CHAPTER - 9

CONCLUSIONS
AND
SUGGESTIONS
CHAPTER 9

CONCLUSIONS AND SUGGESTIONS

9.1 CONCLUSIONS

From the work reported in the present study, a number of conclusions and inferences regarding software systems may be drawn. Firstly, this study provides a conceptual and practical framework for the measurement of various software characteristics like complexity, testability, expected number of faults etc. and their importance in the development and maintenance of software systems.

Finding answers for the following questions were the motivation factors of this work:

1. How to design program length estimators with reference to modern programming languages?
2. How to design unified software complexity metrics for accounting various aspects of complexity like size, control structures, nesting etc.?
3. How to predict software systems size and complexity before coding?
4. How to resolve a contradiction about level of Ada language raised in an earlier study?
5. How to establish relationship among software complexity metrics and also between complexity and various software characteristics like testability, expected number of faults?
6. How to design techniques for faults prediction in a software module?

The purpose of the work done in this study was to answer the above mentioned questions. A computer program is made of statements and in turn, a statement consists of tokens - operators and operands. Earlier program length estimators were based on count of operators and operands only. Modern programming languages such as C, Pascal, Ada, Modula -2 are rich in functions / procedures, keywords/ reserve - words and thus it seems logical to take a separate category for these tokens in addition to operators and operands categories. In the light of this justification, a three term program length estimator \( N_{gr} \) was proposed. We have analysed about 200 computer programs written in Pascal, C, Ada and Modula -2 modern programming languages for this purpose. For comparative study of various length estimators including the proposed three term length estimator, the correlation coefficients between the predicted and observed length were calculated and analysed. We cannot claim that any length estimator is superior over the others. However, in some cases, the proposed length estimator \( N_{gr} \) comes out to be slightly better than the other estimators, namely \( N_h, N_b, N_f \) and \( N_z \).

Different complexity metrics take into account different aspects of program complexity and it is difficult to design a single metric which may address all possible aspects of complexity of a computer program. For example, Halstead's complexity metric, namely, effort takes into account size of the software module only and it does not include other factors of complexity such as...
program flow control, nesting and so on. Similarly, McCabe's cyclomatic complexity number measures the complexity due to program flow control only, but does not keep track of other features of software systems such as operators, operands, etc. Here, we have tried to design a new unified weighted complexity measure which takes into account the complexities due to size of software modules, position of a statement in the logic of a program, types of control structures present in the module, and their nesting. The concept of weight is used to quantify these various aspects of complexity of a software module. Through this metric, we have, also, tried to resolve some existing controversial issues relating to McCabe's complexity measure.

In the present study, we have, also, attempted to predict the size of a software module before its coding. For this purpose, we have tried to generalize the value of a constant parameter for the languages studied in this work. By using the generalized value of this parameter for a particular language, we may predict the size of a computer program to be written in that language by simply using it with the value of unique operands, \( n_2 \). The results obtained by using these values of constants for various languages have been reported and have been found to be encouraging up to some extent. If we could predict the size of a software module before its coding, then we may also predict its complexity (before its coding) by the application of one of the composite complexity models given by Hops et al as explained in chapter 3.

Halstead's theory of software science defines a language level metric which can be used for comparative study of programming languages. The level of the programming languages studied here has been calculated. Ada is a high level, strongly typed language that incorporates modern software engineering concepts. But in an earlier study by our research group, a contradiction has been raised about language level of Ada. Ada is usually considered to be a higher/same level language as Pascal but in that earlier study its level has been calculated as 0.27 placing it much below the assembly language (0.81). In this study, this contradiction has been resolved and level of Ada has been calculated as 1.47 which places it higher than Pascal, FORTRAN, BASIC and C high level languages. One of the main reasons for resolving this contradiction is variations in counting rules. So, counting rules have to be designed rather carefully.

