Applying Metrics to Rule-Based Systems

A Thesis by: Paul Doyle, B.Sc.
Supervisor: Mr. Renaat Verbruggen

Submitted to
Dublin City University
Computer Applications
for the degree of
Master of Science
July 1992

Acknowledgements

To my parents who provided me with more support and encouragement than they will ever know, and to Renaat whose advice and guidance was invaluable

Contents

Applying Metrics to Rule-Based Systems

Abstract

Author Paul Doyle

Since the introduction of software measurement theory in the early seventies it has been accepted that in order to control software it must first be measured. Unambiguous and reproducible measurements are considered to be the most useful in controlling software productivity, costs and quality, and diverse sets of measurements are required to cover all aspects of software

A set of measures for a rule-based language RULER is proposed using a process which helps identify components within software that are not currently measurable, and encourages the maximum re-use of existing software measures. The initial set of measures proposed is based on a set of basic primitive counts. These measures can then be performed with the aid of a specially built prototype static analyser R-DAT Analysis of obtained results is performed to help provide tentative acceptable ranges for these measures

It is important to ensure that measurement is performed for all newly emerging development methods, both procedural and non-procedural As software engineering continues to generate more diverse methods of system development, it is important to continually update our methods of measurement and control This thesis demonstrates the practicality of defining and implementing new measures for rule-based systems

1 Introduction

This chapter introduces the fundamental theory behind the idea of measurement, in particular how it relates to software engineering Section 1 1 explains some of the general aspects of measurement and section 1 2 relates measurement to modern day software development environments Section 1 3 provides an overview of the areas covered by this thesis

1.1 Measurement theory

There are many interpretations of the importance and usefulness of measurement in everyday life It is often surprising however, that a great deal of confusion still exists because of these varied interpretations Before we can really appreciate the theory of measurement we should first clarify what in fact measurement is The following formal definition provides us with a good base from which to work

> *Measurement* is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules

Now we can see that measurement is involved with capturing information about attributes of entities We can define an *entity* as an object, for example a room or a person, or an activity, such as a journey or the test phase of a software project An *attribute* is a feature of these entities which we are interested in This could be the weight of the person, the colour of the room or even the length of time to walk through the room Further discussion relating attributes and entities to a software engineering framework is provided in chapter 2

Using this procedure we can identify typical ambiguity associated with measurement It would be incorrect for example, to state that we measure 'entities' or that we measure 'attributes', rather we measure 'attributes of entities' It would be ambiguous to say that we 'measure a man' since we could measure the weight, height or complexion Similarly we cannot say that we 'measure the weight', since we measure the weight at a specific time under certain conditions (altitude, dressed or undressed, before or after a meal etc)

Returning to our definition we find that measurement assigns numbers or symbols to attributes of entities to describe them These numbers or symbols can be any designated scale (for example height could be measure in Metric or Imperial scales) and therefore measurement of the same attribute may return many values depending on the scale used This is an important point to remember when interpreting results There is a tendency to believe that a number obtained from a measure is a precise representation of the attribute Measurement however is not that clear cut There are many different views on what is and is not measurement, and we can turn to Fenton for a good discussion of the science of measurement [Fenton 91]

1.2 Measurement in software

Software engineering is the term used to describe the collection of techniques concerned with applying an engineering approach to the construction of software products This 'engineering' approach involves managing, costing, planning, modelling, analysing, designing, implementing, testing and maintaining software systems These activities along with supportive tools and techniques are used in an attempt to produce high quality systems that are delivered on time and within a specified budget

Although this approach was proposed nearly two decades ago, the improvements in software products has not matched the initial, perhaps over optimistic hopes for this method Fenton attributes this failure to the less than rigorous approach taken for measurement within this discipline and provides the following observations

1 Producers of software still fail to set measurable targets when developing products Claims are often made regarding the 'user-friendliness', 'reliability' and 'maintainability' of the products without specifying what these mean in measurable terms Producers have only vague notions of their objectives which implies that they cannot fully achieve them Gilb summed up this situation with the following statement

   "projects without clear goals will not achieve their goals clearly" [Gilb 87]

2. We fail to measure the components which make up the real costs of software projects For example we usually do not know how much time was really spent on design compared to testing

3 We do not attempt to quantify the quality of the products we produce Thus we cannot tell a potential user how reliable a product will be in terms of its likelihood of failure in a given period or use, or how much time will be required to port the product to a different machine

Fenton goes on to stress the need for measurement but claims that it is misused and little understood

> *"measurement performed is done infrequently, inconsistently and incompletely Moreover it is quite detached from the normal scientific view of measurement"* [Fenton 91]

Measurement while cited by DeMarco as the key to controlling software production, is not well enough understood to be as effective as once hoped When measurement takes place the motivation for it is not always clear and it could be said that most measurements are done purely to please the quality controller and that the results are unscientific and unreliable However, the aim of software measurement is to control or predict the state of software development This can be done by focusing measures to achieve project specific results. An example would be collecting data in order to monitor and modify development The earlier in the product life-cycle this is done the more control there is on quality in terms of functionality, reliability, cost and scheduling

1.2.1 Software metrics

Now that we have established that software engineering requires measurement it is important to see how this has been done so far First however, we should clarify at this point the difference between a metric and a measure

- *Metrics* numerically characterise simple attributes like length, number of decisions, number of operators (for programs), number of bugs found, cost and time (for processes)

- *Measures* are 'functions' of metrics which can be used to assess or predict more complex attributes like cost or quality

However, since there is an inevitable confusion relating to the phrase 'metric' due to its many different interpretations, an attempt will be made to ensure that the term will not be used, wherever possible

The term *software metrics* however is an all-embracing term given to a wide range of apparently diverse activities These include

- Cost of effort estimation models and measurements
- Productivity measures and models
- Quality control and assurance
- Data collection
- Quality models and measures
- Performance evaluation and models
- Algorithmic complexity
- Structural and complexity metrics

These activities are listed in an order that represents a progression from the topics that are concerned with high level goals down to the foundational material on which these may depend Most of these will be expanded in the following three chapters

1.3 Measurement for rule-based systems

Most of the measurement activities mentioned in section 1 2 2 were developed for third generation languages However, there is a need to ensure that measurement is performed for all software systems This may require the development of new measures for new software development methodologies

The aim of this thesis is to take a langauge which incorporates methods of development for which no measures exist and to define those measures under a standard framework which is presented in chapter two To ensure that measures which already exist are not 'reinvented', chapters three and four provide a summary of the more common and frequently implemented measures Chapter five goes on to describe a process through which measures may now be defined, and chapter six suggests a new set of measures for a rule-based language This is followed by an initial analysis of the results obtained using a prototype static analyser to collect data

1.4 Summary

Software metrics are not well defined methods for assessing software quality Producers of software should have well defined goals before measurement techniques relevant to these goals can be identified Only then can results be obtained from measurement that have meaning One of the primary concerns with software engineering is not the fact that measures are not being used but that when they are, they are not focused on a particular goal and the results are unscientific and not applicable to the real world

The use of measurement within software to provide information on quality, cost and development schedules needs to be extended to encompass all methods of software development, and not just the more traditional third generation languages The following three chapters provide the required background for software measurement as it is currently used in todays development environments, while chapters 5,6 and 7 discuss the use of newly proposed measures for rule-based systems

2 A framework for measurement

2.1 Software entities and attributes

The use of software metrics is often perceived to be quite straight forward. Measurements are performed on specified parts of the system and conclusions are drawn from these results This however, while true in one sense, is a naive over simplified approach taken by many involved in quality assessment A typical set of tasks required to successfully take advantage of these techniques are provided below.

- Set up a framework for the metrics
- Have a clear understanding of the aims of measurement
- Decide how this measurement will take place

The success of the measurement process will be based on how strictly these steps are adhered to Too often measurement theory is employed without a clearly stated target or aim In these cases it is difficult to make claims regarding any aspect of the software Such haphazard systems would not provide sound analytical results with which a high degree of confidence could be associated The collection of evidence should be a clearly defined process which is specified before measurement begins

The first step relating to the framework is a matter of deciding between various existing methods for incorporating measurements into a software assessment scheme Before this can be done however, we must first identify the components within software which we may wish to measure It is widely agreed that there are three distinct entities whose attributes we are interested in measuring.

●	Processes	- software related activities (usually with time factors)
●	Products	- deliverables such as documents and source code
●	Resources	- items that are input to processes

Within these entities we can identify internal and external attributes which are items of interest upon which we would like to perform either measurement or prediction

- Internal attributes refer to measurement of the given entities in terms of themselves

- External attributes refer to how given entities relate to their environment

Typically, managers and users tend to be more interested in the external attributes of entities, such as the cost-effectiveness of a production, or the degree of reliability or useability of the software It has been observed however, that these attributes are traditionally the most difficult to measure, mainly due to the lack of quantifiable definition with which to assess them Subjectivity plays a major part in resisting the establishment of standards for such attributes. For example, the useability of a system is as yet still assessed by the 'feel' of the interface. Efforts are being made to standardise interfaces, but as yet these do not cover all forms of software applications

Current attempts to measure external or high level attributes have been based on the identification of more primitive or lower level sub-attributes which are called internal attributes For example, we could take an external attribute such as maintainability, and attempt to ascertain its value by dividing it into source code simplicity and consistency Relevant measures could then be devised to quantify these two internal attributes In general it can be stated that measuring external attributes is based on the measurement of related internal attributes Evidence obtained from internal attributes is used to support external ones because we cannot measure external attributes directly

2.1.1 Processes

Processes are software related activities which normally have a time factor An example of a typical process in the system development life cycle would be the construction of specification and design documentation, integrated testing or the development of the entire software system from the initial specification stage through to the installation stage A process may be time dependent and not activity dependent This could relate to a specific time period of the project development, for example, *Month of September*

External attributes associated with this entity are related to general notions of quality, stability, controllability, observability and cost-effectiveness. The main problem associated with these attributes is that they are not very well understood and they tend to be very subjective, for example, *the controllability of testing procedures.* Therefore it is difficult to define objective measures It is hoped that from experience obtained using these subjective measurements more objectivity will be developed.

Although there is a large degree of subjectivity associated with external attributes, objective internal attributes have been identified These directly measurable attributes are

- Time - duration of the process
- Effort - associated with the process
- Incidents - the number of incidents of a particular type arising during a process eg *no of bugs found during testing*

Given the fact that there are very few directly measurable internal attributes it is still possible to combine them to form indirect measures We must keep in mind however, an understanding of what is captured by the indirect measure For example, in a process for formal testing, the average cost of identifying errors during processing *AC* could use the indirect measure below where *Cost* relates to the cost incurred in performing the formal testing and *Number of errors* is the number of errors detected in the software as a direct results of this testing process

$$AC = \frac{Cost}{Number\ of\ errors\ found} \qquad \textbf{(Equation 2.1)}$$

In this example the external attribute of cost has been related to a specific item and then quantified using the internal attribute associated with frequency of defined incidents Similarly other expressions of cost may also be quantified

2.1.2 Products

Products within software are usually seen as deliverables of the software development life cycle These deliverables are physical entities which are typically documents and code resulting from software development These could be specification and design documents, source code, user and installation manuals and testing specification documentation at various levels of detail External attributes associated with these products are numerous Recently the draft standard ISO9126 which is a list of proposed product quality characteristics has been approved [ISO9126] This now ensures that a common set of external attributes may now be identified for product analysis A set of proposed internal attributes are also supplied but they are not part of the approved standard It is however a significant step towards standard measurement techniques for product attributes Below are the ISO9126 standardised external product attributes

Functionality

"A set of attributes that bear on the existence of a set of functions and their specified properties The functions are those that satisfy stated or implied needs "

Reliability

"A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time."

Useability·

"A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users."

Efficiency

"A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions "

Maintainability

"A set of attributes that bear on the effort required to make specific modifications "

Portability

"A set of attributes that bear on the ability of software to be transferred from one environment to another "

These external attributes are related to both documentation and source code As already stated the internal attributes associated with these are not standardised and there are many suggested ways of measurement At present an informative appendix to ISO9126 exists containing proposed internal attributes which are given below

Functionality		
	Suitability	Appropriateness of a set of functions to specified tasks.
	Accuracy	Production of agreed results or effects
	Interoperability.	Ability to interact with specified systems.
	Compliance	Adherence to specified standards or conventions.
	Security	Prevention of unauthorised access to programs and data.
Reliability		
	Maturity	Frequency of failure by faults in the software
	Fault tolerance	Maintenance of specified levels of performance.
	Recoverability	Re-establishment of performance after software faults.
Useability		
	Understandability	Users effort for recognising logical concepts.
	Learnability	Users effort for learning its application.
	Operability:	Effort for operation and operation control
Efficiency:		
	Time behaviour	Throughput rate in performance of functions.
	Resource behaviour	Resource amount and duration required
Maintainability		
	Analysability	Effort required to identify deficiencies or failure.
	Changeability	Effort required for fault removal or modification.
	Stability	Risk of unexpected effect of modification
	Testability	Effort required for validating the modified software
Portability		
	Adaptability	Opportunity for adaptation to different environments.
	Installability	Effort to install s/w onto specified environments.
	Conformance	Adherence to standards relating to portability

2.1.3 Resources

These are considered to be the inputs of software production Examples are personnel (individual or teams), materials, tools (software and hardware), and methods The cost of employing these inputs is one of our primary interests, it has a high degree of relevance to all resources and it is easily measured (sometimes the cost is directly related to the number of attributes)

With personnel we must introduce productivity as well as cost We can only realistically consider the productivity of a programmer with respect to some activity such as the time taken to code, test, and design a program or the volume of output such as the number of lines of code written over a period of time, or the number of pages in the specification and design documentation An example would be the definition of software productivity *P* as being the *amount of output* divided by the *effort input*, where the output is measured in lines of code and the input is the effort in man months

$$P = \frac{\underline{\textit{Amount of output}}}{\textit{effort input}} \qquad \textbf{(Equation 2.2)}$$

This formula is derived from the product (amount) and the process (effort) This, however, is not a true representation of the productivity of software since *Lines Of Code* does not have a direct relation to productivity It should be noted that although many measurements are feasible, the underlying principles behind their use should first be assessed

Other attributes of interest to personnel are age, experience and intelligence With teams they are size, structure, and experience of team leader Measurement of these attributes are often based on empirical evidence and is not easily quantifiable

2.2 Prediction and assessment

Measurement within software is typically a process involving the assessment of an attribute of an entity However a prediction is often required regarding attributes of entities which do not yet exist For example, at the end of the product life cycle it is possible to accurately determine the cost of development We would like to predict this cost at the early stages in product development to accurately budget resources Similar predictions are required to determine development schedules and effort

Fenton made the distinction between prediction and assessment by first providing the following definition

"A model is an abstract representation of an object" [Fenton 91]

This implies that there are many different types of models, but there are primarily two main models of interest within software measurement

- Models which are abstract representations of process, products, and resources These are used to define unambiguous measures

- Models which are abstract representations of the relationships between attributes of entities These relate two or more measures in a formula

If we take the second model and look at an example, we can eliminate some of the confusion relating to the difference between prediction and assessment Below is a simple model

$$E = \frac{l}{a} \qquad \textbf{(Equation 2.3)}$$

where l is the number of lines of code in a software product, E is the effort required to produce the product and a is a constant. The extent to which this model is used for assessment or prediction depends on how much we know about the parameters of the model If the parameter l is known then we are assessing the value of E and not predicting it However, if the project is still in the specification stages and we are estimating the number of required lines of code (maybe based on the functionality of the system) then we are using this model to predict the effort required for software development In this case we would need a prediction (or estimation) procedure to determine the value of l We can see from this that a model is merely a formula and on its own is insufficient for performing prediction Methods to determine the model parameters and procedures to interpret the results are also required The following definition formalises these concepts

"A prediction system consists of a mathematical model together with a set of prediction procedures for determining unknown parameters, and interpreting results." [Littlewood 88]

