( int n, int array[], int key )
{
    while ( array[ n ] != key )
    {
        if ( array[ n ] == 999 )
        {
            break;
        }
        else
        {
            ++n;
        }
    }
}
```

STLCT Number of Local Variables Declared

This is the number of local variables declared in a function of storage class auto, register, or static. These are variables that that have no linkage.

The function below has an STLCT value of 2.

```
int other_result;
extern int result;

int test()
{
    int x;                              /* 1 */
    int y;                              /* 2 */

    return (x + y + other_result);
}
```

STLIN Number of Maintainable Code Lines

This is the total number of lines, including blank lines and comment

lines, in a function definition between (but excluding) the opening and closing brace of the function body. It is computed on raw code. STLIN is undefined for functions which have `#include`'d code or macros which include braces in their definition.

The function below has an STLIN value of 5.

```
int fn()
{
    int x;                /* 1 */
    int y;                /* 2 */
                          /* 3 */
    return (x + y);       /* 4 */
    /* Comment Here */    /* 5 */
}
```

Long functions are difficult to read, as they do not fit on one screen or one listing page. An upper limit of 200 is recommended.

STLOP Number of Logical Operators

This is the total number of logical operators (`&&`, `||`) in the conditions of `do-while`, `for`, `if`, `switch`, or `while` statements in a function. The example function below has a STLOP value of 2.

```
void fn( int n, int array[], int key )
{
    while (( array[n] != key ) && ( n > 0 ))
    {
        if (( array[n] == 999) || ( array[n] == 1000 ))
        {
            break;
        }
        else
        {
            ++n;
        }
    }
}
```

STM07 Essential Cyclomatic Complexity

The essential cyclomatic complexity is obtained in the same way as the cyclomatic complexity but is based on a 'reduced' control flow graph. The purpose of reducing a graph is to check that the component complies with the rules of structured programming.

A control graph that can be reduced to a graph whose cyclomatic complexity is 1 is said to be structured. Otherwise reduction will show elements of the control graph which do not comply with the rules of structured programming.

The principle of control graph reduction is to simplify the most deeply nested control subgraphs into a single reduced subgraph. A subgraph is a sequence of nodes on the control flow graph which has only one entry and exit point. Four cases are identified by McCabe[5] which result in an unstructured control graph. These are:

- a branch into a decision structure,
- a branch from inside a decision structure,
- a branch into a loop structure,
- a branch from inside a loop structure.

However, if a component possesses multiple entry or exit points then it can not be reduced. The use of multiple entry and exit points breaks the most fundamental rule of structured programming.

The example below has a STM07 value of 4.

```
void g( int n, int pos, int force )  /* STM07 = 4
                         Cannot reduce control graph to
{                        Cyclomatic Complexity of 1 */
    int nlines = 0;
    while ( --n >= 0 )
    {
        pos = back_line( pos );
        if ( pos == 0 )
        {
            if ( ! force )
            {
                break;
            }
            ++nlines;
        }
    }
}
```

STM19 Number of Exit Points

This metric is a measure of the number of exit points in a software

[5] McCabe, T. J. (1976) *A Complexity Measure*, IEEE Transactions on Software Engineering, SE-2, pp. 308-320.

component and is calculated by counting the number of return statements. A function that has no return statements will have an STM19 value of zero even though it will exit when falling through the last statement. This is regardless of whether the function is declared to have a return value or not (i.e. returns void). Exits by special function calls such as `exit()` or `abort()` are ignored by this metric.

The example below has an STM19 value of 3.

