The Use of Complexity Metrics in Testing CA Gen Projects

Abstract

Projects that do not have sufficient testing resources to perform 100% testing of new and changed programs may wish to focus their testing efforts on programs that are more likely to contain errors.

Simple statement count metrics are compared with Cyclomatic Complexity counts and these are correlated with historical data from a large Gen application to understand the relationship between these metrics and the number of changes that have affected each program.

The results indicate that Cyclomatic Complexity is a useful indicator of the probability that a program will contain errors and hence a useful metric to use in prioritising testing effort.

Introduction

The objective of this study is to investigate the usefulness of complexity metrics in managing the testing of CA Gen developed applications.

IET have developed a code coverage testing tool (pathvIEw), which measures how complete the testing is for each program (action block) based on the percentage of statements that have been executed during testing.

Ideally the testing of the application would ensure 100% test coverage of all changed programs, i.e. every statement in the code is thoroughly tested. However this can often not be achieved, especially without using comprehensive automated testing software and associated test scripts, and when testing resources are limited.

For situations where testers wish to focus their testing efforts, this study aims to establish whether there are any metrics that would indicate which programs are more likely to contain errors

This study will also be comparing results of various methods of calculating complexity and the differences between them and the benefits of each one, as well as less established methods that have been suggested as more useful.

Complexity Metrics

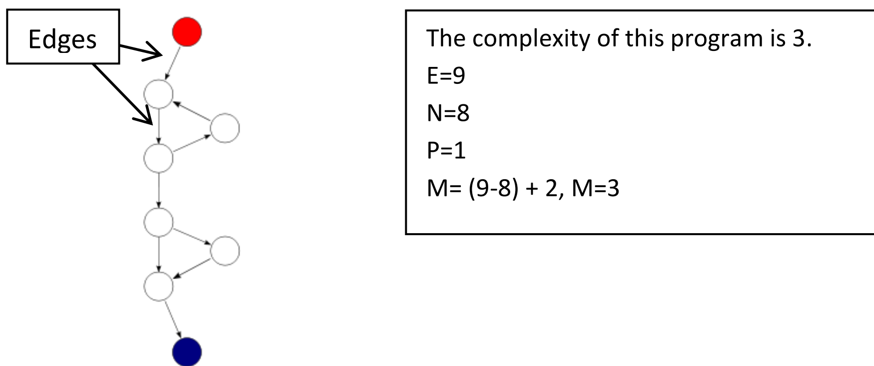
A complexity metric produces a measure of how complex a program is, based on factors such as lines of code and types of statement.

The simplest metric is *Lines of Code* (LOC) which is a simple count of the number of lines in the source program. Whilst often used because it is so simple to calculate, and perhaps modified to only count logical lines of code (i.e. ignoring comments), there are many disadvantages of LOC, for example, it is dependent on coding style and each type of statement has equal weighting. For CA Gen action blocks, number of statements is the nearest equivalent to LOC and is perhaps more accurate because of the highly controlled structure of the action block syntax.

Cyclomatic Complexity is a measure of how complex a program is, and was originally developed by Thomas McCabe in 1976. The Cyclomatic Complexity is a count of the number of linearly independent paths within a program. For instance a simple linear program that has no decision points has a complexity of 1, whereas if it contained an IF statement, then there are two separate paths through the code and so it would have a complexity of 2. A way of calculating the Cyclomatic Complexity of a program is to look at its control flow graph.

The formula for the complexity is: **M= (E-N) +2P**

where: M= complexity, E= the number of edges in the graph, N= the number of nodes, P= the number of connected components (always 1 for a single program).



A easy formula of calculating Cyclomatic Complexity for CA Gen action blocks is
Number of Decisions +1.

A decision in this context is a possible branch in the code, and the method adopted in this study is to increment the count by one for each of the following types of statements:

- IF
- ELSE IF
- REPEAT – UNTIL
- WHILE
- FOR
- CASE
- Database exception clause (except for *when successful*)

The original McCabe method treated each branch as adding one to the complexity count. A variation of this method (extended cyclomatic complexity) includes Boolean operators in the count. In this study we refer to the original method as CC1 and the extended method as CC2.