Using the same model different results may be obtained depending on the prediction procedures used Much confusion exists in software regarding assessment and prediction, but since the ultimate goal is prediction even assessment metrics are claimed to be part of prediction systems

2.3 Measurement frameworks

Now that standard methods of measurement have been introduced they should be set into a comprehensive framework which incorporates the ideas introduced in this chapter We consider which processes, products and resources are relevant to each method, which attributes (internal and external) we are measuring, and whether they are performing assessment or prediction We will cover briefly all of the topics introduced earlier but our main emphasis will be on product measurement

2.3.1 Cost and effort estimation

This is concerned with the prediction of cost and effort process attributes. The most well known model is undoubtably Boehm's COCOMO (COnstructive COst MOdel) [Boehm 81] model for estimation of size and effort of software products

Within COCOMO there are three models, basic, intermediate, and detailed, each of which can be used at different stages of development These models were derived from data obtained from applications written in Fortran, PL 1 and COBOL. The following formula calculates effort estimation in person months

$$Effort = aS^{b} * Product\ of\ Cost\ Drivers \qquad \textbf{(Equation 2.4)}$$

S is measured in thousands of delivered source instructions (usually LOC) a and b are determined by the mode of development There are three modes: organic - small to medium DP projects, embedded - ambitious but tightly constrained, and semidetached -somewhere between the previous two

COCOMO's 15 cost drivers fall into four categories Product attributes, personnel attributes, computer attributes, and project attributes The model provides default values for cost driver attributes, but these should be modified as more historical data becomes available relating to actual cost COCOMO provides estimates for development effort and schedule divided into three phases; product design, programming and integration testing

The COCOMO estimation formula is usually done at the specification stage of software development The formula above is as explained earlier, only a model of software development, and the prediction procedures for determining a,b, and S along with the model make up the prediction system It is hard therefore to talk about the COCOMO model for cost estimation since different prediction procedures will yield different results Also this is not a very satisfactory system since the prediction of the attribute for lines of code could be as difficult as the prediction of the original attribute of cost (or effort)

2.3.2 Productivity measures and models

Here we are measuring the resource attribute *Personnel* (teams or individuals) during a process (usually a calendar time period) The most common models are those that take productivity as a function of the output of the personnel during a process, divided by the input (cost) of the personnel during that process as explained in section 2 1 3

The resource attribute of productivity is assumed to be captured as an indirect measure of a product attribute measure and a process attribute measure

2.3.3 Quality models and measures

Quality modelling [McCall 77] involves relating metrics, internal attributes and external attributes to some theoretical framework It is used to associate external product attributes (sometimes referred to as factors) to internal attributes (known as the criteria) which in turn are evaluated by using proposed sets of measures.

It is generally agreed that the use of software engineering methods leads to construction of products with certain structural properties These properties are characterised by internal attributes such as those proposed under ISO9126 Some may even state that the verification of the correct implementation of these methods will ensure 'satisfactory' levels of external attributes expected by software users, for example, reliability, maintainability, and useability Thus the assumption that *good* internal structure leads to *good* external quality is part of most software quality models

We must however realise that software engineering only provides the <u>framework</u> from which improvements in software are possible by encouraging the use of improved techniques, ie projects in which the best common practice techniques are applied routinely are deemed more likely to have a satisfactory end product than those developed ad-hoc However none of these methods can guarantee the level of external attributes since so much depends on how these methods are applied to individual problems Also there is no standard scaling system for determining the 'level' of external attributes

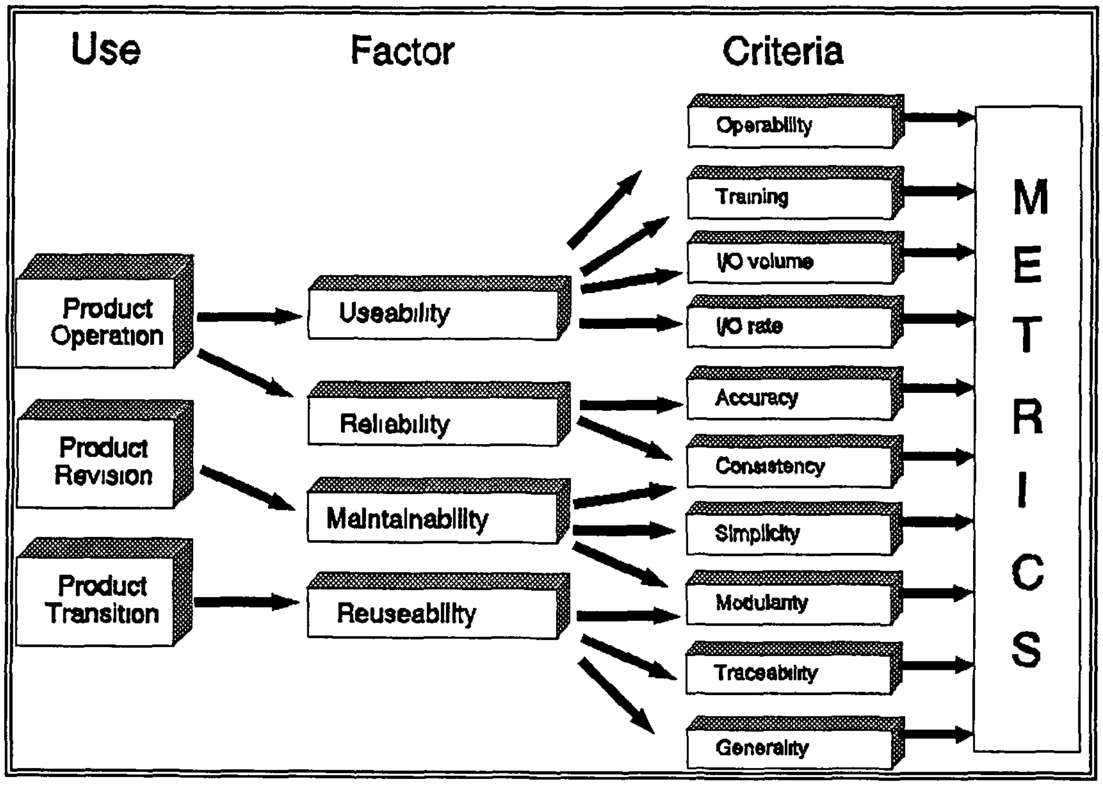


Figure 2-1 A typical Quality Model

Although there is an 'intuitive feel' regarding the connection between the internal structure of software products and external product attributes, there is very little scientific evidence to establish specific relationships This is perhaps the result of difficulties in setting up relevant experiments and a lack of understanding of how to measure important internal product attributes properly

Defining models of quality aids in the development of a structured process through which attributes of software may be measured, recorded, and re-used in future projects By providing reliable data, based on historical and measured values, prediction and assessment techniques may be used to control productivity, cost and quality Measurable targets may be set within software projects which will increase confidence in the producer's claims to specified external attributes Without these attributes being made quantifiable little weight can be associated with claims of a product's level of quality

2.3.4 Data collection

Models for cost/effort estimation, productivity assessment and prediction depend on the accurate measurement of process and resource attributes. Much of the work in data collection for all but product attributes must be done by hand so effort is aimed at setting in place rigorous procedures for gathering accurate measures of the process and resource attributes Three main techniques for gathering data are as follows

Software analysers - data is generated automatically as programs are submitted to a compiler, specified measures are performed at this time. Dynamic analysis is performed when measures are performed will the program is executing Software analysers are by nature algorithmic and objective

Report forms - are logs which are completed by analysts and programmers at various milestones in the product development process Entries could include date, time, activities, effort in hours. Forms can often cause confusion regarding the data required.

Interviews - To avoid misunderstanding of the data required while completing forms-based-questionnaires, interviews are often conducted The same information is obtained using forms.

Most forms of data collection, with the exception of software analysers suffer from one major problem they are not reproducible Elements of subjectivity introduce uncertainties which affect the degree of confidence associated with the data.

2.3.5 Performance evaluation

Usually measures in this area deal with the product attribute *efficiency* This can be defined in terms of time (response times and processing times) and storage (amount of resources used and the duration of such use) Efficiency is mainly seen as an external attribute for executable code, however it can be an internal attribute which looks at the algorithmic complexity of a program and identifies repetition of source code etc

In the case of time efficiency assessment is performed by determining key inputs and basic machine operations, and then working out the number of basic operations required as a function of input size In terms of our framework, one possible approach is to determine the efficiency of the algorithm as an internal attribute which can be used to predict the external attribute *efficiency of executable code*.

2.3.6 Structural and complexity metrics

Most external product attributes are high level and very difficult to measure, so we are often forced to consider measures of internal attributes of products Within complexity there are two different issues to be addressed

Computational complexity

Concerned with the efficiency of the algorithm in its use of machine resources

Psychological complexity

Concerned with characteristics of the software that affect programmers performance in composing, comprehending and modifying software

Curtis encompasses these notions into one definition

"Complexity is a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software " [Curtis 88]

Complexity is a function of the software and its interaction with other systems (machines, people, other software) To devise a complexity model we must combine specific metrics according to some theory or hypothesis Types of complexity models which exist are as follows

- Problem complexity
- Design complexity
- Program / Product complexity

The measurements defined should represent the difficulty that a programmer/analyst encounters when performing tasks such as designing, coding, or maintaining software There are numerous measures which are concerned with measuring internal structural and complexity attributes which will be detailed in chapter 3

2.3.7 The GQM paradigm

It was initially stated that for measurement to be successful we must first have objectives in mind Once those objectives have been established we should use the framework described in this chapter to identify relevant attributes and entities to be measured. This goal oriented approach is consistent with the Goal/Question/Metric paradigm of Basili and Rombach [Basili et al 88] which is a well known framework for a wide spectrum of software measurement The idea is that a goal must be identified before measurement begins, this should lead to questions, and these question can be answered with the use of measurement The goals are normally defined in terms of purpose, perspective, and environment and to help define goals a set of templates are provided

Template for goal definition

- Purpose to characterise / evaluate / predict / motivate etc the process / product / metric / model etc in order to understand / assess / manage / engineer / learn / improve it Example *To evaluate the maintenance process in order to improve it*

- Perspective Examine the cost / effectiveness / correctness / defects / changes / product measure etc from the viewpoint of the manager / customer / developer Example *Examine the cost from the viewpoint of the developer*

- Environment The environment consists of the following process factors, people factors, problem factors, methods, tools, etc Example: *The maintenance staff are poorly motivated programmers who have limited access to tools*

Guidelines are also provided for process and product related questions. The questions that are addressed are the definition of the process or product and relevant attributes When we come to defining the measures it is understood that in many cases more than one measure will be required for one question and these may include subjective measures

We can now see how GQM can be related to our framework A goal or question can be related to entities each having a choice between assessment or prediction (entities and attributes need to be defined first) We now are concerned with quantifying the attributes of a product, process or resource The leaves of the hierarchy tree are directly measurable attributes of entities

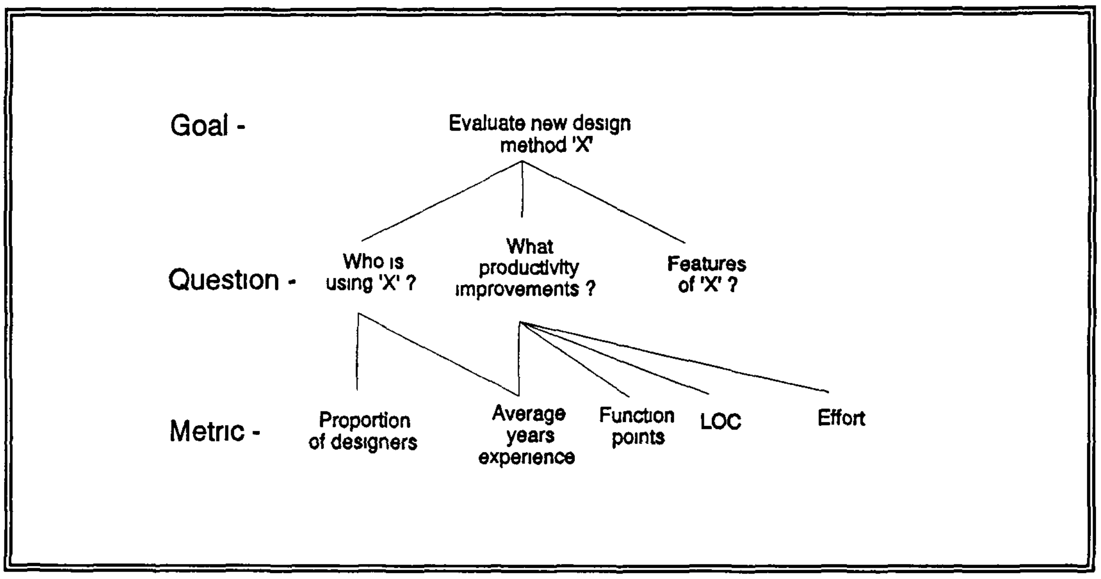


Figure 2-2 The GQM approach

2.4 Summary

To ensure that a system of measurement is implemented correctly, an understanding is required of what the aims of the measurement are Typically there are three entities, resource, process, and product which are to be evaluated. For measurement to be scientifically based, external attributes need to be identified and their corresponding internal attributes The identification of measures to quantify specific internal attributes will be affected by the type of information required, most measures perform some assessment of the software, but are usually claimed to be part of a prediction model Models of how these frameworks are defined are typically Boehm and McCall quality models or the GQM approach The following two chapters give us a review of the more common measures applicable to the product entity

3 Specification and design measures

3.1 Introduction

The prediction and assessment of software is a process which is not restricted to one stage of the software development life cycle It is important to define each of these activities as accurately as possible to identify measures that are phase dependant A common software life cycle is the *waterfall model* Such a model has been described by Boehm [Boehm 81], Figure 3-1

Requirements and Specifications - This phase should produce a complete specification of the required functions and performance characteristics of the software It should also look at resource needs and preliminary development cost estimates

Product Design - This phase should produce more detailed module specifications including their expected size, the necessary communication among modules, algorithms to be used, interface data structures, and internal control structures. It should highlight important constraints relative to timing or storage, and include a plan for testing the individual modules

Programming/Coding - This phase should produce an implementation of the modules in the chosen language together with unit testing and subsystem testing

System Integration - This phase, usually completed by a group independent of the original analysts and programmers, should subject the integrated modules to extensive testing to ensure that all functional requirements are met Errors are, of course, corrected as discovered

Installation/Acceptance - This phase should deliver the product to the users organisation for final acceptance tests within the operational environment for which it is intended Documentation manuals are delivered, training is conducted, problems recorded and corrected until the customer accepts the product

Maintenance - This is a continuing phase in which additional discovered errors are corrected, changes in code and manuals are made, new functions are added, and old functions are deleted

