An Investigation into Two Aspects of Software Engineering : Specification and Complexity Measurement

Dissertation submitted to the Jawaharlal Nehru University in Partial Fulfilment for the award of Degree of

MASTER OF TECHNOLOGY

MURALIKRISHNA, G. V.

School of Computer and System Sciences
Jawaharlal Nehru University
New Delhi-110067

1987

CERTIFICATE

This work entitiled "An Investigation Into Two Aspects of Software Engineering: Specification and Complexity Measurement" embodied in this dissertation has been carried out in the School of Computer and System Sciences, Jawaharalal Nehru University, New Delhi-110067 and is original and has not been submitted so far in part or full for any other degree or diploma of any University.

Muralikrishna, G.V.
Student.

Prof. K.K. Nambiar
Dean

Prof. K.K. Nambiar
Supervisor.

CONTENTS

PREFACE

Software Engineering is gaining importance in containing the gallopaing costs of software systems. In this dissertation two different aspects of software engineering are examinied. The first one is specifications and second is complexity measurement of a software system. Regarding the specifications, an efficient algorithm is developed for verfying these specifications of the system for their correctness and completeness. The algorithm takes the specifications in the form of decision tables and it needs the set of logical dependencies that are present in between the conditions of the decision table.

About the complexity measurements, a review of many software metrics that are available to this date is attempted. They were first described and then compared with each other. These metrics, if standardised, can play a great role in increasing the reliability and maintainability of the software system. Here I wish to express my most sincere thanks and deep sense of gratitude to Dr. K.K. Nambiar, Prof. and Dean, School of Computer and System Sciences, Jawaharalal Nehru University, New Delhi, for his keen interest, inspiration and constructive

criticism during the course of investigation. His endless succession of argument and discussion always provided me a stimulating atmosphere and keen interest for research.

I am also thankful to the Libreary Staff of Jawaharlal Nehru University and Indian Institute of Technology, New Delhi for their cooperation

Thanks are also due to my friends who all helped me to complete this work in time.

Finally, I am grateful to University grants commission for providing me financial help in the form of Junior Research Fellowship.

G.V. Muralikrishna
Student.

CHAPTER 1

I N T R O D U C T I O N

1.0 BACKGROUND : SOFTWARE ENGINEERING

The declining cost of computer hardware has resulted in an increase in both the number and complexity of new applications. To control rising costs, many of the ideas and practices of the established engineering disciplines have been applied to software development. The term "software engineering" was chosen in 1968 to describe techniques, tools, and disciplines that support every stage of software lifecycle.

The use of software engineering practices has been shown to significantly reduce program development costs on large projects. A recent study by a manufacturing company in USA showed an average cost reduction of 73% over forecosts for three projects. However, it is estimated that the techniques are not being widely used, underlining a need for the education of both programmers and managers in this area.

The software development process may be characterized by a number of key steps collectively called the software life cycle :

(1) Specification: The software requirements, i.e. the system functions and operational constraints, must be established and specified.

(2) Design: A software design must be derived from an analysis of the software requirements.

(3) Implementation: The software design must be realised in a programming language which can be executed on the target computer.

(4) Testing: The implementation must be tested to ensure that the completed system meets the software requirements.

(5) Operation and Maintenance: The system must be installed and used. If system errors are discovered these must be corrected and changes to the original requirement may involve adding additional constraints to the system.

The complete process and sequences of operations involved in the software development can be shown clearly by a familiar "waterfall" model which is shown in Figure 1.1.

It has been recognized that specification is an extremely important tool in a large scale software design. It should be noted that an important aspect of specification is that it must be precise. Since specification contains the information that the designer explicitly assumes about the system, lack of precision can hurt the design in many ways, e.g., the problem being solved may turn out be not the intended one, or later refinements of a program may not be consistent with early design decisions.

After the software specification phase is completed, specification is then transferred into design and later implemented in the form of computer programs. It has recently been shown that the complexity of a program is one of the major causes of unreliable software.

In general, complexity of an object is the measure of the mental effort required to understand that object. Easy human understanding of a program is an essential requirement for reducing the cost of maintenance of the

software system[1]. If a program is voluminous, its complexity is automatically more. In recent investigations, it is proposed that the complexity of a program design should be considered as a function of the relationships among modules. The complexity of a module is a function of the connections among the program instructions within the module.

1.1 SPECIFICATIONS

Studies show that any error made in the requirements stage is three orders of magnetitude more expensive than the one made in the coding stage [RAMA86]. The importance, therefore, of early detection of errors can not be over emphasized. Requirement specification is a technique to describe the functionality of a system, in enough detail, so that erroneous assumptions are not made during design and implementation. The main problem in writing specifications is that large systems are so complex that even the description of their functionality is difficult and error-prone. One of the reasons for this is the ambiguity in the medium of expression. The customer

---

1. Here we are not concerned about the computational complexity of an alogarithm, which is a measure of the computer time and memory needed for solving the problem.

generally uses natural language, which is inherently ambiguous, to express the specifications. Only limited success has been achieved in processing natural languages automatically and the situation is unlikely to change in the foreseable future.

To solve the problem of ambiguity in natural language, one of the way is to use decision tables and another and most recent one is to use a formal requirements language. Here, in this dissertation, decision tables are used.

After collecting the requirements, how the systems analyst makes sure that the list is correct and complete? As explained earlier, it is important to verify the specifications for their validity and sufficiency. The first objective of this dissertation is to develop an algorithm to verify the software system's specifications (in the form of decision tables) for their completeness and correctness. The discussion is limited to the systems that can be modelled by a decision table. Current methods for checking are inadequate, when logical dependence relationships exist among conditions. The earlier work in this direction will be discussed here.

[KING68] formally defined logical errors in decision tables as that occur whenever the decision table has ambiguity or incompleteness and gave a method for checking decision table. [KING69] later improved the method and defined logical dependence between conditions in a decision table and demonstrated how this could be used to pinpoint logically impossible rules. The logically impossible rules may be excluded from the set of apparently ambiguous rules, and one may thus identify real ambiguities. [KING69] advocated the use of first order predicate calculus to detect the logically impossible rules.

[RAJA78] observed that in order to detect logical errors in a program incorporating decision tables, the interrelationship between the different parts of the program should be considered. It is not rare that a mistake in an arithmetic statement executed before executing a decision table causes a condition in that decision table to be erroneously satisfied. Further, due to restrictions on variable values or computations performed earlier, a set of rules in a decision table may become logically impossible. Thus restricting the scope of analysis to a decision table in isolation is not sufficient.

[RAJA78] established that for a non-trivial class of decision tables and programs with embedded decision tables, error checking can be performed by a computer program. The paper further established that it is not necessary for the analyst to supply statements of relations between conditions. This is important, since obtaining such statements of relations is, in itself, a non-trivial problem.

In essence, the paper developed an algorithm which will delve into the logical dependence relationships among conditions. Also, an algorithm to detect logical error was developed, based on determining whether a set of linear inequalities has or has not a solution.

The previous investigations have concentrated on detecting logical errors in a decision table. Little has been said about the completeness of a decision table. If a system is modelled and specified by a decision table, then the issues of completeness and correctness are of paramount importance, because in this context rarely is anything worse than an incomplete and/or incorrect specification. One specific objective of this dissertation is to develop a procedure which can be used for the type of analysis that checks the completeness and correctness of a decision table.

1.2 SOFTWARE COMPLEXITY

As defined in section 1.0 the software complexity is the mental effort required to understand the software system seeing the source code of the system. It is now an established fact that the software complexity determines the reliability and cost of maintenance of the system.

The published literature discusses only specific implementations of algorithms. What is missing is an explicit recognition that beginning with the problem statement or specification there exist, in general, multiple solutions, and the programming process can be envisaged as a combination of both analysis and synthesis processes aimed at identifying the most desirable solution among a large number of feasible alternatives. The search for a particular solution forms the core of the relationship between reliability and complexity and is the dominant factor that influences the reliability of programs.

Currently, more time is spent maintaining existing software than in developing new code. In fact, resources invested in maintenance have been estimated to be three times higher than those required during development. Metrics computed from the initial code which could estimate

either the reliability of modification or the time required to implement these would prove invaluable to the software manager, who must allocate the time and resources necessary for software maintenance.

In the past one decade, many complexity metrics were developed using different aspects of computer programs. The most straight forward and widely applicable approach is based on program size. The counts like number of lines, procedures (modules), etc. which are representative of the volume of the source code can be used as software complexity metrics. Because this metric has many limitations, many new metrics were proposed.

The forerunner is the Halstead's software science [HALS77], which treats the software field as a science, like any other physical sciences, rather than as an art. Many more matrics based on graph theory are prepared and some people advocated hybrid metrics.

All the metrics developed till now are having many limitations and they are yet to gain acceptance in practice. Eventhough this field produced lot of literature, the

importance of the definitions proposed is still to be established. In a later chapter an attempt is made to study and compare different complexity metrics.

1.3 OUTLINE OF DISSERTATION

The remainder of the dissertation is divided into three chapters. Chapter 2 attempts to explain decision tables which are useful in Chapter 3. In Chapter 3, an algorithm is developed for checking a decision table for completeness and correctness. In Chapter 4, a study of different complexity metric is made.