For example, the statement:

IF a=1 AND b=1 adds a complexity of +1 in CC1, but +2 in CC2.

The study compared the results of an additional two types of method, CC3 which only increments the count by one for each CASE OF clause and ignores each individual CASE clause and CC4 which counts distinct IF/ELSEIF statements, i.e. if the same IF statement is present multiple times, it is only counted once. An advantage of CC4 is that logic that repeats the same test multiple times (for

example debug statements nested within ***IF debug = 'Y'*** do not have an artificially high complexity count.

The table below summarises each of the four methods used in the study.

Statement Type	CC1	CC2	CC3	CC4
IF / ELSE IF	+1	+1 for IF/ELSEIF and +1 for each AND/OR clause	+1	+1 for each distinct IF/ELSEIF clause
REPEAT – UNTIL	+1	+1	+1	+1
WHILE	+1	+1	+1	+1
FOR	+1	+1	+1	+1
CASE OF			+1	
CASE	+1	+1		+1
Database exception clause (except for *when successful*)	+1	+1	+1	+1

Study Data

The data used for the study is a sample of around 1500 programs (action blocks) from the GuardIEn product, which is a change and configuration management tool developed by IET using CA Gen.

GuardIEn provides suitable data for the study as it has been developed over a 20 year period; it contains a large variety of programs ranging in complexity from simple action blocks with a few statements to large and complex ones with thousands of statements. In addition, every change to the programs over the period of time has been logged in a change request database, and this provides historical data on how many times the program has been modified.

For each action block the following metrics were calculated:

- STMT: number of statements (only executable statements have been counted, excluding disabled statements and comments)
- CC1: Complexity Count using CC1 (i.e. Boolean operators not included)
- CC2: Complexity Count using CC2 (i.e. Boolean operators included)
- CC4: Complexity Count using CC4 (i.e. counting distinct IF statements)
- CR: Number of individual Change Requests (changes).

In the data the correlation has been measured as a Pearson correlation coefficient; this gives a value between -1 and 1: where 1 is directly proportional and -1 is inversely proportional and 0 is completely uncorrelated. A value above 0.5 indicates a high positive correlation and a value below -0.5 a high negative correlation.