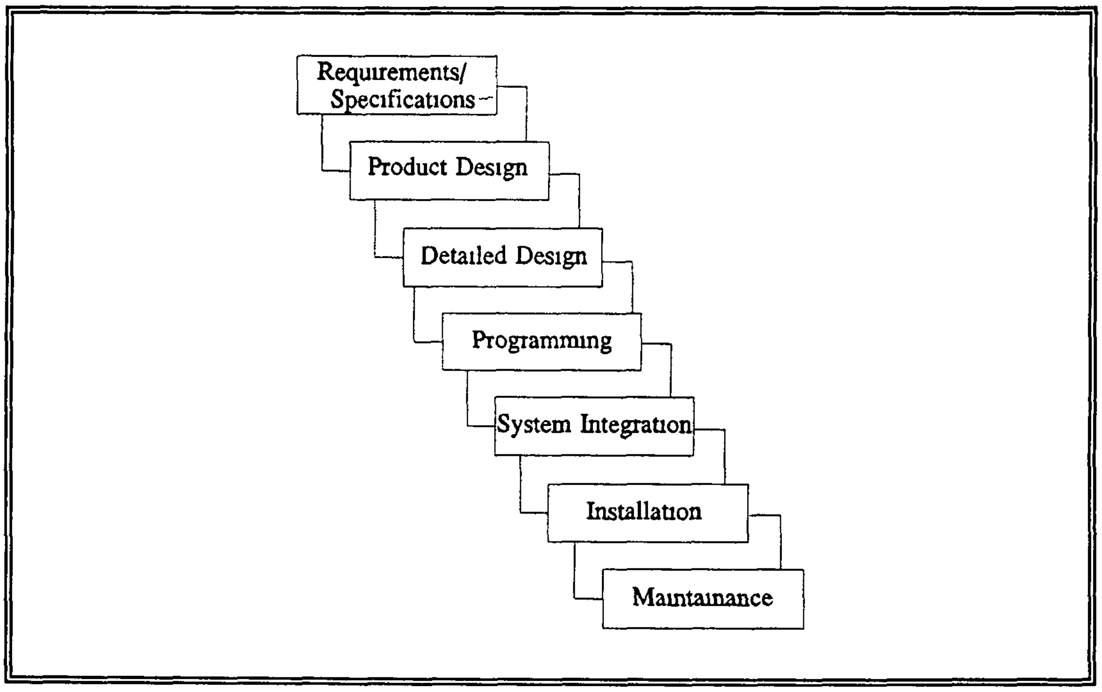


Figure 3-1 The waterfall model of the software life cycle

These stages are generally sequential but tend to be interdependent. Changes from one phase filter through to cause significant changes in others Measures are not usually defined to fit neatly into the seven stages described, however, four general categories exist which contain related activities

Design - This phase contains the Requirements/Specification, Product Design, and Detailed Design phases All preliminary work before actual coding begins

Coding - This phase is the same as the Programming/Coding phase Code is written in this phase

Testing - This phase contains the Systems Integration and Installation/ Acceptance phases. Internal testing with same or *live* test data is performed in this phase

Maintenance - This phase is the same as the Maintenance phase Software is now in operation

This chapter describes some of the more common metrics is in use under the above classifications Emphasis is placed in the later stages in product rather than process entities

3.2 Specification measures

Initial costs and time estimates are required at the earliest stages of product development Measures during the analysis stage which attempt to provide these predictions are based on the system specification Since the specification describes the requirements of the system and not the implementation method, quantitative measures of the true function to be delivered, as perceived by the user, will be provided Most research into specification metrics has been done by DeMarco

3.2.1 DeMarco's "BANG" metric (specification weights)

Bang is a function metric, an implementation-independent indication of system size The information content (size) of the specification model is a direct measure of the quantity of usable system functions to be delivered The measure is based on the decomposition of each part of the specification model down to their primitive levels such as data elements, objects and relationships The relationship between these primitives is then quantified using a weighted factor

Bang is the earliest predictor of size which is used to drive the cost model and hence is only a very rough estimator

Primitive Components of the Model

A component of the specification model is considered primitive if it is not partitioned into subordinate components Each part of the specification model (functional model, data model and state transition model) is divided down to its primitive level, see Table 3 1

Types of primitives

- Functional primitives (Functional Model)
- Data Element is the primitive data item (character, number etc). Data Elements are contained in the data dictionary component of the functional model
- The primitive component of retained data organisation is the object. An object is a group of stored data items, all of which characterise the same entity (Data Model)
- The primitive component of retained data interconnectedness is the relationship (Data Model)

Table 3.1

Partitioning vehicle	**Is used to partition**	**To produce as primitives**
Function network	System requirements	Functional primitives
Data dictionary	System data	Data elements
Object diagram	Retained data	Objects
Object diagram	Retained data	Relationships
State diagram	Control characteristics	States
State diagram	Control characteristics	Transitions

The following primitive counts provide basic measures for use with Bang.

FP	count of functional primitives lying inside the man-machine boundary
FMP	count of modified manual functional primitives (functions lying outside the man-machine boundary that must be changed to accommodate installation of the new automated system)
DE·	count of all data element
DEI	count of all input data elements
DEO	count of all output data elements
DER	count of data elements retained (stored) in automated form
OB	count of objects in the retained data model
RE	count of relationships in the retained data model
ST	count of states in the state transition model
TR	count of transitions in the in the transition model
TC_i	count of data tokens around the boundary of the ith functional primitive (evaluated for each primitive) , a token is a data item that need not be subdivided within the primitive
RE_i	count of relationships involving the ith object of the retained data model(evaluating for each object)

DeMarco identified two systems classifications with which Bang could be used.

(a) Function strong systems

(b) Data strong systems

The former relates to systems that can be thought of almost entirely in terms of the operations they perform on data The latter relates to systems that can be thought of in terms of the data they act upon, the data groupings and the interrelations rather than the operations

(a) Formulating Bang for Function-Strong Systems

The principle component of Bang for function-strong systems is FP However some functions cost more to implement than others Variations exist in both size and complexity which must be adjusted for in the model

(i) Correcting for Variations in Function Size

Correcting for size is based on the observation that the function model has reduced the system to a series of linked primitive *transformations* Output tokens are generated from input tokens in each transformation The information content or size of a transformation can be approximated as a function of TC_i the number of tokens in the transformation Studies [Halstead 77] into how size varies with TC leads to the following relationship

$$\mathit{Size}\ (\mathit{Primitive})\ \alpha\ TC_i * \log_2 (TC_i) \qquad \textbf{(Equation 3.1)}$$

Table 3 2 provides weighted values based on this formula

Table 3.2

Data Weighting for Size Correction of Functional Primitives

TC_i	Corrected FP Increment (CFPI)
2	1.0
3	2.4
4	4.0
5	5.8
6	7.8
7	9.8
8	12.0
9	14.3
10	16.6

The corrected FP (CFP) is now

$$CFP = \sum CFPI_i \qquad \textbf{(Equation 3.2)}$$

(ii) Correcting for Variations in Complexity

DeMarco reasoned that the complexity of primitives do not vary greatly and when they do they have a discernible pattern. Sixteen well defined categories were identified and a correction factor for each was given.

- *Separation* - primitives that divide incoming data items
- *Amalgamation* - primitives that combine incoming data
- *Data direction* - primitives that steer data according to a control variable
- *Simple update* - primitives that update one or more items of stored data
- *Storage management* - primitives that analyse stored data, and act based on the state of that data
- *Edit* - primitives that evaluate the net input data at the man-machine boundary

- *Verification* - primitives that check for and report internal inconsistency
- *Text manipulation* - primitives that deal with text strings
- *Synchronization* - primitives that decide when to act
- *Output generation* - primitives that format net output data flows
- *Display* - primitives that construct two-dimensional outputs (graphs, pictures)
- *Tabular analysis* - primitives that do formatting and simple tabular reporting
- *Arithmetic* - primitives that do simple mathematics
- *Initiation* - primitives that establish starting values of stored data
- *Computation* - primitives that do complex mathematics
- *Device management* - primitives that control devices adjacent to the computer boundary

Table 3 3 contains a suggested set of correction factors for these categories

Table 3.3

Complexity Weighting Factors by Class of Function

Class	Weight	Class	Weight
Separation	0 6	Synchronization	1 5
Amalgamation	0 6	Output Generation	1 0
Data Direction	0 3	Display	1 8
Simple Update	0 5	Tabular Analysis	1 0
Storage Management	1 0	Arithmetic	0 7
Edit	0 8	Initiation	1 0
Verification	1 0	Computation	2 0
Text Manipulation	1 0	Device Management	2 5

Complexity weighting factors are environment dependent Relative complexity of arithmetic and formatting functions for example would be different in C and Cobol The weighting factors given need to be altered to suit the project's development environment

(b) Formulating Bang for Data-Strong Systems

This variation of Bang relates to systems that have a significant database and most of the effort is allocatable to tasks having to do with implementing the database itself The most obvious primitive count to base measurement analysis upon is ***OB***, the count of objects in the database Adjustments are required to account for the different costs of implementing different objects Table 3 4 provides weights for each object as a function of its relatedness to other objects

Table 3.4

Relation Weighting of Objects

RE_i	**Corrected OB Increment (COBI)**
1	1 0
2	2 3
3	4 0
4	5 8
5	7 8
6	9 8

Bang is the sum of the COBI over all objects

$$Bang = \frac{\sum COBI}{OB} \qquad \textbf{(Equation 3.3)}$$

Prediction using Bang

This measure is a quantitative indicator of the net useable function from the user's point of view It can be used early in the life cycle to predict effort and can be used as in cost models to predict development costs Below are two algorithms provided by DeMarco for the computation of Bang The first is for function-strong systems and the second is for data-strong systems

ALGORITHM 1: Computation of Bang for Function-Strong System

```
Set initial value of FUNCTIONBANG to zero
For each functional primitive in the function model

 Compute Token Count around the boundary
   For each incoming our outgoing data flow

      1  Determine how many separate tokens of data are visible within the primitive
         This is not always the same as the count of data elements  If a group of data
         elements can be moved from input to output without looking inside, it
         constitutes only a singe token

      2  Write Token Count at the point where the data flow meets the primitive

   Set Token Count = sum of tokens noted around the boundary

 Use Token Count to enter Table 3 1 and record CFPI from the table
 Allocate primitives to a Class
 Access Table 3 2 by Class and note the associated Weight
 Multiply CFPI by the accessed Weight
 Add Weighted CFPI to FUNCTIONBANG
```

ALGORITHM 2: Computation of Bang for Data-Strong Systems

```
Set initial value of DATABANG to zero.
For each object in the retained data model:
    Compute count of relationships involving that object. Use the relationship
    count to access Table 3.3 and record COBI accessed.
    Add COBI to DATABANG.
```

It is worth remembering that measures derived from the specification model are only as good as the model itself. If it specifies something other than the system required, the metrics will also be astray. Also it the requirements change and the model is not revised then the measures will be out of data and useless.

3.2.2 Function points

Function points [Albrecht 79] are implementation independent measures useable in the early stages of the software life cycle. As with Bang, function points use the requirements specification and are a weighted sum of counts of user visible product features. The aim of function points is to provide a measure of size which can be used to drive cost models such as COCOMO. The following points should be remembered before using function points.

- They cannot be derived without a full software system specification, a user requirements document is *not* sufficient.

- Differences of 400 to 2000% in the number of function points counted at the start and finish of system development are not uncommon. This can be due to the introduction of non-specified functionality, or the fact that the level of detail in the specification is coarser than that of the implementation.

Therefore the number and complexity of inputs, outputs, and enquiries will be underestimated. Using function points for prediction may not always be useful.

- Elements of subjectivity are required for function point counting which eliminates the possibility of automating the process. Detailed counting rules are required to ensure sufficient levels of consistency.

- The counting rules need tailoring to the specific analysis methods used. [Ratcliffe et al 90].

- Function points have been successful in DP applications but their use in real time and scientific applications is controversial.

- Within function points there is an adjustment based on the technological complexity of the product. This involves assessing the impact of 14 factors on a six-point ordinal scale. This introduces more subjectivity.

Computing the Value FP

The first step in obtaining a value for FP is to first compute the *unadjusted function count* UFC. The number of `items' of the following types must be counted:

External inputs -	Those from the user which provide distinct application-oriented data. Examples are file names and menu selections. These do not include enquiries.
External outputs -	Those to the user which provide distinct application-oriented data. Examples are reports and messages.
External enquiries -	These are interactive inputs requiring some response.
External files -	These are machine readable interfaces to other systems.
Internal files -	These are logical master files in the system.

Havıng ıdentıfied the varıous types of ıtems, each ıs gıven a subjectıve 'complexıty' ratıng of eıther *sımple, average* or *complex* Weıghtıng factors for each are gıven ın Table 3 5

Table 3.5

Weighting Factors for FP ordinal scale

ITEM	Weighting Factor		
	Sımple	Average	Complex
External ınput	3	4	6
External output	4	5	7
User ınquıry	3	4	6
External fıle	7	10	15
Internal file	5	7	10

In theory there are 15 dıfferent varıetıes of ıtems (each five types multıplıed by the three levels of complexıty) so we have

$$UFC = \sum_{i=1}^{15} (No\ \ of\ items\ of\ variety\ i) * (weight_i) \quad \textbf{(Equation 3.4)}$$

The adjusted functıon poınt count FP ıs derıved from UFC by multıplyıng ıt by a *technıcal complexıty factor* TCF

$$FP = UFC * TCF \qquad \textbf{(Equation 3.5)}$$

Factors Contributing to Complexity (TCF)

F1	Relıable back-up and recovery	F2	Data communıcatıons
F3	Dıstrıbuted functıons	F4	Performance

F5·	Heavily used configuration	F6	On-line data entry
F7	Operational ease	F8	On-line update
F9	Complexity interface	F10	Complex processing
F11.	Reusability	F12	Installation ease
F13:	Multiple sites	F14	Facilitate change

Each factor is rated on a 'scale' 0,1,2,3,4,5, where 0 means it is irrelevant and 5 means it is essential Then TCF is

$$TC = 0\,65 + 0\,1\sum_{i=1}^{14} F_i \qquad \textbf{(Equation 3.6)}$$

TCF varies from 0 65 if all $F_i = 0$ to 1 35 if all $F_i = 5$

One fault with function point is that it includes subjective notions of complexity, both internal and external If these measures could be performed separately then it might be possible to develop a measure that provides a measure of true functionality

3.3 Design models

A design is a model of a particular way of meeting the system requirements. A design should be a formal representation of the software to be implemented and it should be thought of as a rigorous blueprint for construction It must be recorded and kept up to date throughout the duration of the project

"Design is the determination of what modules & what intermodular interfaces shall be implemented to fulfil the specified requirements " [DeMarco 88]

Program and system designs are based around decomposition down to the module level The product of the decomposition into modules can be seen as the *design module* The completed design model consists of a partitioning of the whole into its modules and a census of all interfaces between these modules The design model is complemented by a set of internal model specifications describing their contents

The following points indicate some important relationships between design and implementation of design

- One to one relationship between modules indicated in the design & modules implemented in the code

- One to one relationship between intermodular connections indicated in the design and intermodular references (CALLS etc)

- One to one relationship between intermodular data interfaces indicated in the design and intermodular shared data implemented in the code

To summarise, designs should describe all data interfaces between modules Many project teams fail to complete the design stage by ignoring inter-module interfaces Such deficiencies deprive development teams of important feedback regarding the validity of the partitioning and hence the design DeMarco outlined the following test to see if design was correctly performed (Figure 3-2)

<u>The Did-We-Really-Do-Design Test</u>

1 *Put the design into a sealed envelope*
2 *Give the completed software to an outside expert, someone who is not familiar with the original design*
3 *Ask your expert to derive the design implied by the implementation*
4 *Compare the derived design with the design in the envelope*
5 *If the two are not identical, you didn't really do design*

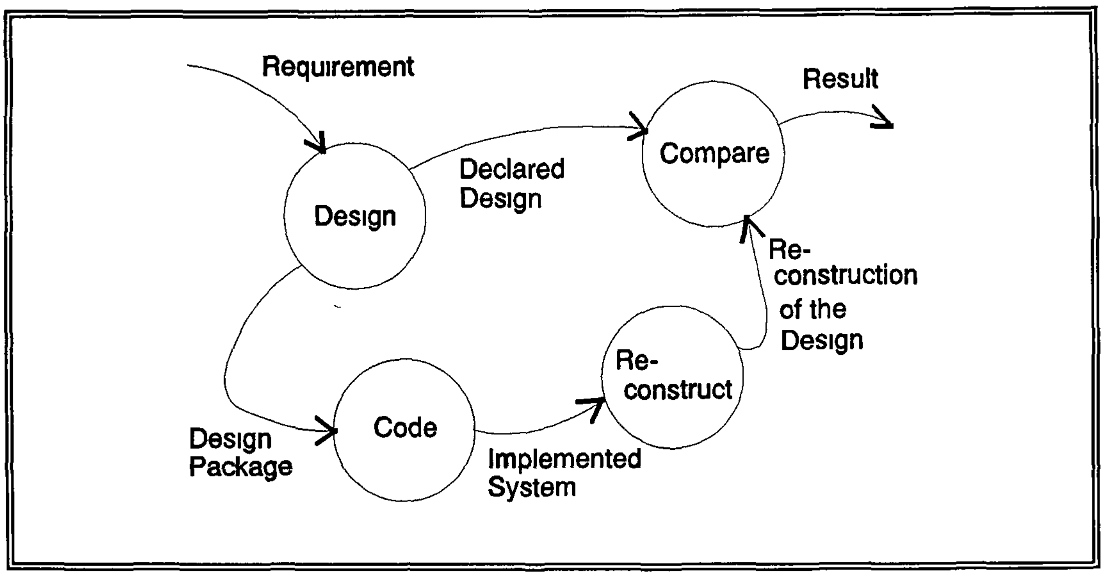


Figure 3-2 The Did-We-Really-Do-Design Test

A common technique for design modelling is to represent a design as a hierarchy of modules (Figure 3-3)