```
void f( int a )
{
   return;           /* 1 */
   if ( a ) return;  /* 2 */
   a++;
   return;           /* 3 */
}
```

STM29 Number of Functions Calling this Function

This metric is defined as the number of functions calling the designated function. The number of calls to a function is an indicator of criticality. The more a function is called, the more critical it is and, therefore, the more reliable it should be.

STMCC Myer's Interval

This is an extension to the cyclomatic complexity metric. It is expressed as a pair of numbers, conventionally separated by a colon. Myer's Interval is defined as STCYC : STCYC + L

Cyclomatic complexity (STCYC) is a measure of the number of decisions in the control flow of a function. `L` is the value of the QA C STLOP metric which is a measure of the number of logical operators (`&&`, `||`) in the conditional expressions of a function. A high value of `L` indicates that there are many compound decisions, which makes the code more difficult to understand. A Myer's interval of 10 is considered very high.

The example below has a STMCC value of 3:4 because the cyclomatic complexity is 3 and there is one connective (`&&`) used in the conditions.

```
int divide(int x, int y)
{
    if (y != 0)                /* Condition 1 */
```

```
    {
        return x / y;
    }
    else if (x == 0 && y > 2)  /* Condition 2 */
                               /* Conditional expr 1 */
    {
        return 1;
    }
    else
    {
        printf("div by zero\n");
        return 0;
    }
}
```

Note:
In the calculation of STMCC, the ternary operator (`?:`) is ignored.

When exporting metric values or displaying in the metrics browser, rather than attempt to display a value pair the value of L is chosen for STMCC.

STMIF Maximum Nesting of Control Structures

This metric is a measure of the maximum control flow nesting in your source code.

You can reduce the value of this metric by turning your nesting into separate functions. This will improve the readability of the code by reducing both the nesting and the average cyclomatic complexity per function.

The code example below has an STMIF value of 3.

```
int divide(int x, int y)
{
    if (y != 0)                          /* 1 */
    {
        return (x/y);
    }
    else if (x == 0)                     /* 2 */
    {
        return 1;
    }
    else
    {
        printf("Divide by zero\n");
        while(x > 1)                     /* 3 */
            printf("x = %i", x);
        return 0;
```



```
        }
    }
```

STMIF is incremented in "switch", "do", "while", "if" and "for" statements. The nesting level of code is not always visually apparent from the indentation of the code. In particular, an '`else if`' construct increases the level of nesting in the control flow structure, but is conventionally written without additional indentation.

STMIF is one of the three standard metrics used by QA C for demographic analysis.

STPAR Number of Function Parameters

This metric counts the number of declared parameters in the function argument list. Ellipsis parameters are ignored.

STPBG Path-Based Residual Bug Estimate

Hopkins, in Hatton & Hopkins[6] investigated software with a known audit history and observed a correlation between Static Path Count (STPTH) and the number of bugs that had been found. This relationship is expressed as STPBG.

$STPBG = \log_{10}(STPTH)$

STPDN Path Density

This is a measure of the number of paths relative to the number of executable lines of code.

STPDN = STPTH / STXLN

STPDN is computed as zero when STXLN is zero.

STPTH Estimated Static Path Count

This is similar to Nejmeh's[7] NPATH statistic and gives an upper bound on the number of possible paths in the control flow of a function. It is the number of non-cyclic execution paths in a function.

The NPATH value for a sequence of statements at the same nesting

[6] Hatton, L., Hopkins, T.R., (1989) *Experiences With Flint, a Software Metrication Tool for Fortran 77*, Symposium on Software Tools, Napier Polytechnic, Edinburgh, Scotland.

[7] Nejmeh, B.A. (1988), *NPATH: A Measure of Execution Path Complexity and its Applications*, Comm ACM, 31, (2), p. 188-200.

level is the product of the NPATH values for each statement and for the nested structures. NPATH is the product of:

- NPATH(sequence of non control statements) = 1
- NPATH(`if`) = NPATH(body of `then`) + NPATH(body of `else`)
- NPATH(`while`) = NPATH(body of `while`) + 1
- NPATH(`do while`) = NPATH(body of `while`) + 1
- NPATH(`for`) = NPATH(body of `for`) + 1
- NPATH(`switch`) = Sum(NPATH(body of `case 1`) … NPATH(body of `case n`))

Note:
`else` and `default` are counted whether they are present or not.

In `switch` statements, multiple case options on the same branch of the `switch` statement body are counted once for each independent branch only. For example:

```
switch( n )
{
    case 0 : break;    /* NPATH of this branch is 1   */
    case 1 :
    case 2 : break;    /* NPATH for case 1 & case 2   */
                       /* combined is 1               */
    default: break;    /* NPATH for this default is 1 */
}
```

NPATH cannot be computed if there are "`goto`" statements in the function.

The following code example has a static path count of 26.