Complexity vs. Statements

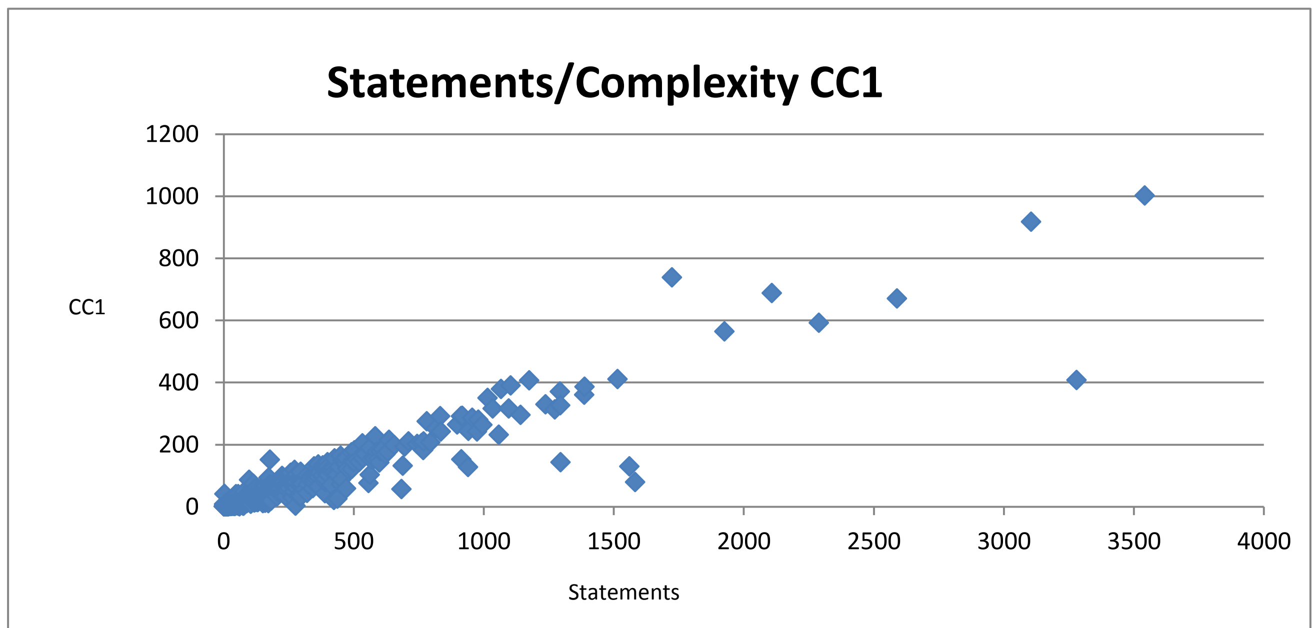


Figure 1 – Statements vs. Complexity (CC1)

Fig. 1 plots STMT against CC1 for each action block. As shown in the table below, there is a high degree of correlation between the complexity count and the number of statements.

The correlation was also calculated for CC2 vs. STMT and CC4 vs. STMT. These results indicate that there is very little change in correlation when using CC1, CC2 or CC4.

Method	Correlation
Statements/CC1	0.945
Statements/CC2	0.943
Statements/CC4	0.936

Correlation with Changes

The sample data included the number of changes made to a program. These changes include fixes, enhancements and other changes implemented because of external factors, for example, changes in the 3rd party products used by or supported by the application.

A hypothesis of this study is that the number of changes introduced to a program over time is strongly correlated with the inherent testing complexity of the program. If it has been developed and never changed, then it is likely to have been less complex to test because it was error free and had a stable specification. In contrast, a program that has been changed a lot is likely to have had a combination of a complex and varying specification and/or many errors in its implementation.

The study therefore decided to compare the correlation between the CRs for a program and its CC and STMT metrics as shown in Figures 2 and 3.