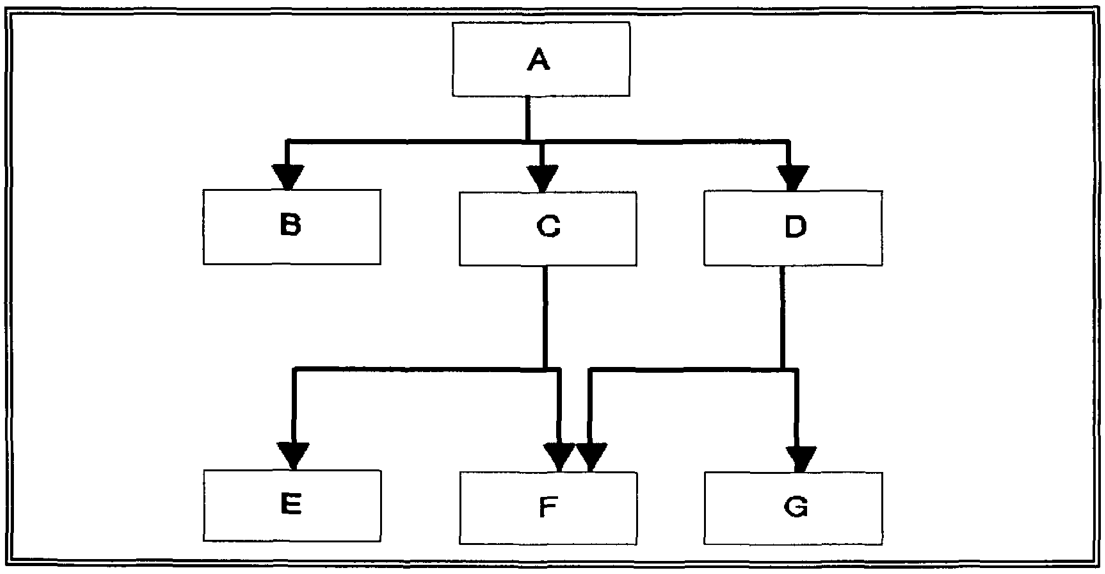


Figure 3-3 Hierarchy of modules

The significance of such a hierarchy is that the manager starts the managed model by passing control to it. The managed model does its work and then returns control to its manager One module in the hierarchy is working at any given time. Traditionally control is passed up and down, never sideways

Invocation is the act of passing control to a subordinate module All the lines connecting modules on the hierarchy represent invocations and all are shown The hierarchy is a statement of all the possible invocations in the system If two modules share data or control parameters then they are said to be coupled This information is shown along side the lines of invocation on the hierarchy

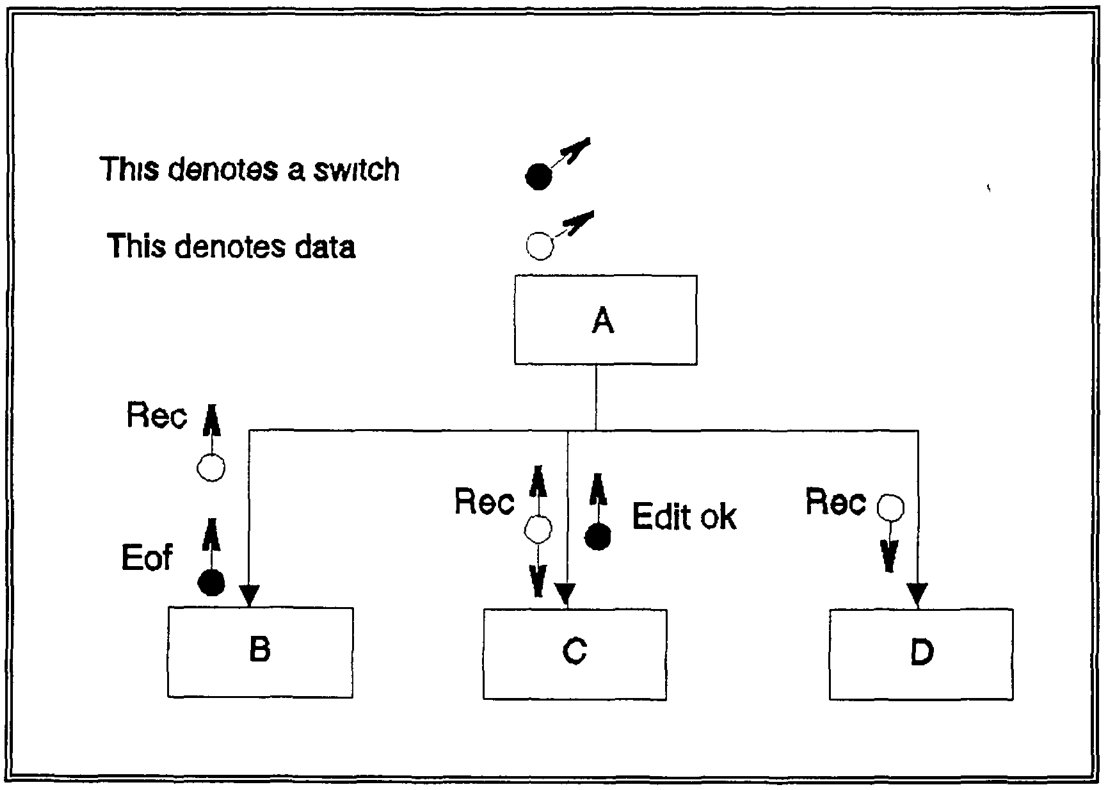


Figure 3-4 Invocation conventions in module hierarchies

To develop meaningful design measures we must ask the following question *When is the partitioning into modules complete?* DeMarco proposed the following rule:

<u>**Rule:**</u> *The design partitioning is complete when the modules are small enough to be implemented without any further partitioning A simple test of adherence is that no implemented module shall need an internally named procedure*

So far nothing has been suggested about *how* you ought to design systems or what is a good or bad design, although much work has been done by Yourdon in this area [Yourdon et al 79] This section provides a understanding of what a design model is so that the following section relating to specific measures makes sense.

3.4 Design measures

As projects approach conclusion the prediction of the cost, size and effort of the system should be converging to their actual values For this to happen more accurate projections are required as the development proceeds as shown in Figure 3-5 For example, cost predictions made during the design phase must improve on those made during the analysis phase Measures of product cost based on the design must be more reliable and precise Initial estimates of function are based on Bang which are implementation independent and not related to *how* the specification is implemented

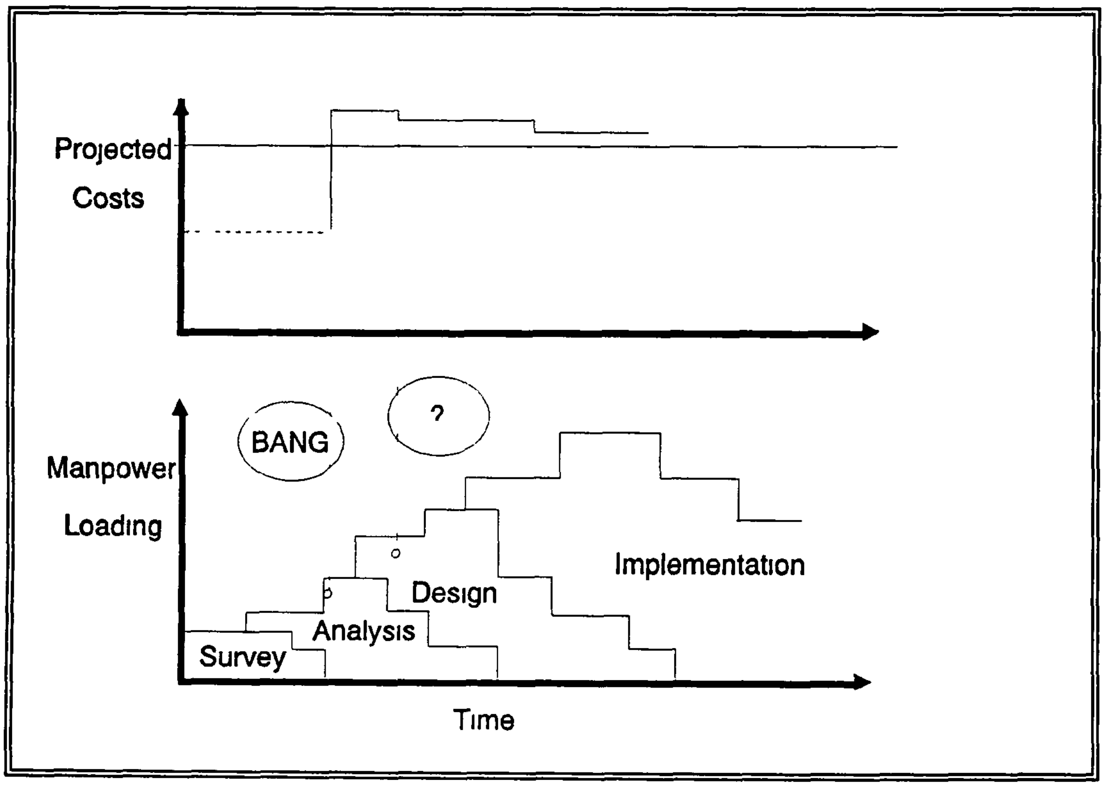


Figure 3-5 Improving cost prediction over time

3.4.1 Design weight

To improve on Bang we need to incorporate a measure which includes the effort implied in the design This implementation-dependent predictor is usually called the *Design Weight* The following steps are required to produce this predictor.

1 Calculate primitive metrics (derived from the design model)
2 Calculate composite predictor (design weight) using a weighted formulation
3 Collect data from a range of projects
4 Produce a prediction line equation
5 Projection of new development costs from the prediction line equation and the observed value of the predictor

Primitive Design Measures

The design model should contain a hierarchy of modules with all connections and couples indicated, and a design data dictionary describing all data items (couples, tables, files, database(s), and structured data types) The following are basic measures observable from such a model

MO Count of modules

CO· Count of intermodular normal connections (a normal connection is a reference from inside one module to another *whole* module, that is, a CALL or PERFORM or other subroutine invocations

DA_i Count of data tokens explicitly shared along normal connections to and from module i (Evaluated for each module)

SW_i Count of control tokens (switches) shared along normal connections to and from module i (Evaluated for each module)

EN	Count of *encapsulated data groups* in the design model (an encapsulated data group is a data area made available to a limited number of modules)
EW_i	Count of encapsulation width of data group i (width is defined as the number of modules with access to the group)
ED_i	Count of encapsulation depth of data group i (depth is defined as the number of data elements contained in the group)
PA	Count of pathological connections (a pathological connection is a reference from inside one module to part of another module, that is, a GOTO to an internal label)
PD_i	Count of pathological data tokens shared by module i (a pathological data token is one that is obtained from a module not connected to module i by any normal connection)
PS_i	Count of pathological control tokens shared by module i

Usually, if it is impossible to provide counts for the primitives listed then the developers have probably gone about the design in such a way that DA, SW, PD, and PS are not apparent, ie they have concentrated on the control structure rather than on the data sharing It is important to examine the volume and complexity of the interfaces

Design weight collects these primitive metrics together to use as a predictor for remaining implementation effort, coding and testing Initial efforts to use MO, the count of modules, as a predictor of effort proved quite disappointing as modules of similar size require different efforts, usually relating to how complex they are. It is generally thought that effort required in module design, coding and testing varies with the number of decisions in the module

Predicting Decision Count Inside Modules

A good estimate of the decision count within a module is to assume that the internal structure of each module is isomorphic to the data structure at its boundaries [Warnier 76] Figure 3-6 gives the full set of isomorphisms between data structure and process structure

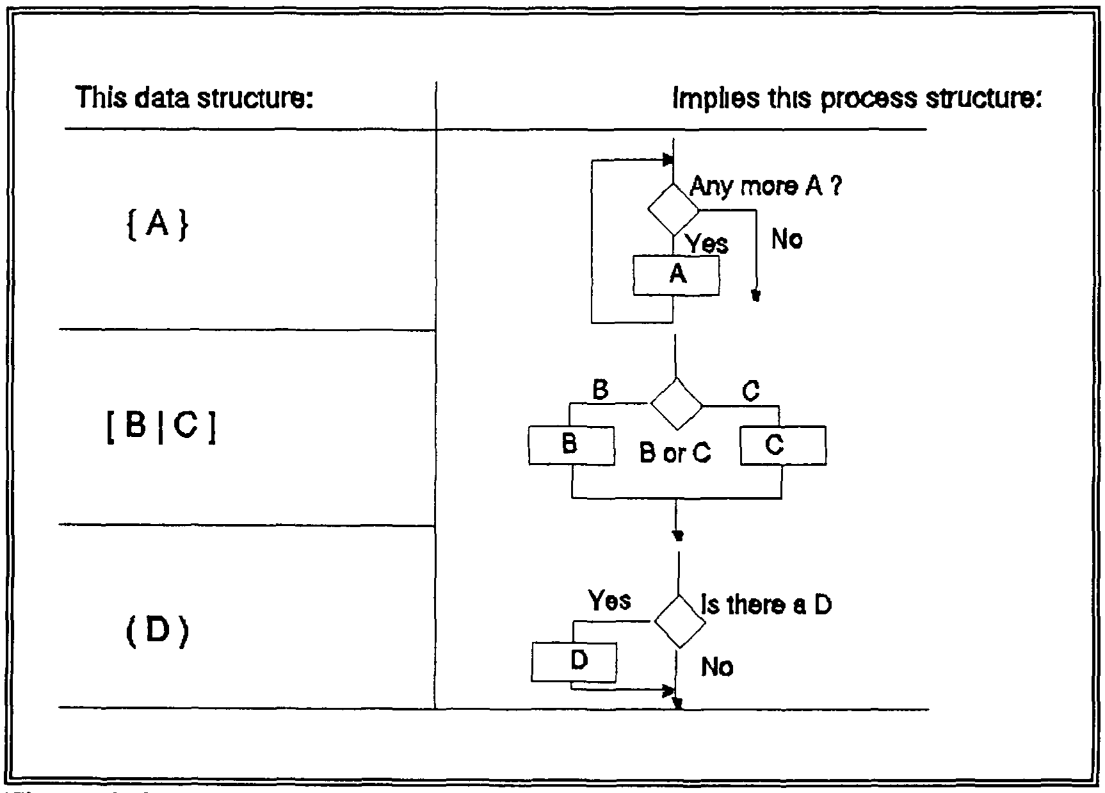


Figure 3-6 Data and Process parallels

The procedure for predicting decision counts from the data structure observed at the modular boundary is

Start with the decision count = 0

1 *Write down a data dictionary formulation of all data arriving at the module boundary Express the result at the token level (from the viewpoint of the module)*

2. *Analyse the data structure of the result, applying the following rules*

(a) *For each iteration in the data structure, add one to the decision count*

(b) *For each two-way selection (Either-Or) in the data structure, add one to the decision count*