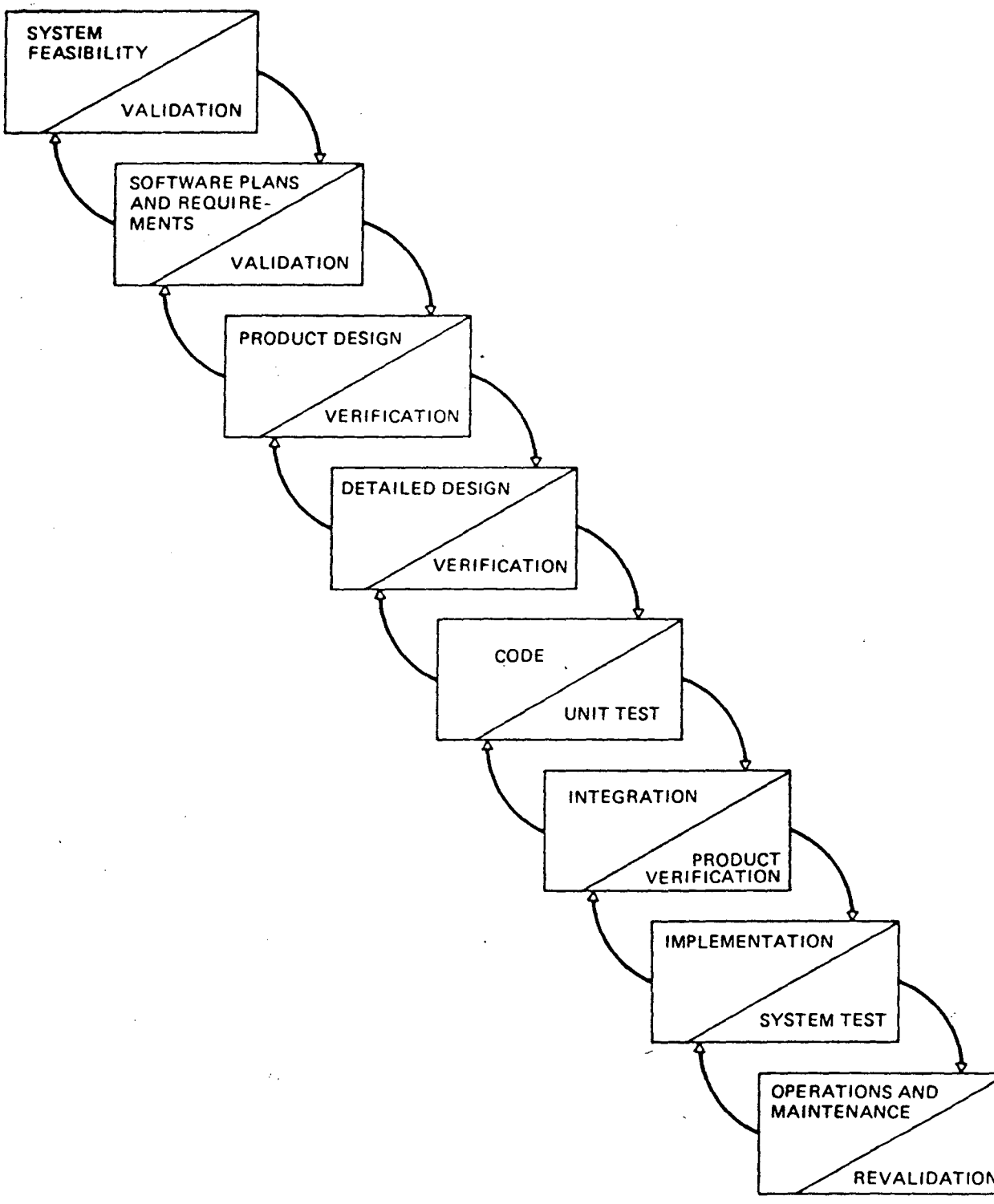

Figure 1.1 The "waterfall" model of the software life cycle.

CHAPTER - 2

DECISION TABLES

2.1 INTRODUCTION

The decision table can be best used by programmers, analysts, and other users of computer facilities because they provide a simple tabular representation of complex decision logic. Decision tables, although developed primarily as man-to-man communications, can ease the problems of programming and documentation in many applications where the feasibility of using the traditional flowcharts, narrative discriptions, or other communicative media is questionable.

Eventhough many higher level programming languages are available, there is still a wide gap between computer specialists and users. So, especially in management-to-man communication there is a possibility of misunderstanding in systems analysis and design and in implementing the chosen procedure into a workable computer program. Decision tables, being easily understandable can fill this communication gap. In addition, because decision tables succintly display any conditions that must be satisfied

before any prescribed action is performed, they are becoming popular in computer programming and system design as devices for organising logic, especially when attempting to handle very complex situations, and to account for every possible combination of conditions. Furthermore, the extent and nature of the changes required to update or revise an application programme is easily provided by the unique form of the problem statement in decision tables.

Flowchart is a graphic language form that has also been widely used for man-to-man communications. Flowchart was specifically developed for the purpose of representing operations related to computer activities, such as system analysis, system design, programming, documentation etc., can also frequently be utilised for noncomputer related activities. A comparison between decision tables and flowcharts is made in the following section.

2.2 DECISION TABLE STRUCTURE

A decision table provides a tabular representation of information and data. Information displayed in this manner is easily comprehended visually, even if the table of information represents a complex logical problem. A

decision table is a structure for describing a set of decision rules, [POOC74]. The basic structure of a decision table is universely accepted as that illustrated in fig 2.1.

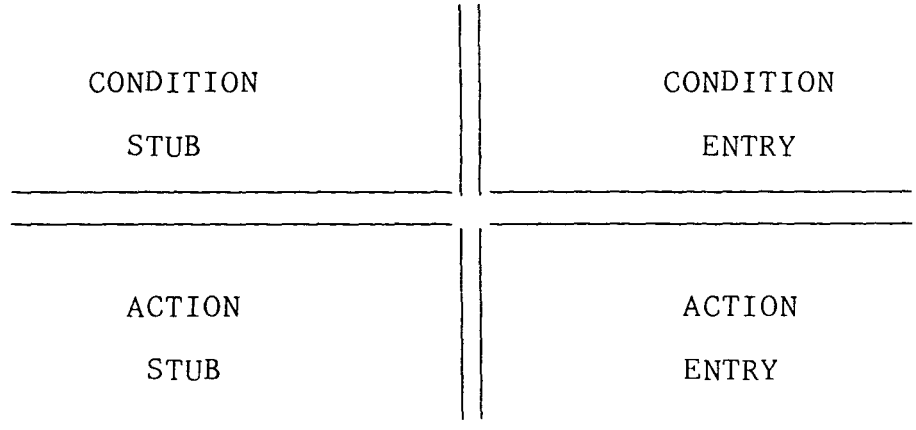

Fig. 2.1: Decision table structure.

The decision table can be divided into four quadrants. The upper left quadrant, called the condition stub, should contain all those conditions being examined for a particular problem segment. The condition entry is the upper right quadrant. These two sections described the set, or string of conditions that is to be tested. The lower left quadrant, called the action stub, contains a simple narrative format for all possible action, result from the conditions listed above the horizontal line. Action entries are given in the lower right quadrant. Appropriate

actions resulting from the various combinations of responses to conditions will be indicated in the action entry. An example of decision rules and the IF-THEN function are illustrated in table 2.1.

		Decision Rule 1	Decision Rule 2	Decision Rule 3	Decision Rule 4
IF					
AND					
AND			CONDITION ENTRY		
AND	CONDITION STUB				
THEN			ACTION ENTRY		
AND	ACTION STUB				
AND					

Table 2.1: Decision Table

The meaning of different sections of table 2.1 can be described as follows. Each decision rule is a combination of responses to conditions in the condition entry quadrant. The decision rules are numbered for identification purposes in the rule header portion of the table. The top most horizontal line represents IF, while the remaining horizontal lines represents AND, and the double horizontal line THEN. Note that the condition half of the table is separated from action half by a double horizontal line and stub sections are separated from the entry section by a double vertical line. These lines

improve the readability of a table, and can be preprinted on forms. In addition to these, each table is given a table header which serves the purpose of table identification.

If a condition in the condition stub is true a Y is entered for that particular rule in the condition entry; if the condition is false an N would be entered. In a situation where a practical condition is irrelevant, a 'don't care' would be indicated by the use of a dash (-) or an I. Two other entries, the * and $ are used to indicate mutual exclusion of one condition with another on a rule by rule basis [KING69]. Whenever the case arises within a single rule that the satisfaction of some "required" test (Y or N entry) makes some other required entry a foregone conclusion then the special entries * (in place of N) or $ (in place of Y) can be used to indicate this fact. The use of these implicit entries can be illustrated by the example of Table 2.2.

2.3 TYPES OF DECISION TABLES

There are three types of decision tables in current use [POOC74]. The limited entry decision table, the most popular and most often used, allows only the entries explained in section 2.2. In the second type of decision

table called extended entry table some conditions are allowed to appear on the condition entry space also. The third variety, mixed entry table, is a combination of limited entry rows and extended entry rows. The extended and mixed entry tables can always be transformed into limited entry tables. This is the reason for not considering the last two types in the treatment here.

2.4 COMPARISON OF DECISION TABLES AND FLOW CHARTS

The decision table is a convinient form for expressing any conditional alternatives, where a particular path to be followed is dictated by a combination of a number of conditions. Flowcharts in such cases can become very complex and difficult to follow, and involve testing for each condition more than once. The advantages of decision table can be listed as follows :

* Clear enumeration of all operations performed.
* Clear identification of the sequence of operations.
* Effective means of communication between people in and out of data processing field, i.e. not limited to computer applications.
* Easy to construct, modify and read.

* It is possible to verify a decision table for its correctness and completness using a computer programme.

* Can be used to documentation applications involving complex interactions of variables. Unlike flowcharts it is not affected by personal preference or jargon.