```
int n;
if ( n )
{
} /* block 1, paths 1 */

else if ( n )
{
    if ( n )
    {
    } /* block 2, paths 1 */

    else
    {
    } /* block 3, paths 1 */
```



```
    /* block 4, paths block2+block3 = 2 */

    switch ( n )
    {
    case 1 : break;
    case 2 : break;
    case 3 : break;
    case 4 : break;
    default: break;
    } /* block 5, paths = 5 */

} /* block 6, paths block4*block5 = 10 */

else
{
    if ( n )
    {
    } /* block 7, paths 1 */

    else
    {
    } /* block 8, paths 1 */
} /* block 9, paths block7+block8 = 2 */
/* block 10, paths block1+block6+block9 = 13 */

if ( n )
{
} /* block 11, paths 1 */

else
{
} /* block 12, paths 1 */
/* block 13, paths block11+block12 = 2 */

/* outer block, paths block10*block13 = 26 */
```

Each condition is treated as disjoint. In other words, no conclusions are drawn about a condition that is tested more than once.

The true path count through a function usually obeys the inequality: Cyclomatic complexity ≤ true path count ≤ static path count

Static path count is one of the three standard metrics used by QA C for demographic analysis.

STRET Number of Function Return Points

STRET is the count of the reachable return statements in the function, plus one if there exists a reachable implicit return at the } that terminates the function.

The following example shows an implicit return:

```
void foo( int x, int y )
{
   printf( "x=%d, y=%d\n", x, y );
/* Here with implicit return.  Hence STRET = 1*/
}
```

Structured Programming requires that every function should have exactly one entry and one exit. This is indicated by a STRET value of 1. STRET is useful when the programmer wants to concentrate on functions that do not follow the Structured Programming paradigm, for example those with switch statements with returns in many or every branch.

STSTx Number of Statements in Function

These metrics count the number of statements in the function body. There are 3 variants on the metric:

STST1 is the base definition and counts all statements tabulated below.
STST2 is STST1 except block, empty statements and labels are not counted.
STST3 is STST2 except declarations are not counted.

The following chart shows the statements counted by the STST metrics:

Statement Kind	STST1	STST2	STST3
block	yes	ignore	ignore
simple statement followed by ;	yes	yes	yes
empty statement	yes	ignore	ignore
declaration statement	yes	yes	ignore
label	yes	ignore	ignore
break	yes	yes	yes
continue	yes	yes	yes
do	yes	yes	yes
for	yes	yes	yes
goto	yes	yes	yes
if	yes	yes	yes
return	yes	yes	yes
switch	yes	yes	yes
while	yes	yes	yes

The following example shows statements counted by STST1, which

for this function yields a value of 10:

```
void stst( void )
{
   int i;              /* 1 */
label_1:               /* 2 */
label_2:               /* 3 */
   switch ( 1 )        /* 4 */
   {                   /* 5 */
      case 0:          /* 6 */
      case 1:          /* 7 */
      case 2:          /* 8 */
      default:         /* 9 */
         break;        /* 10 */
   }
}
```

This metric indicates the maintainability of the function. Number of statements also correlates with most of the metrics defined by Halstead. The greater the number of statements contained in a function, the greater the number of operands and operators, and hence the greater the effort required to understand the function. Functions with high statement counts should be limited. Restructuring into smaller sub-functions is often appropriate.

STSUB Number of Function Calls

The number of function calls within a function. Functions with a large number of function calls are more difficult to understand because their functionality is spread across several components. Note that the calculation of STSUB is based on the number of function calls and not the number of distinct functions that are called.

A large STSUB value may be an indication of poor design; for example, a calling tree that spreads too rapidly. See Brandl (1990)[8] for a discussion of design complexity and how it is highlighted by the shape of the calling tree.

The following code example has an STSUB value of 4.