(c) *For each n-way selection in the data structure, add n - 1 to the decision count*

(d) *For each option (data item that may or may not be present) in the data structure, add one to the decision count*

Table 3.6

Module Weights

Decision Count		0	1	2	3	4	5	6
Token Count	1	1 0	1 1	1 2	1 4			
	2	2.4	2 6	2 9	3 3	3 7		
	3	4.0	4 4	4 9	5 4	6 2	7 2	
	4	5 8	6 3	7 1	7 9	9 0	10 5	12 5
	5	7 8	8 5	9 5	10 7	12 2	14 1	16 8
	6	9 8	10 7	12 0	13 4	15 3	17 8	21 2
	7	12 0	13 0	14 6	16 4	18 7	21 8	26 0
	8	14 3	15 6	17 4	19 6	22 3	26 0	31 0

Design weight is simply the sum of the Module weights over the set of all modules in the design

$$Design\ Weight = \sum Module\ Weight_i \qquad \textbf{(Equation 3.7)}$$

The cost predictor (weight) of each module is now a function of the token count at its boundary, and the predicted decision count inside Table 3 6 contains a suggested set of initial values

3.5 Summary

The system life cycle development model has been used as a partition for metrics, which allows us to identify stage dependent measures This waterfall model enable the identification of four general categories, Design, Coding, Testing and Maintenance So far little research has been performed to provide design measures, which, quite often is seen as a relatively minor stage, however such measures (design and specification metrics) allow early predictions for software cost models Most notably DeMarco's Specification and Design weight measures are among the few that currently exist As the coding phase begins however, a wider range and variety of measures can be implemented, some of which will be discussed in the following chapter

4 Product attribute measures

4.1 Introduction

In chapter 3 the waterfall model of the system life cycle was introduced. Specification and design measures associated with the early stages of this life cycle were described and indicated to be an essential part of the prediction of software product size Predictions of software size are then used to drive cost models such as COCOMO to predict effort and cost of product development This chapter will concentrate on measures associated with the later stages of the software life cycle

Of the three entities defined relating to software products in chapter 2, (Product, Process, and Resource) Product is the most relevant to this research. The ISO9126 quality attributes outlined previously are based on interesting external product attributes and associated internal attributes. This chapter focuses in on well known and implemented internal attributes

4.2 Internal product attributes

Internal product attributes are attributes of software (including documentation) which are dependant on the product itself This section is intended to look in detail at those attributes, and suggested methods of measurement Specifically those related to textual, structural and architectural components of software

Internal attributes are considered to be the key to improving software quality which is one of the main aims of software engineering Internal attributes may be used for quality control and assessment and are the building blocks for measuring complexity

It is generally agreed that the use of software engineering methods leads to the construction of products with certain structural properties These properties are characterised by internal attributes such as those proposed in ISO9126 which were discussed in section 2 1 2 There is a wide consensus among software engineers that these internal structural attributes will help ensure increased quality in the external attributes expected by software users Thus the assumption that 'good' internal structure leads to 'good' external quality is fundamental to most software quality models We can conclude.

'Axiom' of software engineering
Good internal structure ---> Good external quality

We must however, realise that software engineering only provides the <u>framework</u> for producing 'good' software by encouraging the use of structured techniques, ie. projects in which the best common practice techniques are applied routinely are 'more likely' to have a satisfactory end product than those developed 'ad-hoc' However, none of these methods can guarantee the <u>level</u> of quality of external attributes since so much depends on the how these methods are applied to individual problems

Although there is an intuitive feeling regarding the connection between the internal structure of software products and external product attributes, there is very little scientific evidence to establish specific relationships This is perhaps caused by the difficulties in setting up relevant experiments and perhaps a lack of understanding of how to measure important internal product attributes properly

4.3 Textual measurement

Initial coherent software measurement research was performed by Maurice Halstead. Halstead's work was based on the idea that software comprehension was related to a process of mental manipulation of program tokens. This was the first attempt at deriving software measures from a theory, and the first set of measures used in an industrial context. Most textual based measures are concerned with code size and volume. These issues are discussed below along with typical measures associated with them.

The size of a program is an important measure for three reasons. The first is that it is easy to compute after the program is completed, the second is because it is the most important factor for many models of software development, and thirdly that productivity is normally based on a size measure.

While 'size' would seem to be a rather straight forward attribute to assess we find that it becomes quite complex when notions of effort, functionality, complexity, redundancy and reuse become part of the measurement. The reason for the complexity is that size is normally used in the assessment of cost, productivity, and effort. The problem seems to be defining a set of fundamental attributes which will cover the notion of size in software. There appear to be three such attributes of software: length, complexity and functionality. The state-of-the-art for size measurement is that a) there is some consensus view on measuring length of programs but not specifications or designs, b) there is some work on measuring functionality of specifications (which applies equally to designs and programs), but c) there is little work on measuring problem complexity other than what has been done under computational complexity.

4.3.1 Lines of code

The Number of Lines of Code (NLOC) is the most used measure of source code program length However, there is a real need for a standardised definition of '*a line of code*' For example, do we include commented lines and variable declarations, and what happens if a line contains more than one instruction? To avoid such ambiguity, Conte provided the following definition

> *"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line This specifically includes all lines containing program headers, declarations and executable and non-executable statements "* [Conte et al 86]

To show that we now refer to non-commented lines of code we use the abbreviation NCLOC (or ELOC - effective lines of code) This definition however, loses some valuable length information If we wish to determine the number of pages required to print the program source code or what storage space is required for a program, then we need to know the length of the program expressed in terms of commented lines of code

CLOC is the number of lines of commented program text Using this we can define total length,

$$LOC = NCLOC + CLOC \qquad \textbf{(Equation 4.1)}$$

This way we can define indirect measure such as the density of comments in a program *(CLOC/LOC)* If we are seeking a single measure for the length of a program then LOC is preferable to NCLOC However in general it is useful to gather both measures since they are measuring different things If we continue to look for a 'pure' notion of length then we must consider the following measures which are classified as ratio measures

- Measure the length in terms of the number of bytes of storage required for the text.

- Measure the length in terms of the number of characters CHAR in the program text

The good thing about ratio scales is that in principle we can re-scale each in terms of the other For example if α is the average number of characters per line of text then we have the following re-scaling·

$$LOC = \frac{CHAR}{\alpha} \qquad \textbf{(Equation 4.2)}$$

4.3.2 Predicting length using function points

It is usually required to predict the attribute length early on in the product life cycle since it is easily understood and can be used in cost prediction models. One way of predicting length is to relate length to different life cycle products One such system involves taking the function point count obtained from the specification and applying a language dependent expansion ratio, to obtain an estimate of the lines of code required Although such a method may not be entirely accurate it does provide a reasonable estimate if the expansion ratio is based on historical product development data

Expansion ratio for language X

$$\frac{\textit{Size of product at Specification stage 1 (in FP)}}{\textit{Size of product at code stage (in LOC)}}$$

The length in terms if LOC may be estimated from the formulas equation 4 3·

$$LOC = \alpha \sum_{i=1}^{m} S_i$$ **(Equation 4.3)**

Where S_i is the size of the module i (measured in FP), m is the number of modules, and α is the function point to code expansion ratio recorded from previous projects using the same specification and code conventions This is a very general prediction system where the model parameters are estimated by the user

Attempts to establish empirical relationships between length of program code and length of program documentation [Walston et al 79] led to the following observation

$$D = 49L^{1\,01}$$ **(Equation 4.4)**

Where D is the length of documentation measured in pages and L is the length of program code measured in thousands of LOC This is only good for rule of thumb estimations More accurate results are possible when data is collected for specific environments

4.3.3 Token based measures

Halstead's method of code analysis is based on the identification of tokens within the program text Using these tokens he formalised a set of measures to determine the *Volume* of a module or program in terms of its *Length* and *Vocabulary*

Volume = Length x $\log_2$ (Vocabulary)

Length = N_1 (count of all instances of all used operators) + N_2 (count of all instances of all used operands)

Vocabulary = n_1 (count of unique operators used) +
n_2 (count of unique operands used)

The two categories of tokens identified were operators and operands. Any keyword in a program that specifies an action is considered an operator, while a symbol to represent data is considered an operand. Most punctuation marks are also categorised as operators Variables, constants and labels are operands Operators consist of arithmetic symbols (such as +, -, and /), command names (such as WHILE, OR, and READ), special symbols (such as assignments, braces, and parentheses), and even function names

The following measures formalise the most commonly used counts associated with program code, based on Halstead's token identification

1 Vocabulary size

$$n = n_1 + n_2$$ **(Equation 4.5)**

Comments not included

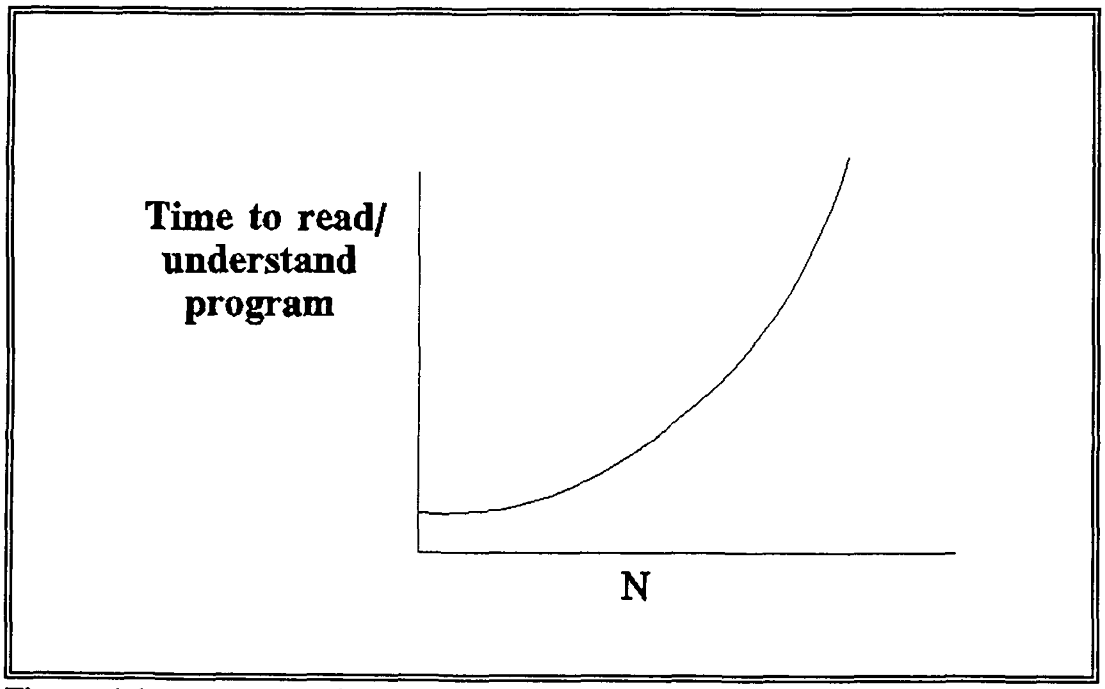


Figure 4-1 Behaviour of *N* as a measure of program legibility

2. Program length:

$$N = N_1 + N_2 \qquad \textbf{(Equation 4.6)}$$

Total use of operands and operators. This is sometimes used as an indicator of the legibility of a program. As N increases, the time required to understand the program also increases. Figure 4-1 graphs this relationship to give an indications of the relationship between program volume, as defined above, and the estimated legibility of the program.

3. Estimating length:

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2 \qquad \textbf{(Equation 4.7)}$$

4. Program volume:

$$V = N \log_2 n \qquad \textbf{(Equation 4.8)}$$

Minimum volume of bits required to encode a program with a vocabulary of n operand and operators and with length N.

5. The potential volume:

This is used to compute the algorithm's smallest possible volume. It would be necessary to use a language where all actions were defined as procedures, eg: $y = sin(x)$.

n^* = ideal vocabulary where

i) n_1^* = function name and the assignment (2)

ii) n_2^* = potential number of input/output operands

$$V^* = (2 + n_2^*) \log_2 (2 + n_2^*) \qquad \textbf{(Equation 4.9)}$$

V^* is a constant for a given algorithm independent of language.

6 Program level

$$L = \frac{V^*}{V} \quad \textit{Where} \quad 0 < L \le 1 \qquad \textbf{(Equation 4.10)}$$

The 'Distance' between the program volume V and the potential volume V^*, is sometimes called the level of abstraction When $L = 1$ we have the ideal situation, but the closer to zero that we get the greater number of operands and operators used The value of 1 may seem ideal, but we must take into consideration the legibility of a program For languages like C it becomes very hard to interpret minimalist code

7 Estimator of program level

$$\hat{L} = \frac{2}{n_1} * \frac{n_2}{N_2} \qquad \textbf{(Equation 4.11)}$$

8 The programming effort

This is related to the number of 'elementary mental discriminations' required to code the program It is derived from $V = N \log_2 n$ (the number of psychological "moments" required to code the program, and $D = 1/L$ (program difficulty)

$$E = \frac{V}{L} \qquad \textbf{(Equation 4.12)}$$

This measure is linked to the number of bugs in a program More mental effort is required the more errors there are in the code This can only be used when a lot a data is available on the real number of bugs in a program A curve may be plotted to show E verses bugs

9. The coding time

Time to code a preconceived algorithm in the language used S is the number of psychologıcal 'moments' per second and ıt has been shown [Stroud 67] that ıt ıs lınked to codıng tıme Typıcal values of S are usually low about 7 or 8

$$5 \le S \le 20 \qquad \textbf{(Equation 4.13)}$$

10 Language level.

Thıs ıs the aptıtude of a language to express an algorıthm It ıs often easıer to solve some problems ın one language rather than another If the mınımum volume V^* goes up then the program level varıes ın proportıon Consequently the constant α can be defined as follows

$$\alpha = L * V^* = L^2 * V \qquad \textbf{(Equation 4.14)}$$

Table 4.1

Values for α for different languages

Language	Mean α
Englısh	2 16
PL/1	1.53
Algol68	1 21
Fortran	1 14
Ass SDC	0 88

11 Approxımatıon of codıng tıme

$$\hat{T} = \frac{E}{S} \qquad \textbf{(Equation 4.15)}$$

The value of S can be adjusted over tıme as more data becomes avaılable regardıng actual codıng tıme

4.4 Structural measures

A substantial amount of research has been devoted to the study of measures derived from the control flow structure of a program. Control structure attributes are usually modelled by directed graphs whose nodes correspond to program statements and where the edge from one node to another indicates a flow of control between corresponding statements These directed graphs are usually called *control-flow graphs* or just *flowgraphs* An example of a program and its flowgraph are shown in Figure 4-2

```
10   INPUT P
20   DIV = 2
30   LIM = INT (sqr(P))
40   FLAG = P/DIV-INT(P/DIV)
50   IF FLAG = 0 OR DIV = LIM THEN 80
60   DIV = DIV + 1
70   GOTO 40
80   IF FLAG <> 0 OR P < 4 THEN 110
90   PRINT DIV; "smallest factor of"; P;
100  GOTO 120
110  PRINT P, "is prime"
120  END
```

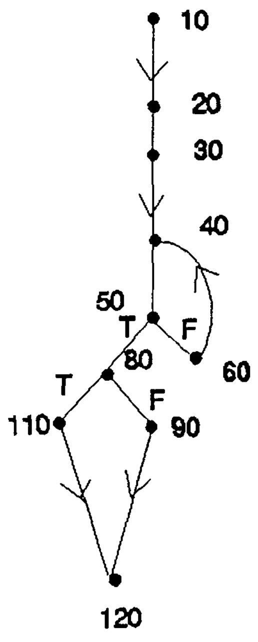


Figure 4-2 A program and its corresponding flowgraph

All programs can be structurally decomposed into primitive components These decompositions may be used to define a wide range of so-called complexity and structural measures The theory of control flow structure is formalized using graph theory Wilson provides a reference to such graph theory as needed [Wilson 72]