* When applied to computer systems decision tables foster better use of subroutines and provide a complete data check for debugging.

* Directly adopted and possibly converted directly to computer operations through symbolic logic computer programmes.

* Easier visualization of relationships and alternatives.

Compared to flowcharts the decision tables have many disadvantages also.

* The decision tables are slightly more difficult to learn.
* For complex situations, they may become extremely large.
* Multiple tables may be needed in certain cases to document decision logic.
* Many people find the graphic display of flowcharts more

meaningful than a tabular description of logic.

* Desire for automatic translation ability causes too detailed requirements for man-to-man communication purposes.

Although decision tables are not the answer to all documentation and programming problems, they do offer certain advantages that overcome some of the drawbacks of flowchart technique. With the state-of-the-art advancing sufficiently for checking a decision table and enable economic conversion of decision tables, their use may show a marked increase.

2.5 USES AND APPLICATIONS OF DECISION TABLES

Decision tables are useful in many areas of applications.

In simulation models: The ability of decision tables in handling complex logic makes them a definite aid in formulating logical flow of simulation models. Here the decision tables are mainly used to determine whether a subprogramme is to be executed at a particular time in the simulation.

In an organisation: The decision tables can be used at

various levels in an organisation. Policies of top management may often be expressed tabularly. Tables may be applied in areas such as engineering, mathematics, personnel, and accounting. Tables allow cross-referencing and more importantly it serves best in documentation.

In Systematics: It is a set of techniques for designing and describing information systems. The basic statements in systamatics namely the elements can be considered as a special case of decision table. This decision table form is more manageble because the entries are limited to combinations of conditions that yield the derivation of only one item.

In Automatic test equipment system: The use of a programming language, based on decision table techniques, permits the test engineer to write test statements easily, and permits programming a test specification with minimal knowledge of programming technique and of the specific test equipment system involved. This type of system can be made to choose a new sequence of tests in accordance with previous results.

In checking the specifications of a system: If decision

tables are used in representing the specification, it is possible to have an automatic checking of the specifications of a software system, which are prepared by the systems analyst from the requirements of the user-customer. It is facilitated by the fact that there is only one unique way for representing the conditions in a decision table, which can be taken as input for an algorithm which is prepared for checking the specifications. One good algorithm was developed in the following chapter for this purpose.

T-H-2363

2.6 AN EXAMPLE

An example of mixed entry decision table is given for a stated problem of declaring students examination results. The flowchart representation was also given for making comparison. This shows that in problems involving complex decision logic decision tables are more convenient. One can easily see that this mixed entry decision table can easily be translated to a limited entry decision table by introducing more number of conditions for each variety of condition entry.

PROBLEM STATEMENT

There are two subjects in the examination called main and ancillary. If a student gets 50 percent or more in

the main subject and 40 percent or more in the ancillary, he passes. If he gets less than 50 percent in the main he must get 50 percent or more in the ancillary to pass. However, the minimum passing marks are 40 percent in the main subject. If a student gets 60 percent or more in the main subject he is allowed to repeat the ancillary subject if the ancillary marks fall below 40 percent. However, there are a group of students in the class who are granted special consideration. Their pass percentage is 40 percent in the main and 40 percent in the ancillary. If they get less than 40 percent in the ancillary they are allowed to repeat that subject if they obtain 40 percent or more in the main subject.

TABLE 2.2

A DECISION TABLE CORRESPONDING TO FLOW CHART OF FIGURE 2.2

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	ELSE
Main marks %	>=50	>=40	>=60	>=40	>=40	
Anc. marks %	>=40	>=50	<40	>=40	<40	
Special status	No	No	No	Yes	Yes	
Pass	x	x	-	x	-	-
Repeat anc.	-	-	x	-	x	-
Fail	-	-	-	-	-	x

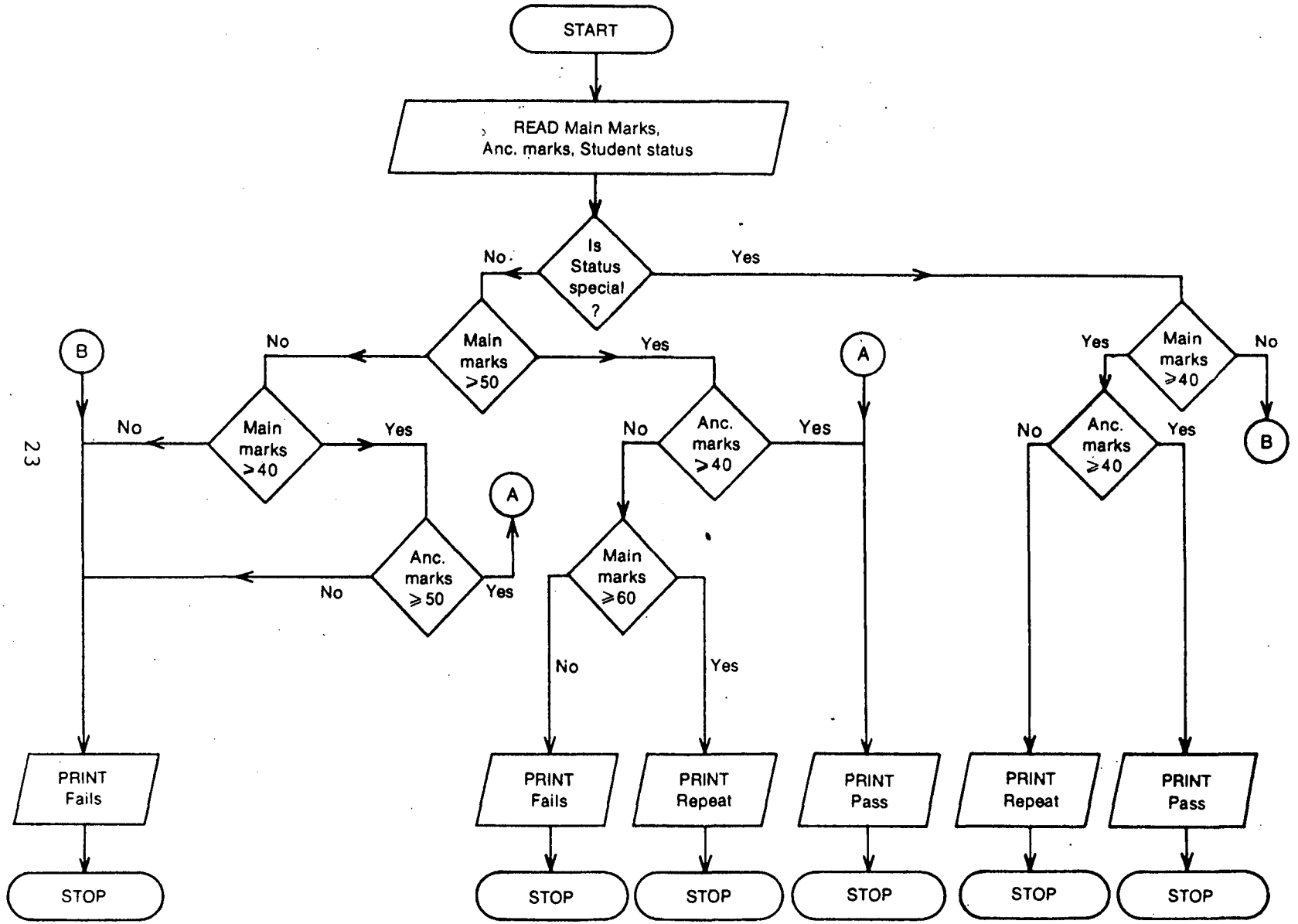

Fig. 2.2 Flow chart depicting examination results processing.

CHAPTER 3

CHECKING A DECISION TABLE

3.0 INTRODUCTION

There are different methods for representing the specifications of a software system. But decision tables have an edge over other methods because it provides an effective means of communication by defining both the problems and their corresponding solutions, especially when the situation demands a number of decision logic conditions. Moreover, with the advent of software packages which produce a program in a computer language, given a decision table. These translators are commercially available.

Moreover, for checking the specifications, it is convenient to use decision tables. Here checking the specification means the verification of the specification for its correctness and completeness, even if there exists logical dependencies between the conditions of a decision table representation. The earlier work in this area is described in Chapter 1. This chapter attempts to develop a more comprehensive and simple algorithm for this purpose.

The algorithm uses some principles of Mathematical logic. In the following section, for greater understanding of the algorithm, a brief discussion about this topic is attempted.

3.1 MATHEMATICAL LOGIC

Mathematical logic is an analytical theory of the art of reasoning, whose goal is to systematize and codify principles of valid reasoning.

A two-valued (either true or false) variables are called logical variables and the operations such as OR operation and AND operation, which are performed between such variables are referred to as logical operators. A variable may stand for a propositional statement.

Some of the needed operators will be introduced here. The truth value of a logical statement having logical operators and variables can be summarized concisely by using a table known as truth table. The truth values of statements having A and B as variables are displayed in Fig. 3.1 for four different logical operators.

A	B	$\overline{A}$	A*B	A+B	$A \rightarrow B$
T	T	F	T	T	T
T	F	F	F	T	F
F	T	T	F	T	T
F	F	T	F	F	T

Fig. 3.1: Truth Table

A logical statement is said to be a tautology provided that it is true for all possible assignments of truth values to its component statements. The following is a list of tautologies which are useful later.