```
extern dothis(int);
extern dothat(int);
extern dotheother(int);

void test()
{
```

[8] Brandl, D.L. (1990), *Quality Measures in Design*, ACM Sigsoft Software Engineering Notes, vol 15, 1.

```
    int a, b;
    a = 1;
    b = 0;

    if (a == 1)
    {
        dothis(a);                  /* 1 */
    }
    else
    {
        dothat(a);                  /* 2 */
    }

    if (b == 1)
    {
        dothis(b);                  /* 3 */
    }
    else
    {
        dotheother(b);              /* 4 */
    }
}
```

STUNR Number of Unreachable Statements

This metric is the count of all statements within the function body that are guaranteed never to be executed. STUNR uses the same method for identifying statements as metric STST1. Hence STUNR counts the following as statements if unreachable.

Statement Kind Counted
block
simple statement followed by ;
empty statement
declaration statement
label
break
continue
do
for
goto
if
return
switch
while

Example yielding STUNR = 4

```
void stunr( unsigned i )
{
   if ( i >= 0 )
   {
      if ( i >= 0 )
      {
         while ( i >= 0 ) return;
      }
      else
      /* unreachable */
      {
         while ( i >= 0 ) return;
      }
   }
   return;  /* unreachable */
}
```

STUNV Unused and Non-Reused Variables

An **unused** variable is one that has been defined, but which is never referenced. A **non-reused** variable is a variable that has a value by assignment, but which is never used subsequently.

Such variables are often indications of the effects of different programmers making changes over time – "ageing software". The code below has an STUNV value of 2.

```
int other_result;
extern int result;

int test()
{
    int y;                      /* Unused */
    int z;

    z = 1;                      /* Non-Reused */

    return (result + other_result);
}
```

STXLN Number of Executable Lines

This is a count of lines in a function body that have code tokens. Comments, braces, and all tokens of declarations are not treated as code tokens.

STXLN is used in the computation of the STKDN and STPDN metrics.

The function below has an STXLN value of 9.

```
void fn( int n )
{
    int x;
    int y;