4.4.1 Flowgraph model of structure

A *graph* consists of a set of points (*nodes*) and line segments (*edges*). In a *directed graph* each edge is assigned a direction indicated by an arrowhead on the edge. The following is a good definition of a *flowgraph*

"A Flowgraph is a directed graph in which two nodes, the start, and the stop node, obey special properties the stop node has out-degree zero, and every node lies on some walk from the start node to the stop node "

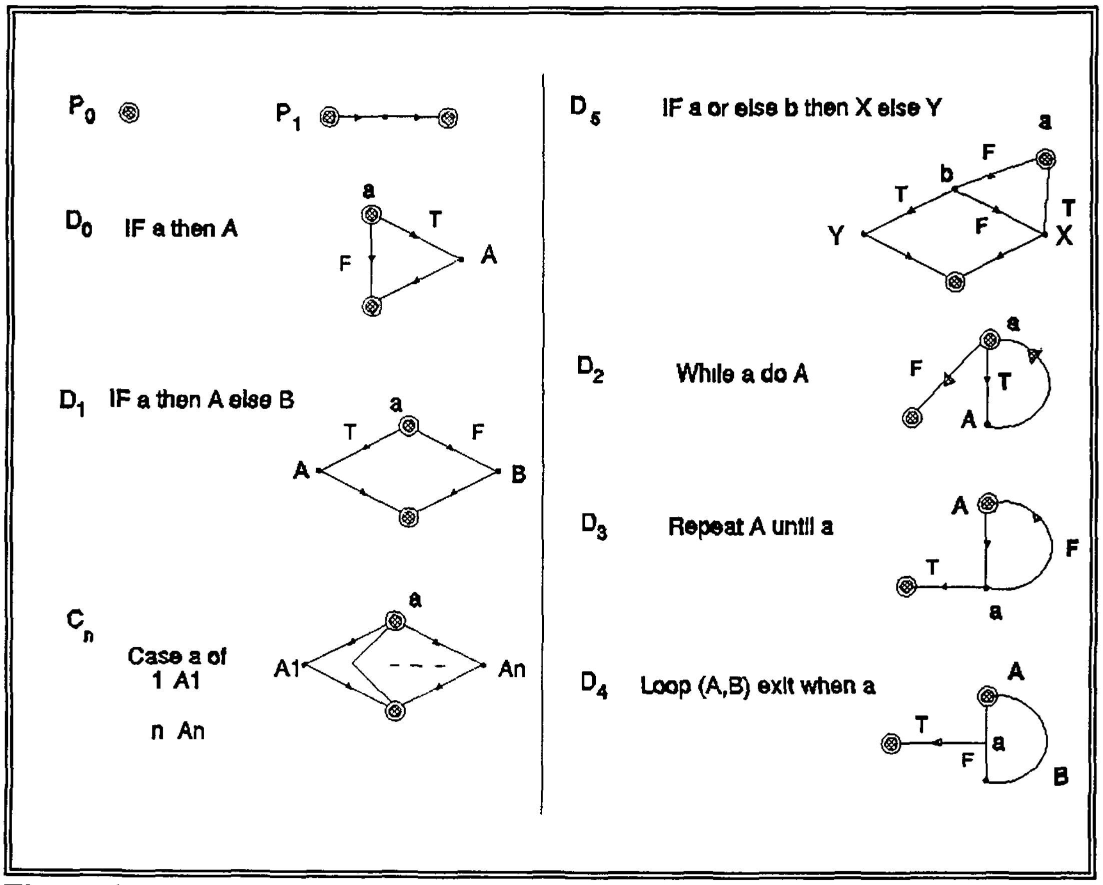


Figure 4-3 Some commonly occurring flowgraphs

Certain flowgraphs appear often enough to merit special names Figure 4-3 depicts the flowgraphs P_0, P_1, D_0, D_1, D_2, D_3, D_4, D_5, and C_n, which we will now refer to by name

4.4.2 Defining program structure

Within structured programming it is often stated that a program is structured if it is 'built up' using only a small number of allowable constructs These are normally said to be *sequence, selection,* and *iteration* as shown in Figure 4-4 [Bohm et al 66]. However, we find that in many languages we are forced to implement what are considered structured constructs by using GOTO statements For example, in Pascal GOTO's are used to implement the construct D_4 if we do not wish to duplicate code unnecessarily We require a more formal definition of program structure which can support many different views and a method for determining the level of structure in an arbitrary flowgraph

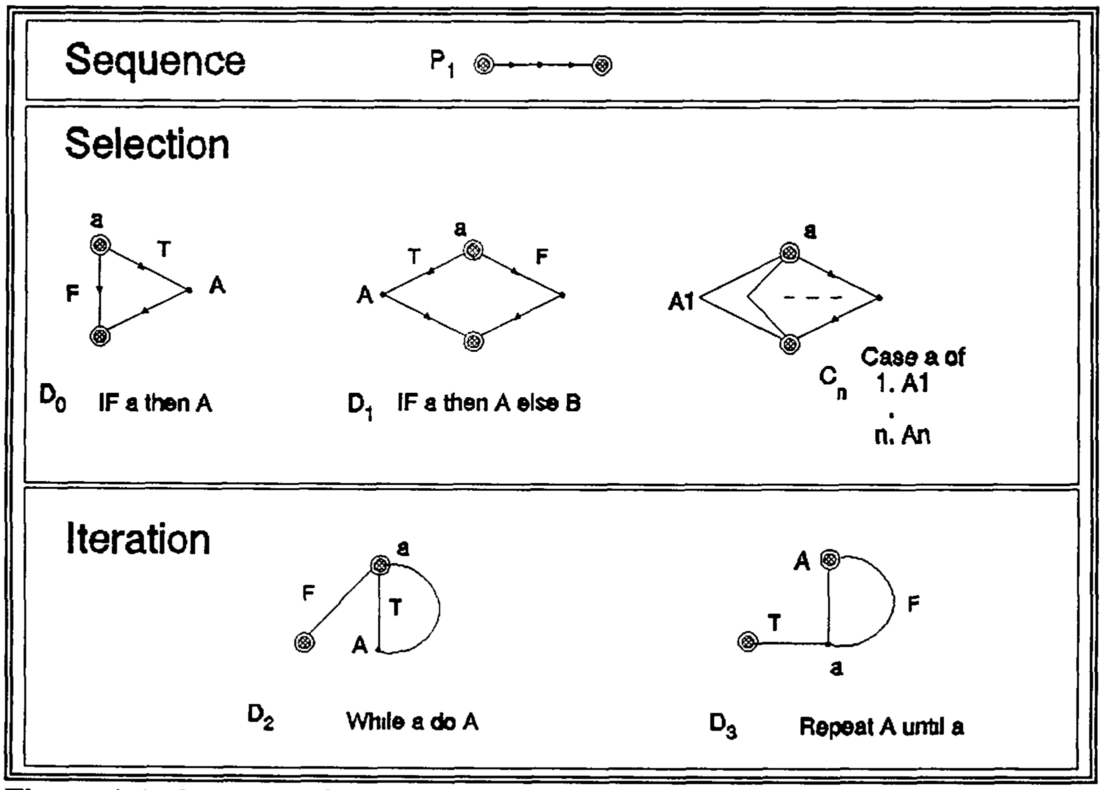


Figure 4-4 Constructs for structured programs

First we nominate a family of prime flowgraphs The set of *S-graphs* consists of the following flowgraphs

- Each member of *S* (called the basic S-graph)

- Each flowgraph which can be built recursively from the family *S* using only the operations of sequence and nesting

We can now define a set of control structures that are suited for particular applications By definition, any control structure composed of this nominated set will be 'structured' in terms of this local standard. ie will be *S-structured*

If we let $S^D = \{ P_1, D_0, D_2 \}$, then the class of S^D-graphs is the class of flowgraphs commonly know as *D-structured* graphs Bohm's results assert that every algorithm can be encoded as an S^D-graph Although S^D is sufficient, it is normally extended to include the structures D_1 (if-then-else) and D_3 (repeat-until)

4.4.3 Decomposing flowgraphs

Associated with all flowgraphs are decomposition trees which describe how the flowgraph is built by sequencing and nesting primes An example of a flowgraph *F* and its decomposition tree is shown in Figure 4-5 Fenton provides the following theorem and also provides a method for determining the unique decomposition tree of a flowgraph [Fenton et al 86]

Prime decomposition theorem Every flowgraph has a unique decomposition into a hierarchy of primes

Flowgraph construction and decomposition is normally generated automatically by most static analysis tools, eg QUALMS [Wilson et al 88]

It is easy to determine if an arbitrary flowgraph is S-structured for some family of primes *S*, by computing the decomposition tree and seeing if any of the nodes are not a member of *S* or P_n If this is true then the flowgraph is not an S-graph We can see that the decomposition theorem shows that every program has a quantifiable degree of structure, characterised by its decomposition tree.

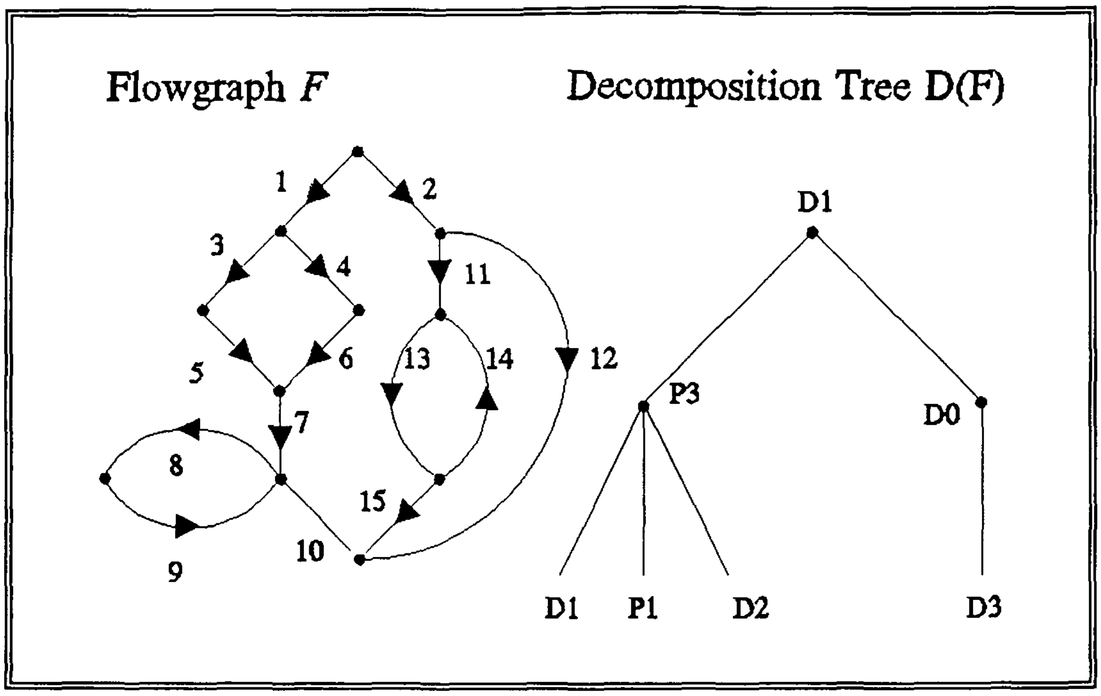


Figure 4-5 Flowgraph *F* and its decomposition tree

4.4.4 Flowgraph based measures

A large number of interesting measures may be defined based on the decomposition tree These measures are usually defined in terms of their effect on primes and the operations on sequencing and nesting

(i) Depth of nesting

To formulate a measure for the depth of nesting within an object α (such as a program modelled by a flowgraph *F*), it is required to observe α in terms of its effect on primes, sequences and nesting

Primes: The depth of nesting of the prime P_1 is zero, and the depth of nesting of any other prime F is equal to one Thus, $\alpha(P_1)=0$ and if F is a prime $\neq P_1$ then $\alpha(F) = 1$

Sequence: The depth of nesting of the sequence F_1, , F_n is precisely the maxımum of the depth of nestıng of F_is. Thus

$$\alpha(F_1; \quad , F_n) = \max (\alpha(F_1), \quad , \alpha(F_n))$$

Nesting: The depth of nestıng of the flowgraph $F(F_1, \quad , F_n)$ ıs equal to the maxımum of the depth of nestıng of the F_is plus one because of the extra nestıng level ın F Thus.

$$\alpha(F (F_1, \quad , F_n)) = 1 + \max (\alpha(F_1), \quad , \alpha(F_n))$$

To see how we use thıs to calculate the value α for a flowgraph, consıder the flowgraph *F* ın Fıgure 4-3

$$F = D_1 ((D_1, P_1, D_2), D_0(D_3))$$

Thus we compute

$$\begin{aligned}\alpha(F) &= 1 + \max (\alpha(D_1, P_1, D_2), \alpha(D_0(D_3))) \\ &= 1 + \max (\max (\alpha(D_1), \alpha(P_1), \alpha(D_2)), 1 + \alpha(D_3)) \\ &= 1 + \max (\max (\max (1, 0, 1) ,2) \\ &= 1 + \max (1, 2) \\ &= 3\end{aligned}$$

Fenton went on to define the propertıes of hıerarchıcal measures as stated ın Table 4 2

Table 4.2

If the followıng characterıstıcs unıquely determıne m for any S-graph F	
M1	m(F) for each F element of S,
M2	The sequencıng functıon(s),
M3	The nestıng functıons h_f for each F element of S
Then we say that m ıs *hıerarchıcal*	

The hierarchial measures may be automatically generated for a program once we know M1, M2, M3 and the decomposition tree

(ii) Length measure

Defining a length measure 'v' which provides a formal measure corresponding to the number of statements in a program where the latter is modelled by a flowgraph

M1: $v(P_1) = 1$, and for each prime $F \neq P_1$, $v(F) = p + 1$ where p is the number of procedure nodes in F

M2: $v(F_1, \quad ,F_n) = \sum_{i=1}^{n} v(F_i)$

M3. $v(F (F_1, \quad ,F_n)) = 1 + \sum_{i=1}^{n} v(F_i)$ *for each prime* $\neq P_1$

Using our earlier example in Figure 4-2 we find that

$$\begin{aligned} v(f) &= 1 + v(D_1, P_1, D_2) + v(D_0(D_3)) \\ &= 1 + (v(D_1) + v(P_1) + v(D_2)) + (1 + v(D_3)) \\ &= 1 + (2 + 1 + 1) + 1 + 1 \\ &= 7 \end{aligned}$$

Once a hierarchial measure has been characterised in terms of the conditions M1, M2, M3 then we have the minimum information needed to calculate the measure for all S-graphs

(iii) Simple hierarchical measures

Number of node measure 'n'

M1 $n(F)$ = # nodes in F for each prime F.