(T_1) : $(p \rightarrow q) * (q \rightarrow r) \rightarrow (p \rightarrow r)$

(T_2) : $v \; v \; p \rightarrow p$

(T_3) : $(p \rightarrow r) * (q \rightarrow r) \rightarrow p + q \rightarrow r$

(T_4) : $(p \rightarrow q) \rightarrow \overline{b} \rightarrow \overline{a}$

(T_5) : $\overline{(p + q)} \rightarrow \overline{p} * \overline{q}$

(T_6) : $\overline{(p * q)} \rightarrow \overline{p} + \overline{q}$

(T_7) : $a \rightarrow b \rightarrow \overline{a} + b$

Any logical function can be written in a standard sum of products form (minterm-form) or in a standard product

of sums form (maxterm form). e.g.,

(i) $f(A,B,C) = A + \overline{B}C$

$= A\,(B + \overline{B})\,(C + \overline{C}) + (A + \overline{A})\,\overline{B}\,C$

$= ABC + AB\overline{C} + A\overline{B}C + A\overline{B}\overline{C} + \overline{A}\overline{B}C$

This standard sum of products form and can be represented as:

$f(A,B,C) = \Sigma m\ (1, 4, 5, 6, 7)$

which is got by assigning '0' for complemented variables and '1' for uncomplemented variables.

(ii) $f(A,B,C) = A + \overline{B}C$

$= (A + \overline{B})\,(A + C)$

$= (A + \overline{B} + C\overline{C})\,(A + B\overline{B} + C)$

$= (A + \overline{B} + C)\,(A + \overline{B} + \overline{C})\,\,(A + B + C)$

This is standard format of sums form and can also be represented as

$f(A,B,C) = \Pi M\ (0, 2, 3)$

which is derived by assigning '1' for complemented variable and '0' for uncomplemented variable.

KARNAUGH MAP

The K-map is a diagram which provides an area to represent every row of a truth table. A K-map for a two variable function is shown in Fig. 3.2.

Row no.	A	B	f
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	1

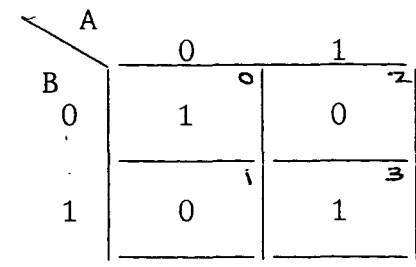

Fig. 3.2: The truth table and K-map for the function

$$f(A,B) = \overline{A}.\overline{B} + A.B.$$

In the K-map each row of the truth table has been transfered to the oppropriate K-map box. Looking at the K-map one can represent the function in minterm and maxterm term notation as

$$f(A,B) = \overline{A}.\overline{B} + A.B = m_0 + m_3 = \Sigma m(0,3)$$

$$f(A,B) = (A + \overline{B}).(\overline{A} + B) = M_1 M_2 = \Pi M(1,2).$$

From the above, it can be easily noted that given a function in minterm specification, one can find it in maxterm specification.

Proceeding on the same lines one can represent a logical function having 3,4,5 or 6 variables. One example will be given in a latter section.

3.2 LOGICAL DEPENDENCIES

The algorithm to be presented in this Chapter, for checking a decision table, first finds all the feasible logical possibilities that the system may assume. These feasible logical possibilities of a decision table is found by traversing the logical dependency relations in the decision table.

Two conditions, C_1 and C_2 are said to be logically dependent if the truth value of C_1 dictates the truth value of C_2 and othrwise they are said to be logically independent. If truth value T of C_1 dictates the truth value T of C_2, then it can be represented as $C_1 \rightarrow C_2$. Similarly, if the truth value T of C_1 dictates the truth value F of C_2, then we have $C_1 \rightarrow \overline{C}_2$.

A set of logical dependency statements can be prepared, if there exists a set of conditions exhibiting dependencies in a decision table. This set of logical dependency statements may be included in the system specification or may be worked out by systems analysts [KING69].

The algorithm to be presented, needs that the logical dependency statements must be minimal and congruent. A set of logical dependency statements is said to be minimal if (a) it embodies all the logical dependencies existing among the conditions in the sense that every dependency is explicitly stated so that no deduction is needed; (b) the antecedent may be composite, but the consequent must be a single condition; (c) the cardinality of the minimal set is minimal.

A set of logical statements is said to be congruent if the following conditions hold whenever possible: (a) every instance of a simple condition has the same truth value; and (b) every instance of a composite condition has the same truth value. For example, the set of logical dependencies $x = [p \rightarrow q, \quad r \rightarrow \bar{p}]$ is not congruent because the two instances of condition p do not

have the same truth value. To make it congruent we can write it as $x = [p \rightarrow q, \; p \rightarrow \bar{r}]$ which is congruent. Sometimes, ascertaining congruency will achieve minimality. For example the set $Y = [a \rightarrow b, \; \bar{b} \rightarrow \bar{a}]$ can be reduced to $Y = [a \rightarrow b, \; b \rightarrow a]$ and further to just $Y = [a \rightarrow b]$, by using mathematical logic tautologies as described in Section 3.1.

A set of logical dependency statements is said to be minimal and congruent if it is both minimal and congruent. Any set is not minimal and congruent, at first instance, can be converted to a set of minimal and congruent logical dependency (MCLDs) statements.

3.3 FEASIBLE LOGICAL POSSIBILITIES

The feasible logical possibilities (FLPs) can be easily found from the MCLDs, as will be explained here. Basically there are four different possibile forms of MCLDs. They are as follows :

Form 1: $p \rightarrow q$

This statement can be rewritten as $\bar{p} + q$.

The feasible logical possibilities can be found by drawing a simple Karnaugh map or these can be found by expressing the

statement in standard sum of products form as follows :

$$
\begin{aligned}
p \rightarrow q &= \bar{p} + q \\
&= \bar{p}(q+\bar{q}) + q(p+\bar{p}) \\
&= \bar{p}.q + \bar{p}.\bar{q} + p.q + \bar{p}.q \\
&= \bar{p}.q + \bar{p}.\bar{q} + p.q \\
&= [00,\ 01,\ 11]
\end{aligned}
$$

Here 1 is used to represent T and 0 to represent F. This notation will be followed here afterwards.

Form 2: $p + q \rightarrow r$

In the same way as mentioned above we can proceed

$$
\begin{aligned}
p + q \rightarrow r &= \overline{p+q} + r \\
&= \bar{p}.\bar{q} + r \\
&= \bar{p}.\bar{q}(r + \bar{r}) + r(p+\bar{p})(q+\bar{q}) \\
&= \bar{p}\ \bar{q}\ r + \bar{p}\ \bar{q}\ r + p\ q\ r + \bar{p}\ q\ r \\
&\qquad + p\ \bar{q}\ r + \bar{p}\ \bar{q}\ r \\
&= [000,\ 001,\ 011,\ 101,\ 111]
\end{aligned}
$$

The total number of FLPs here are five.

Form 3: $p.q \rightarrow r$

Proceeding in the same way as for Form 2, we get FLPs as [000, 001, 010, 011, 100, 101, 111]. So there are seven FLPs for this form.

An algorithm is developed for finding feasible logical possibilities, given the MCLDs. The set of MCLDs are treated as if they are 'and'ed together. This can be implemented on the computer very easily.

An example for illustrating the use of this algorithm is given here. The decision table and its MCLDs are given in Figure 3.2.

The MCLDs are 'and'ed as follows :

$$(p \rightarrow q).(q \rightarrow r).(p \rightarrow r).(s \rightarrow \bar{t}).$$

This expression can be rewritten as

$$(\bar{p} + q).(\bar{q} + r).(\bar{p} + r).(\bar{s} + \bar{t}).$$

This can be plotted on a Karnaugh map to get all the feasible logical possibilities. As the above expression is in the product of sums form, the maxterm representation is convenient to get the list of FLPs.

The K-map representation is shown in figure 3.4.

Condition name	Condition	Rule 1	2	3	4	5	6
p	M > 100	Y	N	N	N	N	N
q	M > 65	$	Y	Y	N	N	N
r	M > 19	$	$	$	Y	N	N
s	N < 40	-	Y	N	N	Y	N
t	N > 70	-	*	N	-	*	N
	Action	A_1	A_2	A_3	A_4	A_5	A_6

(a) The decision table

$p \rightarrow q$

$q \rightarrow r$

$p \rightarrow r$

$s \rightarrow \bar{t}$

(b) MCLDs

Fig. 3.3: The decision table and MCLDs of the example.

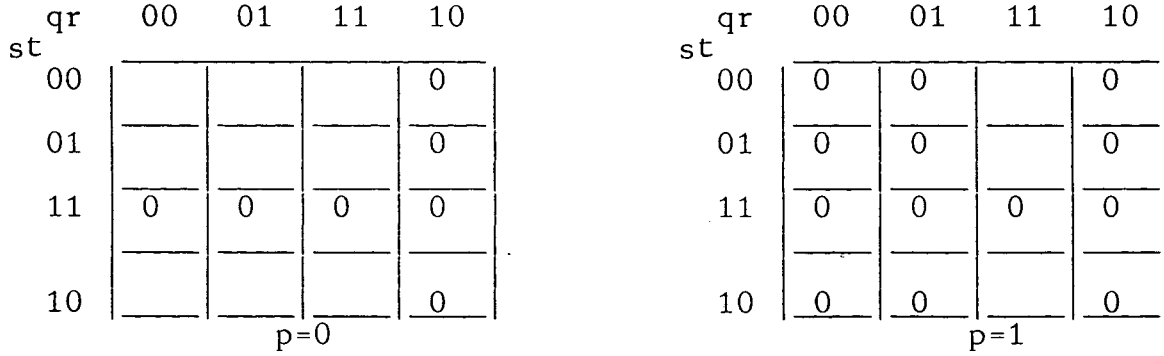

Fig. 3.4: K-map for the example.