    if ( x )                        /* 1 */
    {
        x++;                        /* 2 */
        for (;;)                    /* 3 */
            /* Ignore comments */
            /* Ignore braces */
        {
            switch ( n )            /* 4 */
            {
                case 1 : break;     /* 5 */
                case 2 :            /* 6 */
                case 3 :            /* 7 */
                          break     /* 8 */
                                ;   /* 9 */
            }
        }
    }
}
```


File-Based Metrics

These metrics are all calculated for the source code in a complete translation unit. They are listed alphabetically here for convenience. They are not calculated in this sequence, because of their derivation from each other.

STBME	COCOMO Embedded Programmer Months
STBMO	COCOMO Organic Programmer Months
STBMS	COCOMO Semi-detached Programmer Months
STBUG	Residual Bugs (token-based estimate)
STCDN	Comment o Code Ratio
STDEV	Estimated Development Time
STDIF	Program Difficulty
STECT	Number of External Variables
STEFF	Program Effort
STFCO	Estimated Function Coupling
STFNC	Number of Function Definitions
STHAL	Halstead Prediction Of STTOT
STM20	Number of Operand Occurrences
STM21	Number of Operator Occurrences
STM22	Number of Statements
STM28	Number of Non-Header Comments
STM33	Number of Internal Comments
STMOB	Code Mobility
STOPN	Halstead Distinct Operands
STOPT	Halstead Distinct Operators
STPRT	Estimated Porting Time
STSCT	Number of Static Variables
STSHN	Shannon Information Content
STTDE	COCOMO Embedded Total Months
STTDO	COCOMO Organic Total Months
STTDS	COCOMO Semi-detached Total Months
STTLN	Total Preprocessed Source Lines
STTOT	Total Number of Tokens
STTPP	Total Unpreprocessed Source Lines
STVAR	Number of Identifiers
STVOL	Program Volume
STZIP	Zipf Prediction of STTOT

STBMO COCOMO Organic Programmer Months

STBMS COCOMO Semi-detached Programmer Mths

STBME COCOMO Embedded Programmer Months

The COCOMO metrics are produced for each source code file. You can display an accurate estimate of development costs for a whole project by choosing the COCOMO Cost Model option from the Reports menu. See Chapter 6: Reports for a discussion of the COCOMO cost model and an explanation of the three programming modes: Organic, Semi-detached, and Embedded.

These metrics estimate the number of programmer-months required to create the source code in the respective modes.

STBME = 3.6 * (STTPP / 1000) 1.20
STBMO = 2.4 * (STTPP / 1000) 1.05
STBMS = 3.0 * (STTPP / 1000) 1.12

STBUG Residual Bugs (token-based estimate)

$STBUG = 0.001 * STEFF^{2/3}$

This is an estimate of the number of bugs in the file, based on the number of estimated tokens. Its value would normally be lower than the sum of the function-based STPBG values. For a more detailed discussion of software bug estimates, see Hatton and Hopkins[9].

STCDN Comment to Code Ratio

This metric is defined to be the number of visible characters in comments divided by the number of visible characters outside comments. Comment delimiters are ignored. Whitespace characters in strings are treated as visible characters.

A large value of STCDN indicates that there **may** be too many comments, which can make a module difficult to read. A small value indicates that there **may** not be enough comments, which can make a module difficult to understand.

The code below has 28 visible characters in comments and 33 visible characters in the code. The resulting STCDN value is 0.85. However,

[9] Hatton, L., Hopkins, T.R., (1989) *Experiences With Flint, a Software Metrication Tool for Fortran 77*, Symposium on Software Tools, Napier Polytechnic, Edinburgh, Scotland.

your judgement may be that, in this particular example, any comments are superfluous. STCDN is a useful guide; it need not be treated as prescriptive.

```
int test()
/* This is a test */
{
    int x;
    int y;
/* This is another test */
    return (x + y);
}
```

The value of STCDN is affected by how QA C counts the comments. QA C can count comments in three possible ways:

- all comments, (a)
- all comments except for those from headers, (n)
- inline or internal comments (i). These are comments within functions and comments that annotate a line of code (comments that are on the same line as code at file scope).

You can determine which counting method is used in Comment Count on the Metrics tab of the Analyser Personality or by setting the `-co` option on the command line.

STDEV Estimated Development Time

This is an estimate of the number of programmer days required to develop the source file. Unlike COCOMO statistics, which are based solely on the number of lines of code, this estimate is derived from the file's difficulty factor. It is a more accurate measure of the development time, especially after the scaling factor has been adjusted for a particular software environment.

STDEV = STEFF / dev_scaling

where `dev_scaling` is a scaling factor defined in `qac.cfg`. The default is 6000.

STDIF Program Difficulty

This is a measure of the difficulty of a translation unit. An average C program has a difficulty of around 12. Anything significantly above this has a rich vocabulary and is potentially difficult to understand.

$$STDIF = STVOL / ((2 + STVAR) * \log_2 (2 + STVAR))$$

STECT Number of external variables

STECT measures the number of data objects (not including functions) declared with external linkage. STECT is an indication of the amount of global data being passed between modules. It is always desirable to reduce dependence on global data to a minimum.

```
extern int result;
int other_result;

main()
{
    result = 20 + 30;
    other_result = result * 2;
}
```

The above code sample has an STECT value of 2.

STEFF Programmer Effort

This metric is a measure of the programmer effort involved in the production of a translation unit. It is used to produce a development time estimate.

STEFF = STVOL * STDIF

STFNC Number of Function Definitions

This metric is a count of the number of function definitions in the file.

STFCO Estimated Function Coupling

See Brandl[10]. Since the actual value of Brandl's metric requires a full, well-structured calling tree, STFCO can only be an estimate. A high figure indicates a large change of complexity between levels of the calling tree. The metric is computed as follows from STFNC and the STSUB values of the component functions in the translation unit:

$$STFCO = \sum(STSUB) - STFNC + 1$$

The code example below has an STFCO value of 1 (2 – 2 + 1).