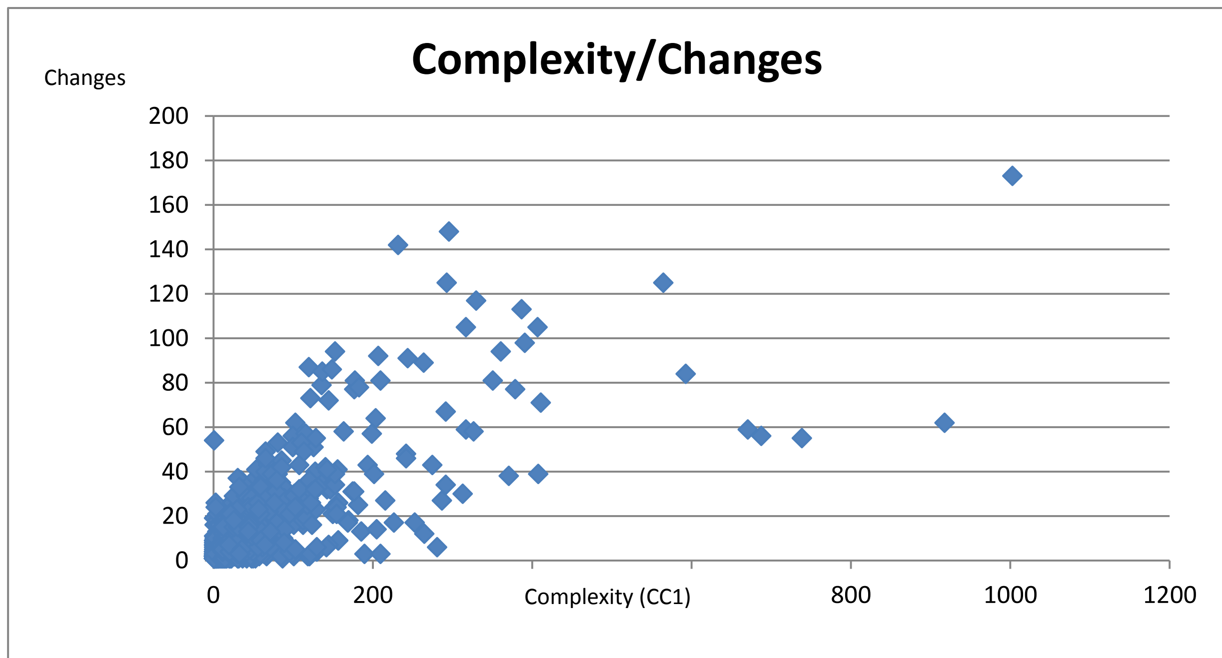


Figure 2: Complexity vs. Changes

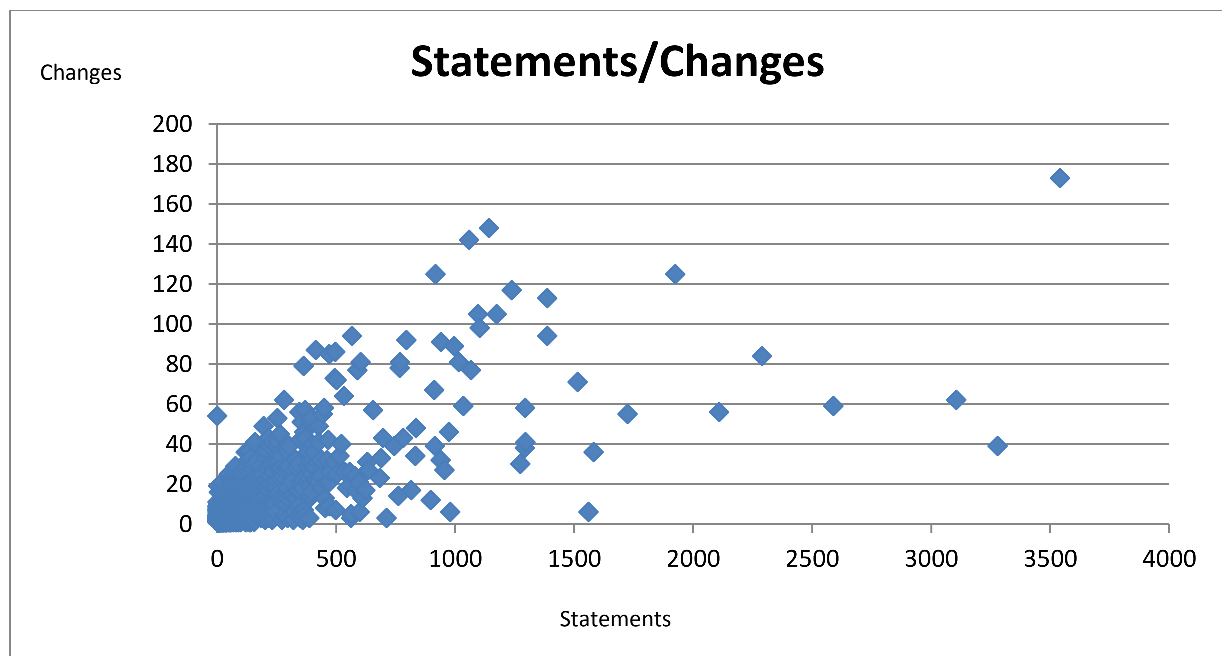


Figure 3: Statements vs. Changes

The table below shows the correlation for these two sets of data. It shows that complexity count correlates marginally more strongly with the number of changes made to a program than the number of statements.

	Complexity/Changes	Statements/Changes
Correlation	0.743	0.718

This implies that that complexity is a slightly better predictor of the amount of testing required for a program.