The resulting maxterm representation is

$$f(p,q,r,s,t) = \prod M\ (3,7,8,9,10,11,15,16,17,18,19,20,21,22,23,24,25,26,27,31).$$

The same function can be written in minterm representation as follows :

$$f(p,q,r,s,t) = \sum m(0,1,2,4,5,6,12,13,14,28,29,30)$$

This corresponds to the following list of FLPs:

FLPs = [00000, 00001, 00010, 00100,
00101, 00110, 01100, 01101,
01110, 11100, 11101, 11110]

The total number of FLPs are 12 for the considered decision table.

3.4 THE ALGORTHM TO CHECK THE DECISION TABLE

In section 3.3, we got the feasible logical possibilities, which is representative of the possible states that the system can assume. In this section an algorthm will be presented for actually checking the decision table for completeness and correctness using these FLPs.

The algorithm is implemented in PASCAL on VAX-11/780. The choice of PASCAL is becasue it has a clean control structure and a rich variety of data representations.

The input to the algorithm is the decision table and its MCLDs. The algorithm first finds FLPs from MCLDs and FLPs stipulated by the decision table. The algorthm checks the decision table for its correctness and points out missing logical possibilities, if the decision is correct but not complete. The decision table shown in figure 3.3 of the section 3.3. is used, as a running example, to illustrate the algorithm.

STEP 1: Find all FLPs from MCLDs as described in section 3.3 and find their total number also. Let us denote them by MCLDSET and MCLDTOTAL respectively. Goto Step 2.

For the decision table of figure 3.3 the MCLDSET found to be (in section 3.3)

```
MCLDSET = [00000, 00001, 00010, 00100,
           00101, 00110, 01100, 01101,
           01110, 11100, 11101, 11110]

MCLDTOTAL = 12.
```

STEP 2: Find all the FLPs stipulated by the rules of a decision table. This can be obtained by scanning each rule, i.e., a column of condition entries of a decision table. If a rule has K 'don't care' terms, then 2 ** K row vectors are obtained, each for a distinct binary combination of the 'don't care' condition. Let us denote this set of FLPs from decision table by DTSET and the total number of FLPs is denoted by DTTOTAL and this number can also be found by

$$DTTOTAL = 2^{**}K_1 * 2^{**} K_2 * - - - * 2 ** K_n.$$

where n is the total number of rules and K_i is the number of 'don't care' terms in the rule i. Goto Step 3.

Applying these to our example decision table we get,

```
DTSET = [00000, 00010, 00100, 00101,
         01100, 01110, 11100, 11101,
         11110, 11111]
```

and DTTOTAL = 10.

STEP 3 : If MCLDTOTAL = DTTOTAL and MCLDSET = DTSET, then the table is complete and correct. If it is so, terminate the algorithm, otherwise goto Step 4.

For the example the above conditions are not true, so, proceed to step 4.

STEP 4: If DTSET ≠ MCLDSET, then find

ADDITIONAL = DTSET-MCLDSET

where '_' has the same connotation as in Pascal language set. If A & B are sets, mathematically,

A-B = [x | (x in A) and (x not in B)].

Goto Step 5.

For our example it can be writen as

ADDITIONAL = [11111]

STEP 5: If DTSET ≠ MCLDSET, then find

MISSING = MCLDSET - DTSET.

Goto Step 6.

For the example, the MISSING will be

MISSING = [00001, 00110, 01101]

STEP 6: If the set additional is having only the rules that are obtained by the expansion of the 'don't care' conditions of rules; then ADDITIONAL is not considered dangerous. If it is so goto step 7 else goto step 9.

For our example the ADDTIONAL is contained only in the 'don't care' of rule 1, so, goto step 7.

STEP 7: If DTTOTAL ≤ MCLDTOTAL and DTSET ≠ MCLDSET, then the table is correct but not complete. An else-clause

needs to be added to include the MISSING possibilities. Goto Step 8.

For the example, $10 \leqslant 12$ is true. So, we can conclude that the table is correct but not complete. An else class needs to be added to include the three logical possibilities, [00001, 00110, 01101].

STEP 8: If DTTOTAL > MCLDTOTAL then if MISSING is empty then the table is correct and complete else the table is correct but not complete. Terminate the algorthm.

It is not applicable to the example.

STEP 9: The table is neither complete nor correct. Terminate the algorithm.

For the example it is not applicable.

3.5 **ANOTHER EXAMPLE**

The second example is taken from [KING68]. In many business data processing situations the conditions are highly related. For example, instalment buying where payments are made in cash on weekly basis, the action taken when an account goes into arrears is a crucial aspect

of the operation. The figure 3.5 shows a simplifiesd arrears procedure. It is seen that the first three conditions are directly related. Thus a No out come for the third condition implies No out comes to the first two conditions. The last two conditions are also directly related.

	Rule								
Condition	1	2	3	4	5	6	7	8	9
p this week's cash greater weekly rate	Y	Y	N	N	*	*	*	*	*
q this week's cash greater than 0	$	$	Y	Y	*	*	N	N	N
r any cash during last 3 weeks	$	$	$	$	N	N	Y	Y	Y
s arrears greater than 3 * weekly rate	-	$	Y	N	-	$	N	Y	$
t arrears greater than 6* weekly rate	N	Y	N	*	N	Y	*	N	Y
Actions									
Send arrears letter A		X							
Send arrears letter B			X						
Send arrears letter C					X				
Send arrears letter D								X	
Note account	X			X			X		
Take special arrears action						X			X

Fig. 3.4: The decision table for second example. (Source:[KING 68])

The set of MCLDs can be writen as

$$\bar{r} \rightarrow \bar{p}$$

$$\bar{r} \rightarrow \bar{q}$$

$$\bar{q} \rightarrow \bar{p}$$

$$t \rightarrow s$$

We can find MCLDSET and MCLDTOTAL as described in section 3.3.

```
MCLDSET = [00000, 00010, 00011, 00100,
           00110, 00111, 01100, 01110,
           01111, 11100, 11110, 11111].
```

So MCLDTOTAL = 12

As explained in the step 2, we can expand all the rules in the decision table to get the elementary rules. This set, called DTSET, can be writen as.

```
DTSET = [00000, 00010, 00011, 00100,
         00110, 00111, 01100, 01110,
         11100, 11110, 11111].
```

DTTOTAL = 11.

Since, MCLDTOTAL $\neq$ DTTOTAL, we proceed to find ADDITIONAL and MISSING.

```
ADDITIONAL = DTSET - MCLDSET
           = [ ]
```

MISSING = MCLDSET - DTSET

= [01111]

The ADDITIONAL is empty, we conclude that the table is correct but not complete. The missing feasible logical possibility is [01111]. This is the case when a customer's current week payment is greater than zero but less than weekly rate and his arreares is greater than six times weekly rate.

So, the arrears procedures specified by the decision table of figure 3.5 is therefore correct but not complete.

We have shown that the possible logical possibilities of a decision table can be obtained by our algorthm. The method will aid the system analyst performing the validation of system modelling.

CHAPTER 4

A STUDY OF COMPLEXITY MEASUREMENTS

4.0 INTRODUCTION

Increasing importance is being attached to the idea of measuring software characteristics. It is only by such a process of measurement that it will be possible to determine whether new programming techniques are having the desired effect in reducing the problems of reliable software production. Unfortunately, many of the qualities of interest such as clarity, ease of testing and maintenance, etc. are highly subjective and so experiments have been performed to correlate subjective grading of programs with measured structural characteristics of source programs. But quantification is a must in making programming a science rather than an art. So attempts are made to evolve software metrics which are to be used to measure and predict software quality. Several software metrics have been developed to measure various kinds of software properties, such as the complexity measure, stability measure [YOU85], reliability measure [GOEL85], reusability measure [PRES83], etc.

Complexity measures offer great potential for containing the galloping cost of software development and maintenance [KEAR86]. This can be used for cost projection, manpower allocation and program and programmer evaluation. Despite the growing body of literature devoted to their development, anlysis and testing, software complexity measures have yet to gain wide acceptance. Nonetheless, new complexity measures continue to appear, and new support for old measures is earnestly sought. Until more comprehensive evidence is available, software complexity measurements should be used very cautiously.

Here in this Chapter an attempt is made to highlight the importance of complexity measurement and a comparative study is made to evaluate different complexity metrics.

4.1 WHAT IS COMPLEXITY?

Basili defines complexity as a measure of resources expended by another system in interacting with a piece of software to perform a given task [BSLI80]. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. This type complexity can be termed as computa-

tional complexity (or dynamic complexity) which is not of our concern in this chapter.

If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing, or modifying the software. In our discussion, the software complexity is used to indicate this difficulty level, i.e., the difficulty present in the interaction between a program and the programmer working on that programming task. In simple words complexity is a measure of how difficult the program to comprehend and work with. This can be termed as structural complexity (or static complexity].

Usually these measures are based on program code disregarding comments and stylistic attributes such as indentation and naming conventions. Measures typically depend on program size, control structure or the nature of module interfaces. Many complexity measures will be introduced in latter sections.

4.2 IMPORTANCE OF SOFTWARE COMPLEXITY

As software scientists attempt to understand software processes and products, it is natural for them to

characterize and measure those aspects of programs that seem to affect cost. Software maintainability is the degree to which characteristics that impede maintenance are present. The costs for software maintenance activities have been observed to outweigh the development costs and take a greater share of the total software budget for many organizations than development costs. This maintainability is driven primarily by software complexity. Their relationship is roughly depicted in Fig. 4.1 [HFLI87].