```
BOOL isActive(CHANNEL c);
```

[10] Brandl, D.L. (1990), *Quality Measures in Design*, ACM Sigsoft Software Engineering Notes, vol 15, 1.


```
BOOL okToRead(TEXTCHANNEL c)
{
    return !isActive(c);
}

BOOL okToPrint(PRINTCHANNEL c)
{
    return !isActive(c);
}
```

STHAL Halstead Prediction of STTOT

STHAL and STZIP are predictions (derived from the vocabulary analysis metrics STOPN and STOPT) of what the value of STTOT should be. If they differ from STTOT by more than a factor of 2, it is an indication of an unusual vocabulary. This usually means that either the source code contains sections of rather repetitive code or it has an unusually rich vocabulary. The two metrics are computed as follows:

STZIP = (STOPN+STOPT) * (0.5772+ln (STOPN+STOPT))
STHAL = STOPT * $\log_2$(STOPT) + STOPN * $\log_2$(STOPN)

QA C checks the average of STZIP and STHAL against STTOT and issues a warning for unusual vocabularies.

STM20 Number of Operand Occurrences

This metric is the number of operands in a software component and is one of the Halstead vocabulary analysis metrics. Halstead considered that a component is a series of tokens that can be defined as either operators or operands.

Unlike STOPN, this metric is the count of every instance of an operand in a file, regardless of whether or not it is distinct. STOPN only counts the operands that are distinct. The code example below has a STM20 value of 8.

```
void f( int a )         /* 1,2 -> f, a      */
{
   if ( a > 1 )         /* 3,4 -> a, 1      */
   {
      ++a;              /* 5 -> a           */
      while ( a > 1 )   /* 6,7 -> a, 1      */
      {
         --a;           /* 8 -> a           */
      }
   }
}
```

STM21 Number of Operator Occurrences

This metric is the number of operators in a software component and is one of the Halstead vocabulary analysis metrics. Halstead considered that a component is a series of tokens that can be defined as either operators or operands.

Unlike STOPT, this metric is the count of every instance of an operator in a file, regardless of whether or not it is distinct. STOPT only counts the operators that are distinct. The code example below has a STM21 value of 22.

```
void f( int a )         /* 1,2,3,4 -> void, (, int, )     */
{                       /* 5 -> {                         */
    if ( a > 1)         /* 6,7,8,9 -> if, (, >, )         */
    {                   /* 10 ->{                         */
        ++a;            /* 11,12 -> ++, ;                 */
        while (a > 1)   /* 13,14,15,16 -> while, (, >, )  */
        {               /* 17 -> {                        */
            --a;        /* 18,19 -> --, ;                 */
        }               /* 20 -> }                        */
    }                   /* 21 -> }                        */
}                       /* 22 -> }                        */
```

STM22 Number of Statements

This metric is the number of statements in a software component. This is a count of semicolons in a file except for the following instances:

- within `for` expressions,
- within `struct` or `union` declarations/definitions,
- within comments,
- within literals,
- within preprocessor directives,
- within old-style C function parameter lists.

The code example below has a STM22 value of 5.

```
void f( int a )

{
   struct { int i;
            int j; }
          ij;           /* 1          */
   a = 1;               /* 2          */
   a = 1; a = 2;        /* 3,4        */
   if
```



```
        ( a >
              1 )
    {
        return;          /* 5         */
    }

}
```

STM28 Number of Non-Header Comments

This metric is a count of the occurrences of C or C++ style comments in a source file, except for those that are within the header of a file. These comments are ones that appear after the first code token or preprocessor directive token. Comments within the file header are considered to be: all comments in a file that precede the first code token.

STM28 is based on the method used to compute STCDN but differs from STCDN in that STCDN counts the visible characters within comments whereas STM28 counts the occurrences of comments. The code example below has a STM28 value of 2.