Comparing Client & Server Logic

GuardIEn is a client/server application with the client (windows presentation) logic concerned with handling the user interface and the server logic executing on a remote server. The majority of the complex business logic is contained within the server logic.

The data was divided into client-side and server-side and the correlations re-calculated. For this analysis, CC4 was used because it has the advantage of not duplicating repeated IF statements. The table below shows the correlations between Statements and CC4 complexity, Statements and Change Requests and CC4 complexity and Change Requests.

It shows that for server logic, there is a much higher correlation between Cyclomatic Complexity and number of changes compared with using simple statement counts. For client logic, the difference was less pronounced and there was a also a lower correlation between number of changes and either statements or complexity.

Correlation	Server Logic	Client Logic
Statements/CC4	0.86	0.98
Statements/CR	0.61	0.69
CC4/CR	0.73	0.67

This implies that complexity count it is a much better predictor of how much testing is required for server logic than the number of statements.

However with client logic the correlation between the number of statements and the complexity is so high that we don’t see a very significant change in the correlations between them and the number of changes.

The following two graphs show the data for Change Requests against Statements and Complexity for server logic.

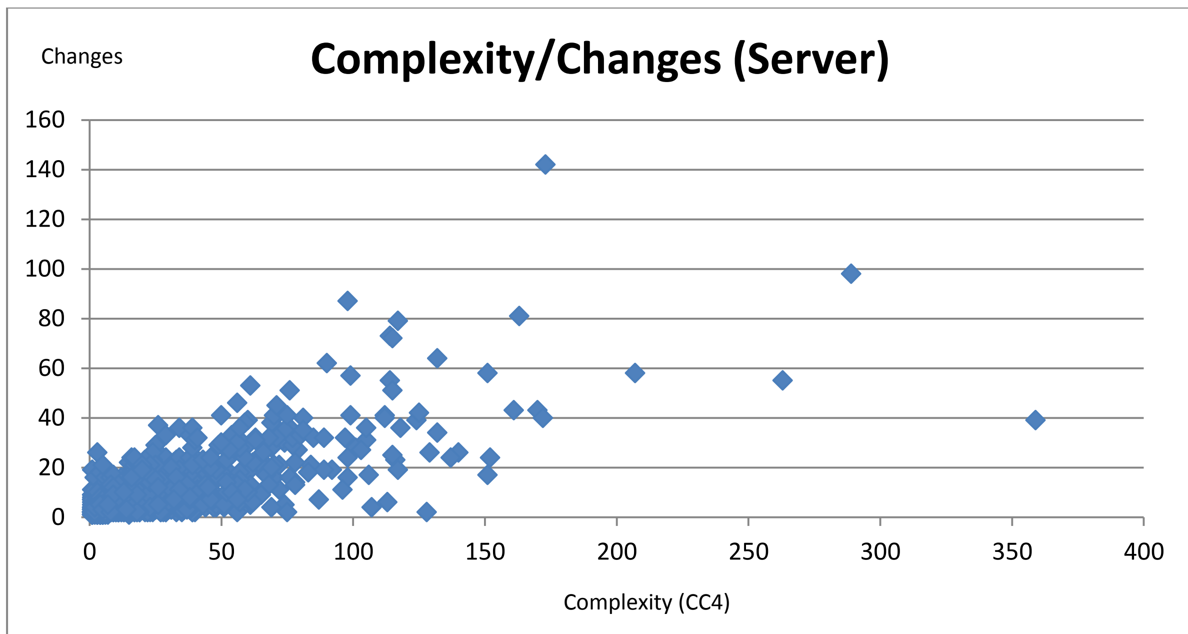


Figure 4: Complexity vs. Changes for Server Logic

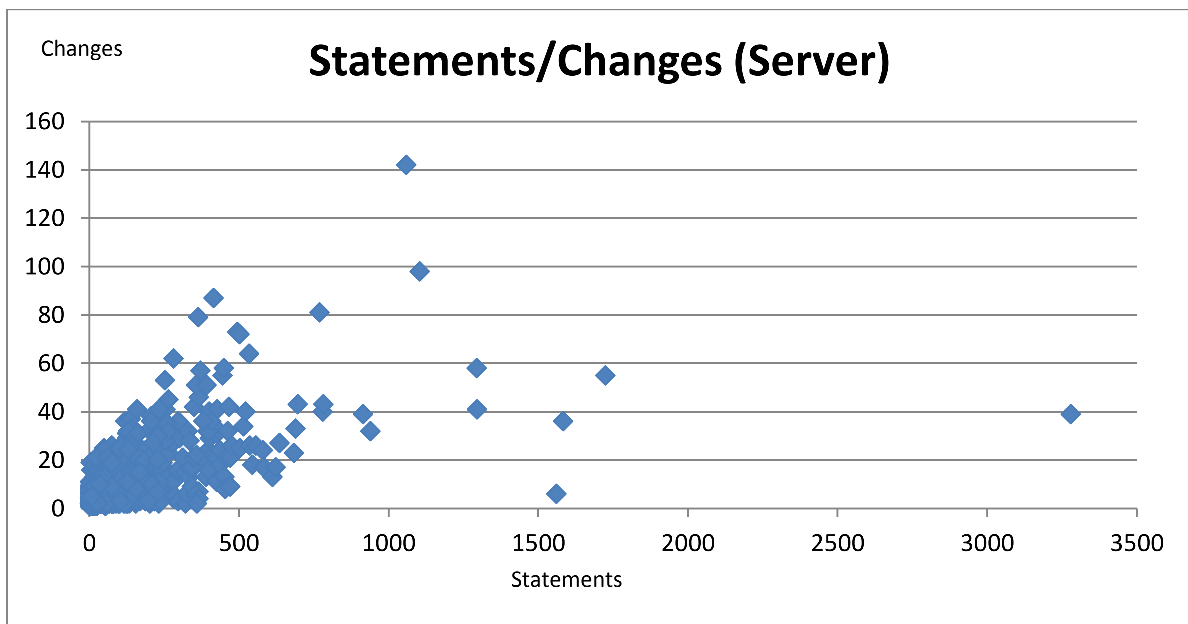


Figure 5: Statements vs. Changes for Server Logic

Complexity Density

As an alternative metric, the study also calculated the Complexity Density, which is the CC divided by effective number of statements, i.e. CC/STMT.