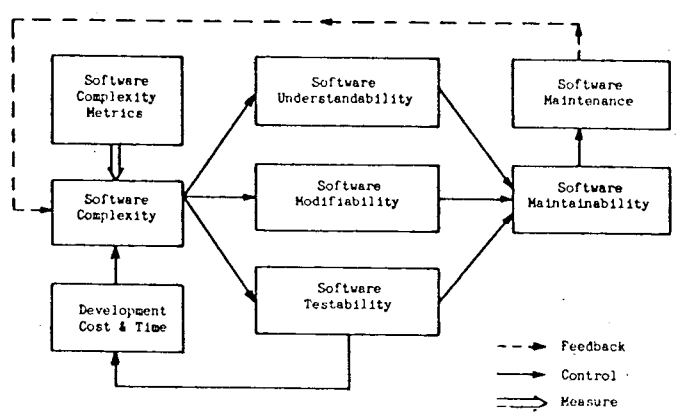

Fig4.1 Importance of software complexity.

According to C.V. Ramamoorthy, the metrics should be applied at requirements phase to predict cost, at design phase to guide the decomposition process, and at coding phase to estimate testing time required [RAMA85].

Dennis Kafura and G.R. Reddy in their paper [REDY87], which is a study of the relationship between complexity metrics and software maintenance, concludes:

[1] that the growth in system complexity as described by the software metrics agree with the general character of maintenance tasks performed;

[2] the metrics were able to identify the improper integration of functional enhancements made to the system;

[3] the complexity values of the system components as indicated by the metrics confirm well to an understanding of the system as people familiar with the system;

[4] Metrics are useful in redesign phase, as they reveal any poorly structured component that may be present in the system.

Advocates of software complexity metrics have suggested that these tools can be used to predict program

length, program development time, number of bugs, the difficulty of understanding a program and the future cost of program maintenance.

Furthermore, Basili [BS480] gives three possibilities for using complexity metrics :

[a] To evaluate the software process and product: a low score on a metric like the number of errors, indicates something desirable about the quality of the process while a high score on the same metric indicates something quite undesirable about the product.

[b] As a tool for software development: In this case, the metric can act as feedback to the developer, telling him to know how the development is progressing. It can be used to predict where the project is going by estimating future size or cost, or it may tell him his current design is too complicated and unstructured.

[c] To monitor stability and quality of an existing product: One can periodically recalculate a set of metrics to see if the product has changed character in some way. It can provide a much needed feedback during maintenance period.

As the complexity metrics are yet to be standardised, these measures should be used very cautiously until more comprehensive evidence is available.

4.3 COMPLEXITY METRICS

In his letter entitled "Goto Statement Considered Harmful", Dijkstra observed that the "quality of programmers is a decreasing function of the density of goto statements" [DIJK68]. This suggests then a very simple measure for complexity, namely the number of gotos in a program. Whilst this may be useful as a measure of unstructuredness for some languages [eg. Pascal, Algol) it is not for others (e.g., Fortran).

Since then, many complexity metrics have been developed and they can be classified into two basic types: (1) static and (2) dynamic. In the former case, measurement of the product is done by static analysis of the source code, while in the latter case, it is collected at run time and may vary from one execution to the other. Here, the attention will be concentrated on static measures which can in turn be devided into three types:

[1] volume: measure the size of a product

[2] Data organization: measures the usage and visibility of data as well as their interactions.

[3] Control organization: measures the comprehensibility of control structure.

Classification of complexity metrics using some common measures of interest is shown in Fig. 4.2. Most of these measures have been used in some way but do not gain full acceptance partly because it is not certain what aspects of the software life cycle the metrics describe and partly because of the difficulty in parameterization.

Many reports are published by researchers for the emperical evaluation of different complexity metrics [BSL183a; BSL183b; CURT79].

I. VOLUME METRICS

This conventional volume metrics are straightforward and widely applicable. This is based on program size, which by virtue of the complexity involved in the volume of information that must be absorbed to understand the problem.

The definable measures of the volume of a program are number of lines (LINES), number of executable statements (STMTS), number of programming units like subroutines (UNITS), average length of a programming module (STM/U), etc.

II. HALSTEAD'S SOFTWARE SCIENCE

It is one of the most well-known complexity metric with several emperical studies. The Halstead measures are functions of the number of operators and operands in the program [HALS77].

Operators fall into three classes and for FORTRAN language, for example, the list will be as follows :

(a) Basic - + - * ** ; / // = () .GT. .GE. .LT. .LE. .NE. .EQ. .NOT. .AND. .OR. .EQV. .XOR. .NEQV.

(b) Keyword - IF THEN ELSE ELSEIF ENDIF DO DOWHILE GOTO ASSIGN CONTINUE ENDDO RERD WRITE TYPE PRINT ACCEPT EOS

(c) Special - Names of subractines, functions.

Operands consist of all variable names and constants such as, TRUE, FALSE and Esm (real). The Halstead's metrics can be defined on the basis of

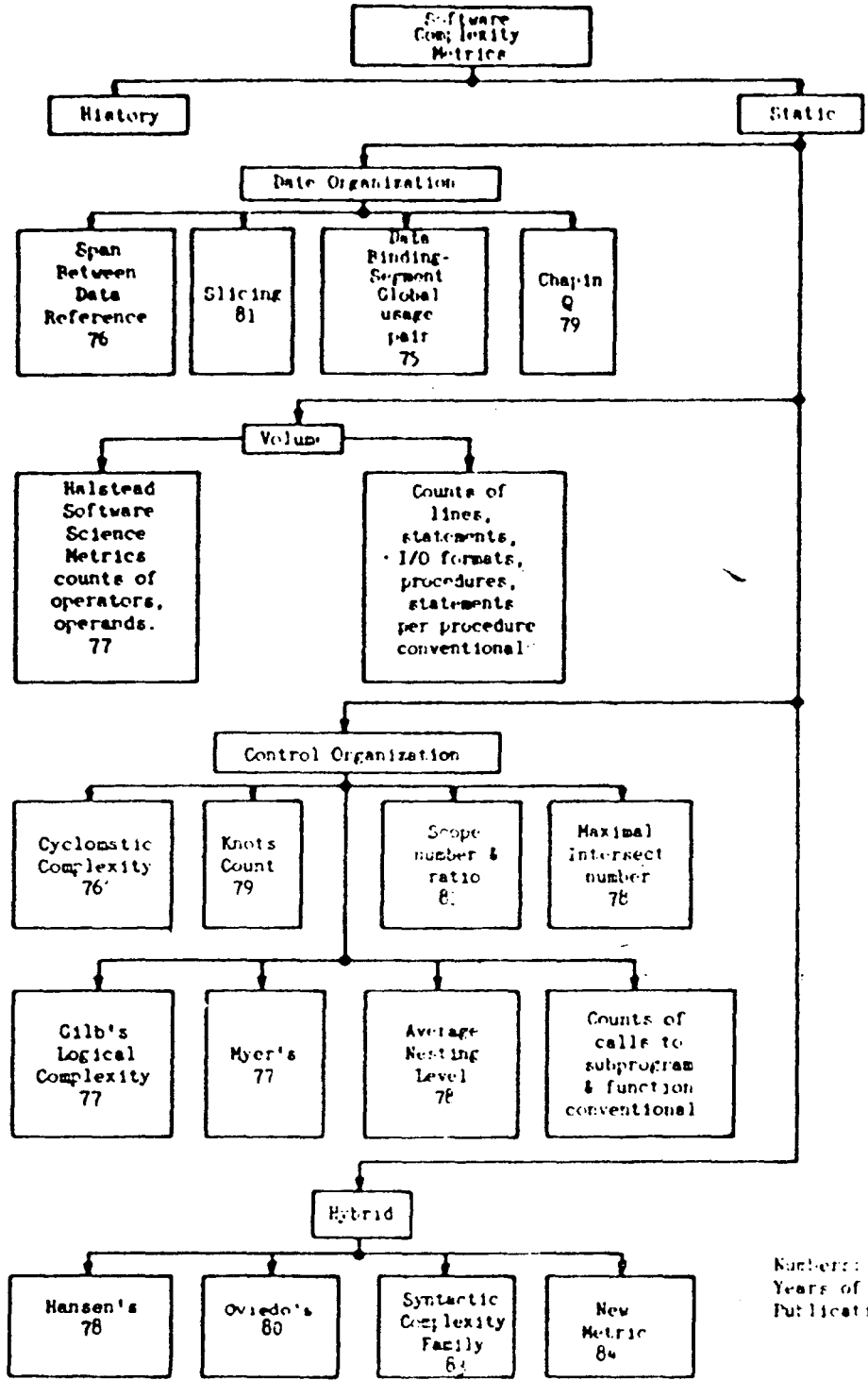

Fig.4.2 Classification of complexity metrics.

number of unique operators n_1,

number of distinct operands n_2,

total number of operators N_1, and

total number of operands N_2.

Then Halstead defines the vocabulary of the program as

$$n = n_1 + n_2$$

and implementation length as

$$N = N_1 + N_2 .$$

He hypothesizes on estimator $N' = \log_2 n_1 + n_2 \log_2 n_2$.

A program volume metric V, which characterizes the size of an implementation, as $V = N \log n$

The potential value $V^* = n^* \log n^*$ represents the minimum algorithm representation in a language where the required operation is builtin. Hence, the potential vocabulary $n^* = n_1^* + n_2^* \geq n_2^* + n_2^*$ because in such a minimal form, the number of operators is two: the algorithm name and ().