Study of software complexity metrics offers a variety of advantages in the design, construction and maintenance of software systems. Many of these metrics have been validated on actual software systems, however, there are only a few studies which establish the relationships among the metrics. In this study, we also report the relationships among three complexity metrics, namely, Halstead's effort, McCabe's cyclomatic complexity number and Stetter's program flow complexity. These three complexity metrics have been applied to the same set of 24 programs written in Modula-2 language. The results of this study establishes high degree of relationship among the three above mentioned complexity metrics. This fact has been supported by the high value of correlation coefficient \( r \) among the three metrics. It seems reasonable also because all these three complexity metrics do not observe any connections of a given module to its environment and concentrate only on the code complexity. It means that they do measure the factors contributing to complexity at the micro-level, or intra-module level only, and not at inter module level. Here, we have also tried to give an empirical support to Henry et al finding that if high correlation coefficient \( r \) exists between cyclomatic complexity measure \( V(G) \) and volume \( V \) for a software system, then that system is more decision bound than computation bound. This fact may be useful for a more careful categorization of systems which may further help for finding software costs and quality more exactly. It is an interesting finding but whether this fact is valid remains to be verified by future studies.
Mere presence of faults does not mean software failure. In software failure three necessary and sufficient conditions, namely, execution of fault, error infection and propagation of this infection to the output are to be met. It is sensitivity analysis which deals with these three stated conditions. In this study, we have proposed technique for calculating testability (amount of testing required to reveal expected number of faults) and also tried to link the testability with software complexity. This study establishes quantitatively that more the complexity, more the amount of testing required to reveal faults. It agrees with intuition also. By knowing the testability of a program, we may utilize testing resources more effectively because a program with low testability requires more testing than a program with high testability. Thus, we may predict about its quality as well.

One of the parameters which decide a quality of the software system is the number of faults present in it. So if somehow, we could predict the faults present in a software module, then we may comment about its quality. In this work, we have tried to propose a technique, namely, bebugging method, for faults prediction in a computer program. In this technique, some fixed number of artificial bugs are inserted in a program. It is, then, debugged by a tester or a programmer. Thus by using the values of artificial bugs and the faults reported by the tester, we can predict the expected number of faults present in the program as explained in chapter 8. For giving empirical support to this technique, we have, also, conducted a class-room experiment. In this experiment, bebugging method was applied to 7 Pascal programs by 15 M.C.A. students. The experimentally predicted faults obtained through this experiment were compared with theoretical predicted faults (Lipow's method) and a positive correlation was found between the two. But, for the verification for real life data, this technique seeks further experimentation.

9.2 SUGGESTIONS FOR FUTURE WORK

Programs used in this work were small and collected from the various standard programming textbooks. In experiment on bebugging method, time constraint of the subjects did not permit us to select large programs. But, in our opinion, these factors do not limit the applicability of these results to large and of real life applications. Most of the programming paradigms recommend 'divide and rule' principle for solving large, complex problems. In order to control the complexity, a large program is divided into smaller parts like procedures, functions, modules and so on. The typical size of a module may range 20-50 lines of code. The overall complexity of the software system will depend upon the complexity of its modules and their interactions. Here, the results of the thesis have been applied at module (intra-module) level. For interaction among modules (inter-module level) and their contribution to overall complexity of the software system, the results need further empirical support through experimentation.

The overall complexity of a software system does not depend upon a single attribute. There are various aspects like size, control flow, nesting etc. which add to complexity. It would be an interesting attempt to design a single unified metric which may quantify maximum possible aspects of complexity of a software system, like, our weighted complexity measure reported in chapter 4. It includes four aspects of complexity, namely, Halstead's size, McCabe's control structures, nesting level of control structures, and position of a statement in the logic of a program. It would be quite interesting to attempt in this direction so that software engineers could rely on a single complexity metric.
The counting rules / strategies for various programming languages used in this study have been
designed very carefully by considering different features of programming languages. But, these are
empirical in nature. Also, a change in counting rules may affect the measurements of various
parameters involved in various software metrics. So, an attempt should be made for
standardization of counting rules for different programming languages which have the similar
features, like, object - oriented languages, functional programming languages and so on. By doing
so, results of the different experiments on the similar languages may be compared uniformly.

The debugging method for faults prediction has been applied to small programs collected from the
open literature. For its further verification, it would be interesting to apply it to real life data
(errors) collected from the industry. For this purpose, it seeks more experimentation.

The sample programs used in this study involve only simple programming constructs like
sequence, looping, iteration and so, more experimentation is needed with programs having the
advanced features of programming, such as overloading, exception handling, tasking, fault
tolerance and so on. It would be quite encouraging and interesting to study the effect of such
features as well.

Also, no attempt has been made to investigate object - oriented, functional, logical, networking, and
relational database oriented languages. Currently, object - oriented languages like C++, Eiffel,
SmallTalk and networking structured programming language, JAVA, and relational database
oriented languages like SQL (Structured Query Language) are very popular and being used in the
development of software systems extensively. Again, it would be very useful, interesting and
challenging task to design software metrics for software applications developed in these modern
languages.