CHAPTER I

INTRODUCTION

In the early years of computer development, as computer generations succeeded one another, the hardware performance-price ratio kept climbing at an average rate of about 25 percent a year; it was still advancing rapidly in the early 1980s. Programming productivity, on the other hand, while hard to measure, gained at a much slower rate - something like 7 - 9 percent per year. The continuing hardware improvement created a series of computer systems on which it was economically feasible to run even larger programs, but the slower rate of improvement in programming led to progressively larger programming organisations requiring more levels of management. Some software systems approached a million instructions and absorbed 5000 man-years of development time. By the mid 1960s the fact that program development almost defied the effective management control was becoming evident.

Programs of this era were not only extremely large but frequently lacked clarity as well. They were typically all in one piece, that is, not modularised. The sequence of program execution from one instruction to the next in line would jump instead to an instruction that was pages away in the program listing. A reader could track one or two of these jumps or branches, but scores of them made the program logic hard to grasp.

Since the product being developed was something of a mystery, it was nearly impossible for managers to find out enough about it to supervise the work of the programmers, to co-ordinate them with other groups, or to estimate how much time such poorly defined tasks would take or how many programmers were needed. It was

under these circumstances that the **NATO Software Engineering Conference (1968)** pointed out directions in which solutions to the **software crisis** should be sought.

Since then several attempts have been made to transform software development from an artistic endeavour to a sound **engineering discipline.** Within this discipline efforts have been made to identify the desirable properties of **quality software** and to provide programming methodologies and language features which can be readily applied to the production of quality software. However, in order for this undertaking to result in a true engineering discipline, it is necessary that tools, techniques and measures be developed for software analysis. Unfortunately, scientific knowledge available on which to base such an important discipline is still very small.

This dissertation therefore attempts to study the following **aspects of software engineering** in order to develop new techniques and measures for **quality software development.** They are :

1. Regular expression representation of programs,
2. Complexity measures based on program text and regular expression,
3. Flowgraph decomposition, and
4. Path generation for program testing.

1.1 STRUCTURAL REPRESENTATION OF PROGRAMS

Regular expressions provide a compact representation of programs. Several authors like Magel [1981], Oulsnam [1982] and others have proposed the regular expression representation of programs. Their regular expression representation is, however, not capable of clearly

identifying the components of a program and their nestedness. It is also not possible to find out whether a program corresponding to a regular expression is structured or not.

In the present work a modified form of a regular expression representation is introduced which overcomes the above problems. The **modified regular expression (MRE)** representation is first defined for structured programs and then extended to unstructured programs. Woodward et al. [1979] have introduced the concept of knots to measure program complexity. A similar concept of knot at program and flowgraph level is introduced to characterise unstructuredness of a program.

For obtaining a structured representation of an unstructured program or flowgraph, the concept of **local-structuring** is introduced. To perform local structuring **Partial-regions (P-regions)** of a flowgraph are defined with reference to a data structure called a **Binary Interval Search Tree (BIST).** A traversal of this tree gives an ordered set of P-regions. The knots in the P-regions can then be eliminated with respect to a basis set to obtain a **structured MRE (SMRE)** representaion. The concept of a graph grammar is used to convert a program into the equivalent MRE form. Using these concepts the following four algorithms are developed:

i) To convert a flowgraph into an MRE,
ii) To convert a program into an MRE,
iii) To convert an MRE into a flowgraph, and
iv) To convert an MRE into a program.