To evaluate the programming effort, propensity of error, and ease of understanding, the program level L of an

implementation is defined as V^*/V, which has the maximum value of unity and can be approximated by $L' = 2.n_2/n_1/N_2$. It follows that only the most succinct expression can have a level of unity. Program difficulty D is the difficulty of coding an algorithm. $D = 1/L$ by definition and can be estimated by $D_2 = 1/L'$.

Halstead hypothesizes that LV remains invariant under translation from one language to another. LV can therefore be regarded as the intelligence context IC of the algorithm which increases only as the complexity of problem solution increases.

The effort required to generate an algorithm is $E = V/L$. It is suggested that E can measure the effort required to comprehend an implementation and is a measure of clarity. Effort E can be approximated by

$$E' = \frac{V}{L'} = \frac{n_1 N_2 .N \log n}{2.n_2} \quad \text{(or)} \quad E'' = \frac{n_1 N_2 .N' \log n}{2.n_2}$$

III. GRAPH-THEORETIC METRICS:

A program can be represented by a flow graph, G=(V,E), where V is a set of nodes and E is a set of edges

Node - A sequential block of code with unique entrance and exit but no internal branch or loop.

Edge - Flow of control between the various nodes.

For an edge (u,v), node u is the initial node and node v is the terminal node. The outdegree of node u is the number of edges emanating from u; the indegree of node is the number of edges incedent at u. Using this flow graph concept, various control metrics can be constructed, which characterizes the control complexity of a given flow graph.

(a) McCabe's Complexity Metric:

McCabe's cyclomatic complexity [MCAB76] is well accepted, intutively reasonable, and easily calculated. The metric V(G) is essentially the cyclomatic number of the program graph +P; where P is the number of strongly connected components of the program graphs (also called units in volume metrics). It is given by

$$V(G) = e - n + 2P$$

where e is the number of edges and n is the number of vertices of the program graph.

In a strongly connected graph, this cyclomatic number is the number of linear independent circuits. For programs with single entry and single exit, V(G) is one plus the number of decisions (that is number of predicates).

This graph-theoritic metric is independent of the program size but depends only on the decision structure. Decision making of a program affects its error probability and development time and cost.

(b) Gilb's metrics:

Gilb gives [GILB77] two metrics: CL, absolute logical complexity (number of binary decisions) and cL, relative logical complexity (ratio of CL to STMTS) which have been supported by some empirical evidence. The latter can be considered as an improvement over pure control metrics as it also takes into account some volume metric. He gave some conventional metrics also like CALLS (the number of subroutine and function invocations); CA+BD (the total number of calls and binary decisions) etc.

(c) KNOT Count:

A Knot occurs when two control transfers intersect, as depicted in Fig. 4.3. since each node is a

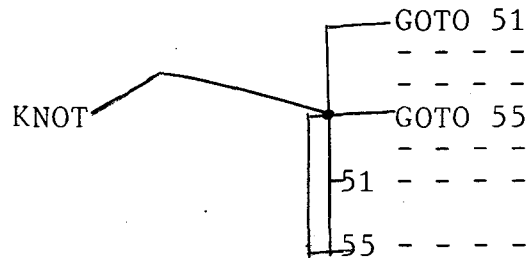

Fig. 4.3: KNOT EXAMPLE

sequence of statements, with no internal branches, a Knot occurs if node b includes at least one line in the example [WOOD79]. Two related metrics can be further defined :

1. KNOT1 - The number of Knots that can be verified.

2. KNOT2 - The total number of potential Knots, assuming every node contains one statement.

It is conceivable that a program with many knots is more complex to comprehend to reflect this.

(d) SCOPE Metric and Ratio:

Nodes with an outdegree 0 or 1 are RECEIVING nodes. Those with an outdegree greater than 1 are SELECTION ones. Given a selection node, we can find atleast one "lower bound" node which succeeds every immediate successor of the selection node. The lower bound node that precedes

every other lower bound is the GREATEST LOWER BOUND (GLB). The number of nodes preceding the GLB and succeeding the selection node, plus 1, yields the ADJUSTED COMPLEXITY (AC) of the selection node. It reflects the scope of "influence" of the selection node. Summing up the adjusted complexity of each node, the SCOPE metric is formed [HARS81a,b].

SORT, the scope ratio metric, is defined as:

(1.0 - N/SCOPE)*100%,

where N = number of nodes in the flow graph excluding terminal node.

SCORT increase towards 100 (percent) as complexity increases.

III LI's HYBRID METRIC

This is a hybrid metric [HFLI87], which integrates software science with the scope of measure and reflects both volume and control organization. The raw complexity of a node V_j is E'_j :

$$E_j \triangleq N_j \log n_j / L'_j$$

where N_j, n_j are local parameters of node V_j, and L' is a global parameter defined previously in this section.

The adjusted complexity for a selection node is the sum of E^{j}' values of every node within the scope of that selection node, plus the value of the selection node itself. A receiving node has an adjusted complexity equal to its raw complexity. The complexity of the overall program is the sum of the adjusted complexities of every node.

The metric can be defined as:

(1.0 - Σ Raw complexities/ Σ adjusted complexities) *100%

This increases towards 100 (percent) as complexity increases.

4.4 VALIDATION OF DIFFERENT METRICS

In the earlier section we have seen several metrics based on characteristics of the software product, which appeared in the literature. Many studies have applied them, to data, from various organisations to determine their validity and appropriateness. However, the question of how well the various metrics really measure or predict effort or quality is still an issue in need of confirmation. Studies

across different environments have been done to answer this question [BSLI83a, BSLI83b, CURT79, HFLI87]. The results of these studies will be discussed here. For validating a metric one has to examine many software projects. This type of work created yet another new field namely Experimental Computer Science.

The first question that is to be answered in this direction is "what are the properties of a good metric?". [KEAR86] says that complexity measures should be graded by its robustness, normativeness, specificity and prescriptiveness. Robustness of a measure means that the metric should be responsive to program modifications and it should show that a reduction in the measure consistantly produce improvements in the program. Normativeness means that the measure should facilitate to provide a norm (a particular figure of complexity) against which measurements can be compared to reject programs having unacceptable levels of complexity. Specificity is the degree to which a measure is able to point out the deficiencies in program construction. The word prescriptiveness means that the ability of the metric to suggest methods to reduce the complexity of a overly complex program.

One of the earliest work done on validation of complexity metrics is [CURT79a]. It reports empirical evidence to show that metrics were related to difficulty programmers experience in understanding and modifying software. But the correlations observed are not as high as those claimed by Halstead. The Halstead and McCable metrics provided some information about program differences, but there were other factors unassessed by these metrics which influence the psychological complexity of the programs. The metrics reportedly predicted programmer performance better on versions of programs which were unstructured or unconnected. Further, neither Halstead's nor McCabe's metrics consider the level of nesting within various constructions (eg. three DO loops in succession will result in metric values similar to those for three nested DO loops).

It also reported the detection of curvilinear relationship between Halstead's E and performance. From this one can conclude that as Haltead's E grows larger a program becomes more psycologically complex, but the increments in difficulty grow smaller and smaller [CURT79B].

TABLE 4.3

Intercorrelations Among Complexity Metrics as Reported by [CURT79b]

Measure	CORRELATIONS	
	E	V(G)
Subroutine :		
V(G)	0.92	
Length	0.89	0.81
Program:		
V(G)	0.76	
Length	0.56	0.90

Note: n=27 and P <= 0.001

Basili in his paper [BSLI83] also reported that none of the metrics examined manifests a satisfactory explanation of effort spent developing software or the error incurred during that process. In this evaluation the effort spent is actually found from interviews and reports of the programs involved. The major results of the investigation are listed below :

[1] Neither software science's E metric, cyclomatic complexity nor source lines of code relates convincingly better with effort than the others;

[2] The strongest effort correlations were derived when models obtained from individual programmers or certain validated projects were considered;

[3] The majority of the effort correlations increase with more reliable data;

[4] The number of revisions appears to correlate with development error better than either software science's B metric, E metric, cyclometric complexity or source lines of code; and

[5] Although some of the software science metrics have size dependent properties with their estimators, the metric family seems to possess resonable internal consistency.

H.F. Li and W.K. Cheung have developed a Fortran static source code analyzer [FORTRANAL] to study 31 metrics on a data base of 255 student programs [HFLI87]. This study is the most comprehensive of all the emperical studies available today. The results of this study are summarized in Table 4.4 in which correlation coefficients between every

possible pair of metrics are tabulated for 18 selected metrics. They made the following remarks.

The Halstead's family of metrics reported to possess reasonable internal consistancy, i.e., with correlation coefficient clause to units, as can be seen from Table 4.4. This suggests that one of them can replace the other in application. The length equation $N' = n_1 \log n_1 + n_2 \log n_2$ appears to be program-size dependent and N' tends to be high for small programs and low for larger ones.