```
/* Header comment 1       Count = 0 */
/* Header comment 2       Count = 0 */

/* Last header comment    Count = 0
   code follows */

#define CENT 100

/* Non header comment     Count = 1 */
void f( int a )
{
  /* Block scope comment  Count = 2 */
   a = CENT;
   return;
}
```

STM33 Number of Internal Comments

This metric is a count of C style or C++ comments in a source file that are within functions or annotate a line of code at file scope. Comments within functions are all comments at block scope. Comments that annotate code are ones that start or end on the same line as code.

STM33 is based on the method used to compute STCDN but differs from STCDN in that STCDN counts the visible characters within comments whereas STM33 counts the occurrences of comments. The

code example below has a STM33 value of 5.

```
/* Header comment 1       Count = 0 */
/* Header comment 2       Count = 0 */

/* Last header comment    Count = 0
   code follows */

#define CENT 100  /* Annotating comment  STM33 = 1 */

int               /* Annotating comment  STM33 = 2 */
    i;

/* Annotating comment STM33 = 3 */ int j; /*
   Annotating comment STM33 = 4 */
/* Non internal comment                  STM33 = 0 */
void f( int a )
{
  /* Block scope comment                 STM33 = 5 */
   a = CENT;
   return;
}
```

STMOB Code Mobility

STMOB = 100 * (STDEV - STPRT) / STDEV

This metric is a measure of the code portability as a percentage of the development time. In practice, most C code achieves over 95% portability; but many portability issues are caused, not by the language itself, but by the library support available on a given machine.

STOPN Halstead Distinct Operands

This is the number of distinct operands used in the file. Distinct operands are defined as: unique identifiers, and **each occurrence** of a literal.

Most literals, except 0 and 1, are usually distinct within a program. Since macros are usually used for fixed success and failure values (such as TRUE and FALSE), the differences in counting strategies are fairly minimal.

The code below has an STOPN value of 11.

```
extern int result;              /* 1 -> result */
static int other_result;        /* 2 -> other_result */

main()                          /* 3 -> main */
{
```



```
  int x;                          /* 4 -> x */
  int y;                          /* 5 -> y */
  int z;                          /* 6 -> z */

  x = 45;                         /* 7 -> 45 */
  y = 45;                         /* 8 -> 45 */
  z = 1;                          /* 9 -> 1 */
  result = 1;                     /* 10 -> 1 */
  other_result = 0;               /* 11 -> 0 */

  return (x + other_result);

}
```

STOPT Halstead Distinct Operators

This covers any source code tokens not supplied by the user, such as keywords, operators, and punctuation. STOPT is used in the calculation of a number of other metrics.

The code below has an STOPT value of 11.

```
extern int result;          /* 1,2,3 -> extern, int, ; */

static int other_result;    /* 4 -> static */

main()                      /* 5,6 -> ()*/
{                           /* 7 -> { */
    int x;
    int y;
    int z;

    x = 45;                 /* 8 -> = */
    y = 45;
    z = 1;
    result = 1;
    other_result = 0;

    return (x + other_result); /* 9,10 -> return, + */

}                           /* 11 -> } */
```

STPRT Estimated Porting Time

This is an estimate of the number of programmer days required to port the source file. It is based on the number of warnings issued by QA C, assuming that there are upper limits and lower limits to how much code can be ported per day and work correctly. The limits and the porting scaling factor are configurable in `qac.cfg`.

Three calculations are made to determine the STPRT metric:

1. A portability scale factor is calculated:

   ```
   port = weighting / 100 * port_scaling
   ```

   `weighting` is the sum of the portability weightings of all the warnings generated by QA C.

   `port_scaling` is a portability scaling factor defined in `qac.cfg`.

   If port is greater than 1.0, it will be reset to 1.0.

2. The porting rate is calculated in lines per day:

   ```
   lpd = min_port + ( 1 - port ) * max_port
   ```

   `min_port` minimum porting rate

   `max_port` maximum porting rate

3. The actual porting time is calculated:

   STPRT = STTPP / `lpd`

STSCT Number of static variables

This metric is computed as the number of variables and functions declared static at file scope. The code example below has an STSCT value of 2.