M2 $n(F_1, \quad , ,F_n) = \sum_{i=1}^{n} (n(F_i) - n + 1)$

M3 $n(F(F_1, \quad , F_n)) = n(F) + \sum_{i=1}^{n} (n(F_i) - 2n)$

Number of edges measure 'e'

This is a ratio measure of one particular view of size

M1 $e(F)$ = #edges of F for each prime F

M2 $e(F_1, \quad , F_n) = \sum_{i=1}^{n} e(F_i)$

M3 $e(F(F_1, \quad , F_n)) = e(F) + \sum_{i=1}^{n} (e(F_i) - $

Number of occurrences of names primes measure 'p'

M1 $p(F) = 1$ if F is prime to be counted, else 0

M2 $p(F1, \quad , F_n) = \sum_{i=1}^{n} p(F_i)$

M3 $p(F(F_1, \quad , F_n)) = p(F) + \sum_{i=1}^{n} p(F_i)$

Is D-structured measure 'd'

This measure yields the value 1 if the flowgraph is 'D-structured' and 0 if its not This is a nominal measure

M1 $d(F) = 1$ for P_1, D_0, D_1, D_2, D_3, D_4, D_4 and otherwise 0

M2 $d(F_1, \quad , F_n) = \min\{d(F_1), \quad , d(F_n)\}$

M3 $d(F(F, \quad , F_n)) = \min\{d(F), d(F_1), \quad , d(F_n)\}$

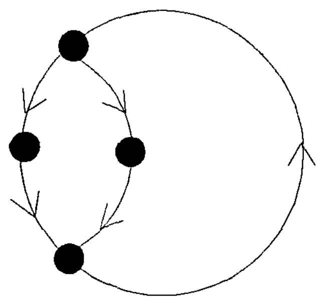
(iv) Structural complexity

McCabe's cyclomatic complexity number *v* measures the number of linear independent paths in a strongly connected program flowgraph *F* [McCabe 76] (for consistency *F* will be continued to be used even though this measures is historically known as *v(G)*). It is formalised as

$$v(F) = e - n + 2 \qquad \textbf{(Equation 4.16)}$$

for a flowgraph *F* with *e* arcs and *n* nodes

McCabes cyclomatic number



$v(F) = e - n + 2$

$v(F) = 5 - 4 + 2 = 3$

Figure 4-6 Calculating *v(F)* in a simple flowgraph

As the value of *v(F)* increases it is implied that more paths require testing, ie a lot of arcs compared to nodes In Figure 4-7 we find a relationship between the number of basic paths and *v(F)* We need to find a point where every small increase in *v*(F) results in a large increase in the number of paths to test This upper limit was set at 10 by McCabe on the basis of empirical evidence

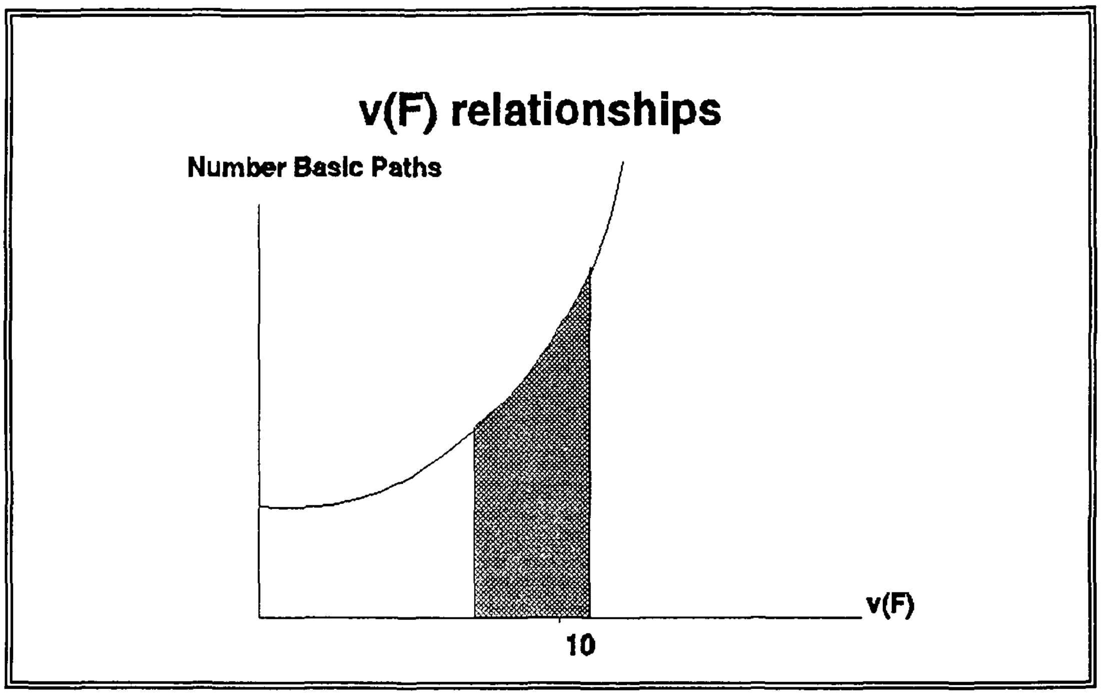


Figure 4-7 Graphing the relationship between *v(F)* and basic paths

v(F) measures the number of *basic paths* in a component We can say that it represents the minimum number of tests required to cover the graph (it should be noted that this does not mean 100% test coverage)

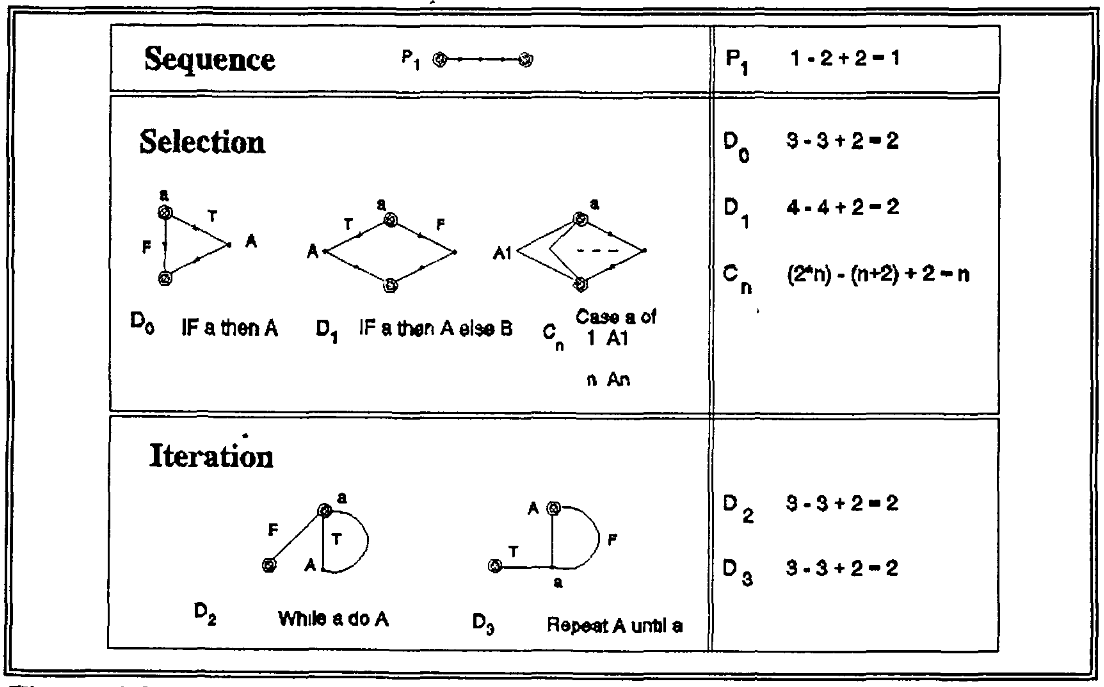


Figure 4-9 *v(F)* of usual constructs using v(*F*) = e - n + 2

Figure 4-8 shows the number of basic paths in a component This shows the number of uniquely different paths possible Paths which can be derived from others are not considered basic, and hence are not included in this count In the example provided you should note that there are only 3 basic paths, the fourth can be derived from the first three

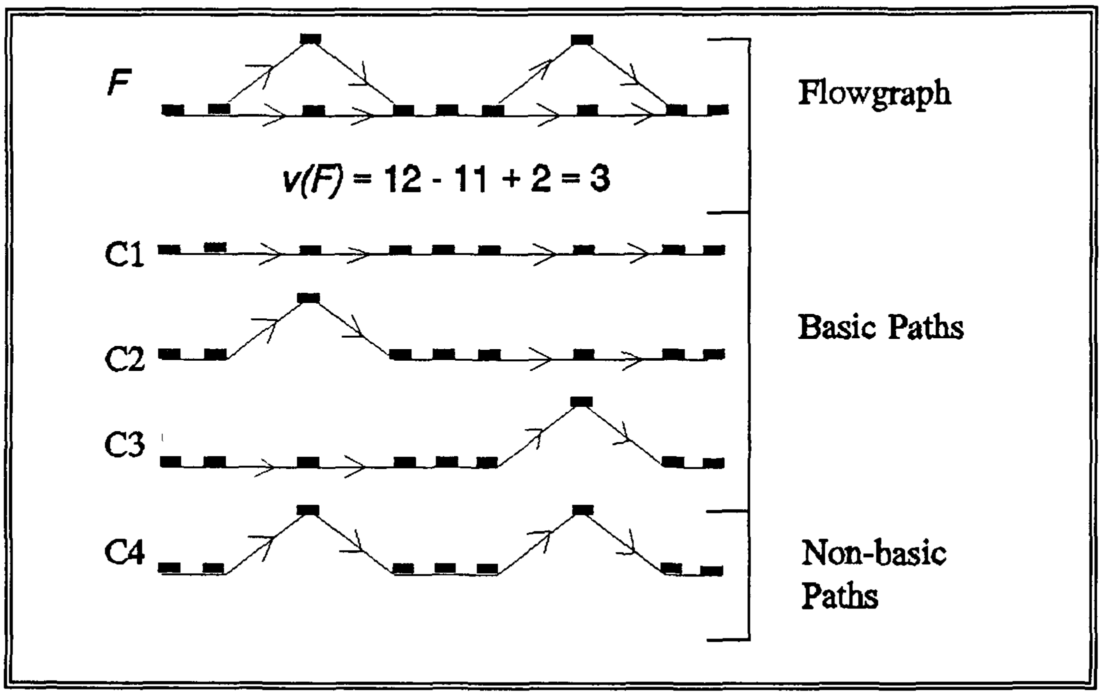


Figure 4-8 Interpreting *v(F)*, counting basic paths

Since we have identified a number of basic constructs within programming, it is an easy task to give the values of *v(F)* for their flowgraphs (see Figure 4-9) All values are predetermined with the exception of the **CASE** statement were we need to know the number of cases provided

If a program is truly structured then we can say that *v(F)* is reducible to one. This is because we have only one input and one output In Figure 4-10 we reduce the graph in a sequence of steps that must be performed sequentially and we are left with one input to a process *c* from which there is one output Figure 4-11 demonstrates the case of unstructured programming where we have either more than one input or more than one output In this case we cannot reduce the graph.

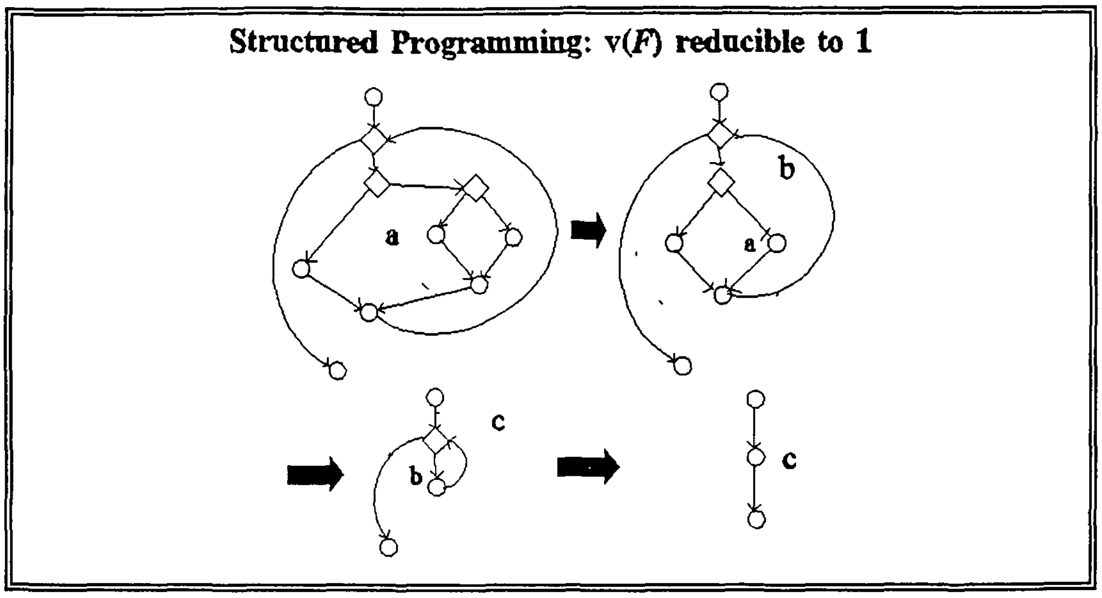


Figure 4-10 Characterising *v(F)* for structured programming

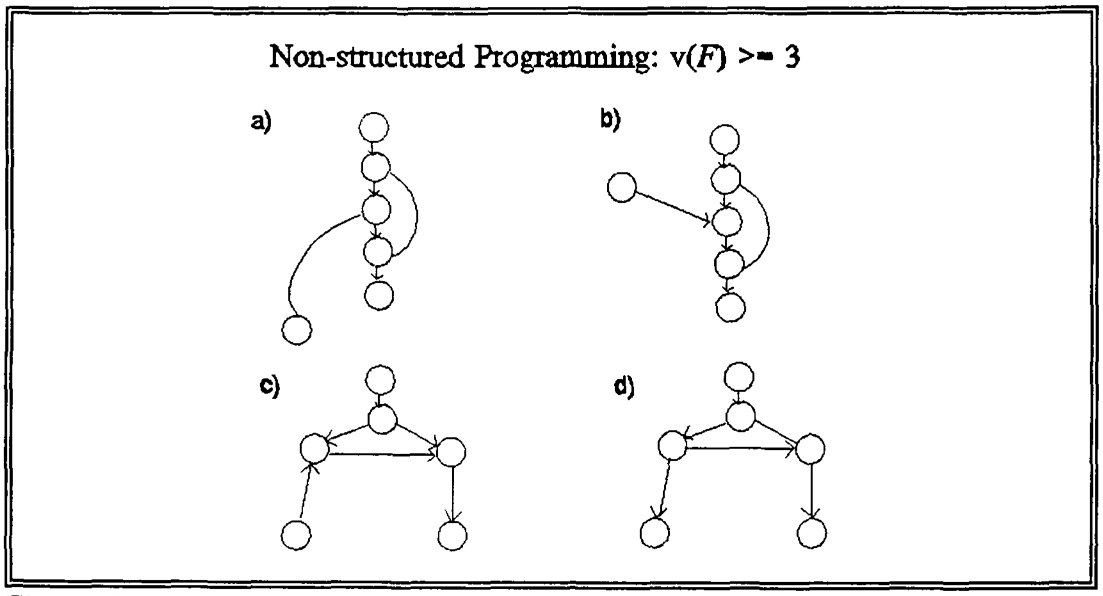


Figure 4-11 Characterising *v(F)* for non-structured programs

(v) More measures for flowgraphs

1 Control density

Mean number of decisions per node This is not a very useful metric as we cannot act on it

$$Dc = \frac{V(F) - 1}{number\ of\ nodes} \qquad \textbf{(Equation 4.17)}$$

2. Number of levels

Maximum number of nested structures plus one This is always one in sequential programming

3 Number of pending nodes

Number of auxiliary exits This is an indication of the ease of testing As the number of auxiliary exits increases so does the difficulty in testing

4 Number of degrees

Maximum number of edges connected to the same node (characterising a critical node) This is not as useful as it may appear since we cannot act on the information obtained It does not identify the most critical nodes.