The correlation between Complexity Density and Changes was -0.04, which indicates that there is no correlation between them and hence the conclusion that Complexity Density is not a useful metric in the context of this study.

Conclusion

The results from the study show that the complexity count correlates more strongly with the number of changes made to a program than the number of statements for server logic. This implies that the complexity count is a more useful predictor of the number of changes compared with using just the number of statements.

However, when dealing with client logic, there is a very high correlation between statement number and complexity, and so either can be used as a reasonable but less accurate predictor of the number of changes.

Using the hypothesis that the number of changes over the life-cycle of a program is a useful measure of how much effort should have been expended on testing a program; the complexity count is a useful metric to use when prioritising which programs to fully test.

However, it must be stressed that these conclusions cannot be considered absolute as they are only showing correlations and therefore the probable outcomes of creating a large complex program. It is also the case that large programs are not always complex and that small programs are not always simple. The amount of changes made to the program may not be the result of errors made in development and therefore cannot be found during testing. It is also possible for a very competent programmer to create a large and complex program perfectly with no need for changes; and a sloppy one to create a small simple program with many errors. However, when testing a number of programs, with limited testing time available, it is a useful way of allocating testing efforts so as to maximise the probability of finding errors.

References:
http://en.wikipedia.org/wiki/Cyclomatic_complexity,
http://www.literateprogramming.com/mccabe.pdf
http://en.wikipedia.org/wiki/Pearson_correlation_coefficient
http://www.enerjy.com/blog/?p=198