```
static int other_result;     /* 1 */
int result;

static int test()            /* 2  */
{
    int x;
    int y;
    int z;

    z = 1;

    return (x + other_result);
}
```

STSHN Shannon Information Content

Also known as the "entropy" H, this metric is a widely recognized algorithm for estimating the space required to encode the functions of a source file. STSHN is measured in bits. It is calculated as follows:

$STSHN = STZIP * \log_2 (\sqrt{(STOPN+STOPT)} + \ln (STOPN+STOPT))$

STTDE Embedded Total Months

STTDO Organic Total Months

STTDS Semi-detached Total Months

The COCOMO metrics are produced for each source code file. You can display an accurate estimate of development costs for a whole project by choosing the COCOMO Cost Model option from the Reports menu. Refer to Chapter 6: Reports for a discussion of the COCOMO cost model and an explanation of the three programming modes, Organic, Semi-detached, and Embedded.

These metrics are a measure of the total number of months required to develop the source code in the respective environments.

$STTDE = 2.5 * STBME^{0.32}$
$STTDO = 2.5 * STBMO^{0.38}$
$STTDS = 2.5 * STBMS^{0.35}$

STTLN Total Preprocessed Code Lines

This metric is a count of the total amount of lines in the translation unit after pre-processing. The pre-processed file will reflect the processing of include files, pre-processor directives and the stripping of comment lines.

STTOT Total number Of Tokens used

This metric is the total number of tokens, not distinct tokens, in the source file. The code example below has an STTOT value of 19.

```
int test()                      /* 1,2,3,4 */
{                               /* 5 */
    int x;                      /* 6,7,8
    int y;                      /* 9,10,11 */
    /*Excluded Comment*/
    return (x + y);             /* 12,13,14,15,16,17,18
*/
}                               /* 19 */
```

STTPP Total Unpreprocessed Source Lines

This metric is a count of the total number of source lines in the file

before pre-processing.

STVAR Number of identifiers

This metric represents the total number of distinct identifiers. The code below has an STVAR value of 5.

```
int other_result;                       /* 1 */
extern int result;                      /* 2 */

int test()                              /* 3 */
{
    int x;                              /* 4 */
    int y;                              /* 5 */

    return (x + y + other_result);
}
```

STVOL Program Volume

This is a measure of the number of bits required for a uniform binary encoding of the program text. It is used to calculate various Halstead vocabulary metrics.

The following is the calculation for the program volume:

STVOL = STTOT * $\log_2$ (STOPN+ STOPT)

STZIP Zipf Prediction of STTOT

STZIP = (STOPN+STOPT) * (0.5772+ln (STOPN+STOPT))

See STHAL.

Project-Wide Metrics

These metrics are calculated once per complete project. The complete list is as follows:

STNRA	Number of Recursions Across Project
STNEA	Number of Entry Points Across Project
STNFA	Number of Functions Across Project
STCYA	Cyclomatic Complexity Across Project

STNRA Number of Recursions Across project

This metric is defined to be the number of recursive paths in the call graph of the project's functions. A recursive path can be for one or more functions. The minimum, and often desirable, value of this metric is zero. Values greater than zero indicate the number of distinct loops in the call graph.

STNEA Number of Entry points Across project

This metric is the number of functions that are not called in the project. It is the number of nodes in the call graph which are not themselves called. For example, main() is not called; hence the minimum (target) value of 1.

STNFA Number of Functions Across project

This metric is the number of function definitions in the project. Note that this metric is the sum of QAC STFNC metric values for each source file included in the project.

STCYA Cyclomatic Complexity Across project

This metric is the sum of cyclomatic complexity values for each function definition in the project. Note that this metric is the sum of QA C STCYC metric values for each source file included in the project.