McCabe's cyclometric measure correlates well with Halstead's, Gilb's, Knot counts, SCOPE, EDGES and NODES metrics. This measure can be viewed as a control

TABLE 4.4
CORRELATION COEFFICIENTS AMONG 18 SELECTED METRICS

	STMTS	LN-CM	NODES	EDGES	McCBE	SCOPE	n2	N1	N2	n	N	N^	V	IC	E^	E^^	CL
STMTS																	
LN-CM	.983																
NODES	.924	.906															
EDGES	.914	.875	.982														
McCBE	.908	.891	.964	.971													
SCOPE	.848	.797	.910	.947	.892												
n2	.898	.877	.896	.889	.872	.826											
N1	.977	.971	.916	.898	.905	.833	.925										
N2	.942	.933	.917	.903	.915	.828	.953	.976									
n	.907	.893	.920	.899	.886	.832	.987	.933	.950								
N	.968	.960	.921	.906	.915	.836	.943	.996	.992	.946							
N^	.896	.878	.913	.898	.881	.837	.989	.925	.947	.998	.940						
V	.960	.949	.927	.914	.918	.852	.956	.990	.992	.959	.957	.958					
IC	.865	.834	.810	.824	.796	.780	.956	.882	.891	.907	.891	.912	.900				
E^	.914	.913	.905	.873	.897	.805	.845	.940	.937	.884	.944	.880	.947	.728			
E^^	.886	.882	.917	.881	.892	.813	.887	.914	.925	.931	.924	.931	.938	.748	.976		
CL	.878	.830	.930	.978	.969	.932	.851	.862	.872	.848	.872	.850	.880	.803	.830	.828	
KNOT2	.871	.830	.919	.948	.923	.877	.855	.861	.872	.848	.871	.845	.873	.815	.803	.799	.943

organization metric (i.e., number of control paths) and to a lesser extent, a volume metric (i.e., number of decisions +1). So the cyclomatic measure seems to bridge the gap between the two categories (VOLUME and control organisation metrics).

The SCOPE number is reported to be not always reliable because the scope number, in essence, is dependent on the no. of nodes in the flow graph. Some programs can be rearranged to give flow graphs with different scope measures. The SCORT and Li's hybrid metric are found to correlate well with each other.

It was noticed that KNOT1 count is much less than KNOT2. And the KNOT2 is found to be much better correlator than KNOT1 with volume metrics.

Similar to KNOT metrics, the absolute logical complexity CL correlates better with those tradinal metrics than the relative logical complexity CL, which takes into account the program size. In fact, CL is the number of binary decisions in the program's logic and can be regarded as a special volume metric.

Regarding all the volume metrics, the number of executable statements (STMTS) is found to be the best one, which correlates well with Halstead's N and V measures.

Volume metrics vs. control organization metrics:

In general, metrics based on measures of program size, have been the most successful to date, with experimental evidences indicating that larger programs have greater maintenance costs than smaller ones. But this technique is not adequate, which can be demonstrated by imagining a 50 line program consisting of 25 consecutive "IF THEN" constructs. Furthermore, volume metrics can only be measured after the design has been carried out fully to the debugged code, making it difficult to take any corrective action at the implementation stage.

As reported by [HFLI87] several control organisation metrics correlate well with value metrics, e.g., McCabe, SCOPE, CL and KNOT2. In general, the control flow metric fails to be comprehensive and do not consider the contribution of any factor except control flow complexity. However, these metrics, can differentiate between two programs of similar volume metrics and certainly related to software quality.

Hybrid metrics attempt to remedy one of the shortcomings of single factor complexity metrics in use. Li's hybrid metric combines a measure of control flow and program size, i.e., SCORT and E' are considered together[1]. The resulting hybrid metric was found to be slightly different from SCORT measures.

Most of the metrics are lacking of context sensitivity. For example, EDGES, NODES, McCabe and CL consider only the node and edge counts and fail to consider the context of each edge and node. Halstead's metrics too cannot take into account the flow of control. Hence, most metrics lack comprehensiveness.

Metrics relation to errors:

[CURT79b] reports from their experiments that the software complexity metrics developed by Halstead and McCabe are related to the difficulty programmer experience in locating errors in code. They can be used in providing feedback to programmers about the complexity of the code they have developed and to managers about the resources that

---

The stated reason for the choice [HFLI82] is that they are orthogonal, i.e., they measure different aspects of complexity and give correlation coefficient by them as only -0.032.

will be necessary to maintain particular section of code. Code which is more complex may also be more error-prone and difficult to test.

Basili et al. [BSLI83] in their study report that the most of the correlations between metrics and errors and weighted errors are very weak with the exception of system changes. These disappointingly low correlations attribute to the discrete nature of error reporting. However, they report that partitioning an error analysis by individual project or programmer shows improved correlation with the various metrics.

4.5 CONCLUSION

Software complexity measures have not realized their potential for the reduction and management of software cost. This failure derives from the lack of a unified approach to the development, testing and use of these measures.

Complexity measures currently available provide only a crude index of software complexity. Advances are likely to come slowly as programming behaviour becomes

understood. Users of complexity measures must be aware of the limitations of these measures and approach their applications cautiously. Before a measure is incorporated into a programming environment, the user should be sure that the measure is appropriate for the task at hand. The measure must possess the properties demanded by the use. Finally, users should always view complexity measurement with critical eye.

REFERENCES

[BSLI80a] V.R. Basili, Tutorial on models and metrics for software management and engineering, IEEE Computer Society Press.

[BSLI83b] V.R. Basili and D.H. Hutchens, 'An empirical study of a syntactic complexity family,' IEEE Trans. Software Eng., vol. SE-9, pp. 664-672, Nov. 1983.

[BSLI83c] V.R. Basili, R.W. Selby, Jr., and T.Y. Philips, 'Metric analysis and data validation across Fortran projects,' IEEE Trans. Software Eng., vol. SE-9, pp. 652-663, Nov. 1983.

[CURT79a] B. Curtis, S.B. Sheppard and P.M. Millman, M.A. Borst and T. Love, 'Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics,' IEEE Transactions on Software Engineering, 5.2, pp. 95-104, March 1979.

[CURT79b] B. Curtis, S.B. Sheppard and P. Millman, 'Third time charm: stronger prediction of programmer performance by software complexity metrics,' Proc. 4th International Conference on Software Engineering, pp. 356-360, Sept. 1979.

[DIJK68] E.W. Dijkstra 'Goto Statement Considered Harmful,' Commun. of ACM, vol. 11, pp.147-148, 1968.

[GILB77] T. Gilb, 'Software metrics,' Cambridge, MA: Winthrap 1977.

[GOEL85] A.L. Goel, 'Software reliability models: assumptions, limitations and applicability,' IEEE Trans Software Eng., vol.SE-11, pp. 1411-1423, Dec. 1985.

[HALS77] Maurice H. Halstead, 'Elements of Software Science,' Elsevier North-Holland, New York, 1977.

[HARS81a] W. Harrison and K. Magel, 'A complexity measure based on nesting level,' ACM SIGPLAN Notices, pp. 63-74, Mar. 1981.

[HARS81b] W. Harrison and K. Magel, 'A topological analysis of computer programs with less than three binary branches,' ACM SIGPLAN Notices, pp. 51-63, Apr. 1981.

[JENS74] K. Jensen and N. Wirth, PASCAL User Manual and Report : Lecture Notes in Computer Science. vol. 18, Springer-Verlag: Berlin, Germany, 1974.

[KEAR86] Josph K. Kearney et al., 'Software complexity mesurement,' Commun. ACM Vol. 29, no. 11, pp. 1044-1050, Nov. 1986.

[KING68] P.J.H. King, 'Ambiguity in limited entry decision tables,' Commun. ACM, Vol. 11, No. 10, pp. 680-684, Oct. 1968.

[KING69] P.J.H. King, 'The interpretation of limited entry decision table format and relationships among conditions,' Computer J., vol. 12, Nov. 1969, pp. 320-326.

[MCAB76] T.J. McCabe, 'A complexity measure,' IEEE Transactions on software engineering, vol. 2, no. 4, pp. 308-320, December 1976.

[MYER77] G.L. Myers, 'An extension to the cyclomatic measure of program complexity,' SIGPLAN Notices, Vol. 12, pp. 61-64, Oct. 1977.

[POLL71] S.L. Pollack, H.T. Hicks and W.J. Harrison, 'Decision Tables: Theory and Practice,' New York: Wiley, 1971.

[POOC74] U.W. Pooch, 'Translation of decision tables,' ACM Computing Surveys, vol. 6, no. 2, pp. 125-151, June 1974.

[PRES83] P.E. Presson, 'Software interoperability and reusability guide-book for software quality measurement,' Rome Air Development Centre, Griffies Air Force Base, NY, Rep. RADC-TR83-174, July 1983.

[RAMA84] C.V. Ramamoorthy and Vick, Handbook of Software Engineering, Van Nostrand Reinhold Company Inc., 1984.

[RAMA85] C.V. Ramamoorthy et al., 'Metrics guided methodology,' Proc. COMPSAC, 1985.

[RAMA86] C.V. Ramamoorthy, "Programming in the large", IEEE Trans. on Software Engineering, vol. SE-12, no. 7, July 1986.

[RAJA78] M. Ibramsha and V. Rajaraman, 'Detection of logical errors in decision table programs,' Commun. ACM, vol. 21, no. 12, pp. 1016-1025, Dec. 1978.

[WOOD79] M.R. Woodward, M.A. Hennell, and D. Hedley, 'A measure of control flow complexity in program text,' IEEE Transactions on Software Engineering, vol. 5, no. 1, pp. 45-50, January 1979.

[YAU80] S.S. Yau and J.S. Collofello, 'Some stability measure for software maintenance,' IEEE Trans. Software Engg., vol. SE-6, pp. 545-556, Nov. 1980.

[YOU85] S.S. Yau and J.S. Collofello, 'On design stability measure for software maintenance,' IEEE Trans. Software Engg., vol. SE-11, pp. 849-856, Sept. 1985.

[HFLI87] H.F. LI and W.K. Cheung, 'An empirical study of Software matrics', IEEE Trans. Software Engg., Vol-13, June 87.