4.5 Architectural measures

Complexity is not considered to be directly related to the number of lines in a program For example, if we had two 5,000-statement programs, the first having only one 5,000 statement component and the second having 50 components of 100 statements, we could not say that these have equal measures of complexity. What is required is a method for analysing the calling relations between components. To do this we require a graphical representation of calling relations between components, usually referred to as the *call graph*

Using graph theory we construct graphs consisting of components of the program (*nodes*) and the calling relation between two components (*arcs*) It is generally assumed that the call-graph has a dıstınguıshed 'root node' whıch corresponds to the hıghest level module, ı e an abstractıon of the entıre system

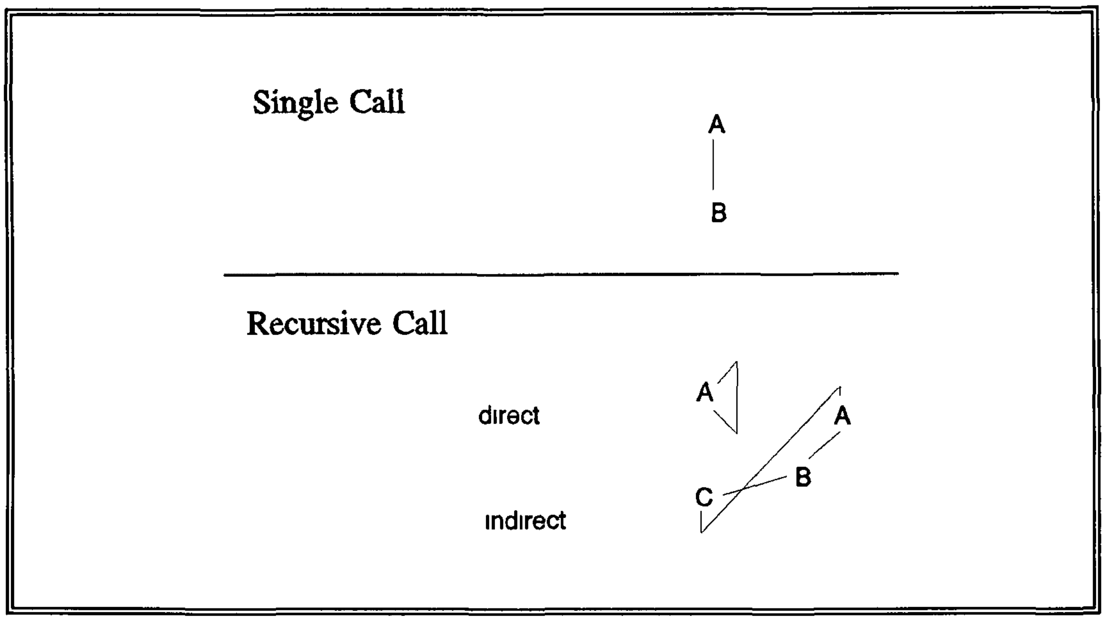


Figure 4-12 Call graph conventıons

In Fıgure 4-12 we read the graphs from top to bottom, so A can call B many tımes ıf the call ıs wıthın a loop structure The call graph helps hıghlıght recursıve calls, the overall hıerarch of the system, calls that skıp levels, ısolated components that may have mıssed the testıng process, nodes wıth a hıgh degree of arcs and helps ıdentıfy multıple roots to the system

4.5.1 Call graph measures

1 Sıze

Number of nodes

Number of arcs

2 Number of paths·

Paths goıng from root to final component

3 Hierarchical complexity

Mean number of components per level

$$H = \frac{Number\ of\ components}{Number\ of\ levels} \qquad \textbf{(Equation 4.18)}$$

4. Structural complexity *S*

Mean number of calls per components

$$S = \frac{Number\ of\ calls}{Number\ of\ components} \qquad \textbf{(Equation 4.19)}$$

5 Accessibility of a component *A(M)*

Measure of the ease with which a component may be accessed

$$A(M_k) = \sum_i \frac{A(M_i)}{N_i} \qquad \textbf{(Equation 4.20)}$$

Where

- $A(M_i)$ Accessibility of node with a line segment to M_k
- N_i Number of components called by M_i

6 Testability of a path

Indication of the ease with which a path may be tested

$$T(P) = \left(\sum_{M_i \in P} \frac{1}{A(M_i)}\right)^{-1} \qquad \textbf{(Equation 4.21)}$$

7 Testability of a program or logical structure

$$T(S) = \frac{1}{N} \left(\sum_{i=1}^{N} \frac{1}{T(P_i)} \right)^{-1} \qquad \text{(Equation 4.22)}$$

8 Entropy

Measure of disorderliness

$$H(G_A) = \frac{1}{|X|} Log_2 \frac{|X|}{|X_i|} \qquad \text{(Equation 4.23)}$$

Where

$\|X_i\|$	=	Number of nodes in path i
$\|X\|$	=	SUM of all $\|X_i\|$
N	=	Number of paths

4.6 Summary

These last few chapters have covered quite extensively the most commonly used measures in software projects This review however, is not intended to be exhaustive but rather offer a representative sample of measures at different stages of the development life cycle All measures presented have one thing in common however, they have all been defined under the influence of the development process of 3GL's The following chapter briefly looks at other development process, while chapters 6 and 7 describe newly defined measures for use with rule-based systems.

5 Measurement for new development methods

5.1 Introduction

As we have seen in chapters 3 and 4, a comprehensive set of measures have been defined for use in software development. Many of these measures were based on the study of program and system behaviour in research environments, and have still to be validated in the commercial world. Attempts have been made to collect data that would instantiate claims of relations between a particular measure and the software. A typical example is Boehm's COCOMO model where studies were conducted to establish parameter values for different development languages. Most of these values were computed for what are commonly called *third generation languages* (3GLs). The whole definition process of metric development has been strongly influenced by the structure of procedural 3GLs. This chapter examines a 'new' language generation and describes a process for developing or modifying existing measures so that prediction and assessment may be performed in this language.

5.2 Language generations

The earliest or *first generation* of computer languages was *machine code*. Programs were not interpreted or complied - the instructions were in a form directly readable by the machine. Computers were programmed with a binary notation. This situation was improved by the use of mnemonic codes to represent operations, although data was still in physical binary address Further advances were the introduction of decimal numbers to represent storage locations and addresses

The *second generation* of languages, which came into use about the mid-1950's was *symbolic assembly language* Symbolic address were used instead of physical machine addresses This advantage was due to the fact that when physical locations of variables or instructions had to be changed, the programmer did not have to re-enter the new physical addresses Popular languages were SAP (Symbolic Assembly Program, for the IBM 704), EASYCODER and AUTOCODER

The *third generation* came into use in the 1960's and were called *high-level languages* Scientific languages such as FORTRAN and ALGOL were introduced, and business languages such as COBOL also became quite popular Programs were becoming hardware independent, with little knowledge required about hardware registers and instruction sets Mathematical expressions and English keywords ensured that programs were easier to write Business applications required sufficient degrees of complexity that error detection became a very real time consuming task in the development of software The productivity levels of software firms became a very serious issue in the 1970's and led to the introduction of software control through measurement, as we have previously discussed

The term *fourth generation language* (4GL) wrongly implies an evolutionary step beyond third generation languages In reality, what is provided is a user-tunable application based on existing programming techniques, which in turn is a full service programming language Most 4GLs ride on top of a database, having been specifically designed to front end such data repositories Figure 5-1 is a representation of the development process from 3GLs to 4GLs The development of the 'new generation' was based on the need to increase the ease of use of the tools being created for existing languages This figure shows a COBOL compiler with a screen generator (eg Forms Management System), report generator, and a database Structured Query Language (eg SQL) The interface between these tools becomes cumbersome and inefficient when they are provided as extensions to existing languages. One solution is to develop a new layer of software above these tools with the ability to interface with these diverse facilities. These 'new' products have been called fourth generation languages, but since there exists no 'new' technology or development strategy, it could be stated that the development of fourth generation languages has been more the work of marketing policy than computer development.

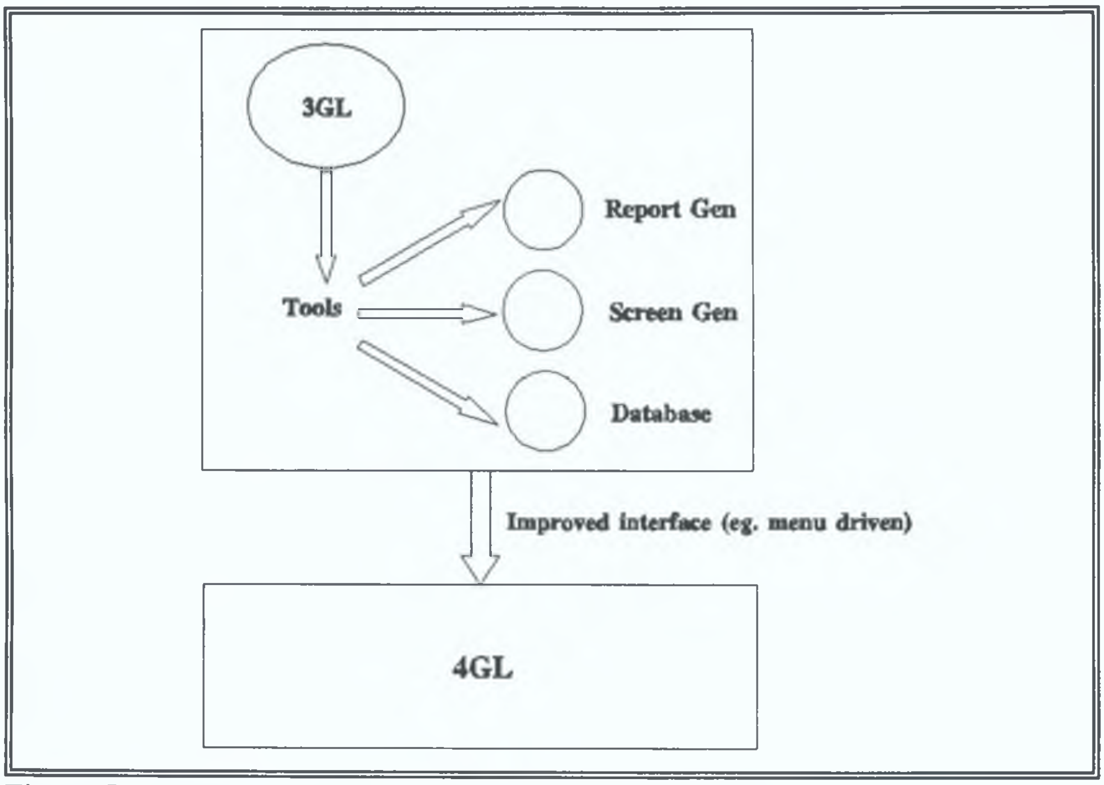


Figure 5-1 **Evolution of Fourth Generation Languages**

5.3 Fourth generation languages

4GL tools have made a significant impact on the productivity of software development. These gains have been credited to features such as ease of use, use of non-procedural code, direct access to database-management systems and reduction in the development time of systems by a significant degree.

The difference between third and fourth generation languages is constantly under debate but to give some reference to the more popular concepts we turn to J. Martin. According to Martin, third generation languages are higher level languages that are standardised and largely independent of hardware, and where system development requires a step by step specification of the program tasks [Martin 85] Fortran, COBOL, C and Pascal are among the more commonly used 3GLs. Programmers of these languages are required to rely on procedures composed of tokens which consist of commands, types and functions

Martin defines 4GLs as non-procedural, end-user oriented languages. However others have claimed associated features with these languages such as report/screen generators, integrated database systems, and ease of use Examples are DBase, Focus, Oracle, Mantis and Powerhouse These languages rely on predefined procedures for performing high level operations, eg sorting Such facilities are generally more powerful but less flexible than those offered in traditional high level languages and hence procedural tools are often required to perform tasks not provided by non-procedural facilities These however are not usually as expressive as 3GL tokens

5.3.1 Principles and components

There are many claims for the use and productivity of 4GLs Most of these claims are from vendors eager to demonstrate superiority in software development using their product Before we can begin to establish the reliability of these claims we need to examine some of the features and principles involved.

Norman Fenton [Fenton 91] claimed that fourth generation languages were designed to increase the productivity levels of software development because demands for software products was growing faster than developer's ability to provide them Martin, being more specific, provided the following list of objectives:

- To speed up the application-building process
- To make applications easy to modify, reducing maintenance costs
- To minimize debugging problems

- To generate bug-free code from high level expressions of requirement
- To make languages friendly so that end users can solve their own problems and put computers to work

Fourth generation languages should allow some applications to be generated with lines of code an order of magnitude fewer than for example, development in COBOL Thus alternative (and perhaps more accurate) names for these tools have been *High Productivity Languages*, *Non-procedural Languages*, or *Application Generators*

Most such languages are dependant on a database and its corresponding data dictionary The dictionary has in some cases evolved into a facility that can represent more than data It can contain screen formats, report formats, dialogue structures, associations among data, validity checks, security controls, authorisations to read or modify data, calculations that are used to create fields, permissible ranges and logical relationships among data

One major distinction made between third and fourth generation languages is the introduction of non-procedural code in the latter A procedural language specifies how something is accomplished, whereas a non-procedural language specifies what is accomplished without describing how Thus we can say that PASCAL is procedural since it contains a precise sequence of instructions for every action Application generators where the user fills in a form to specify the requirement, are non-procedural since there is no concern for the details of the implementation Consider the following example in SQL

SELECT user-name FROM employee-record
GROUP BY user-group

This leaves the software to decide how to extract the information from the corresponding records in the database, sort the user names in alphabetical order and list them

Martin made the point in 1985 that it was too early in the development of fourth generation languages for standardisation He felt that it was too early in the evolution of the technology to make such a move and that it could inhibit creativity It may be worth remembering that many 3GLs have disappeared over the last two decades Jean Sammet's book on programming languages listed 120 3GLs in 1969 [Sammet 69] (which did not include ADA, PASCAL or C), and of those 120, fewer than 10 are now in general use, and the ones that survived tended to be those supported by large organisations Similarly the reduction in the number of 4GLs currently available will be high, but unlike 3GLs the support and standardisation is still missing

5.4 Logic programming in software engineering

Logic in modern times was based on the work of George Boole, an Irish mathematician Boole's work resulted in proto-logic called propositional calculus Since then logicians such as Turner and Von Neumann have made enormous inputs to computer science, but it is only now however, that logic is becoming an important part of the education of computer programmers Logic Programming, whose main advocate was Kowalski [Kowalski 74] has been described as

"A process that involves the use of logic programming languages - Prolog is the best known - which enable one to write programs that approximate to a collection of purely logical statements " [Gibbins 88]

Software acts as the interface between man and machine, and is written in a language like format To ensure that such text is readable, an underlying logic is required It is generally agreed that the way we represent software is a compromise between what computing machines can be made to do, and what human beings can understand

Prolog as we have stated is one of the most popular logic programming languages It consists of a set of logical statements with each statement being either a fact or a rule Facts are categorical, rules are conditional The main idea in Prolog is to formulate a set of rules and facts in predicate logic together with a problem for which a solution is sought [Amble 87] Here is a small well known example.

```
mortal(x) if human(x)        //   x is mortal if x is human
human(x) if Greek(x)         //   x is human if x is Greek
Greek(Socrates)              //   Socrates is Greek
answer(x) if mortal(x)       //   x is an answer if x is
                                  mortal
==> Socrates                 //   This is an answer
```

Prolog has a syntax which is somewhat similar to predicate logic, and contains an inference engine It is a higher level language than, for example, Pascal as it conceals more of the operations of the computer Implementations of solutions in Prolog (where the problem is suited to this language) typically requires fewer lines of code than implementations in 3GLs

The above claims are similar to those made by producers of many 4GLs, and while it is not claimed that Prolog fits comfortable under this broad classification, it is noted that the use of rules and facts have been successfully implemented in commercial systems It is generally felt that knowledge based systems (where knowledge is stored as facts and rules), such as expert systems are considerably different from traditional high productivity languages, and many 4GLs use rules to implement non-procedural components of their systems, such as data integrity checks

5.5 Rule based system

The use of Prolog and other logic based languages has facilitated in the development of Expert Systems [Johnson et al 88] An Expert System or knowledge based system is a system which manipulates 'knowledge' in order to perform a task or tasks The knowledge in such a system is highly structured symbolic data which represents a model of the relationships between data elements and the uses to be made of them The performance of a knowledge-based system depends both on the quality of its factual knowledge and the ways in which this knowledge is applied

Perhaps one of the best known examples of a knowledge-based system is MYCIN [Shortliffe et al 73] which uses a knowledge base of rules to aid in diagnostic problem solving In this knowledge-based or rules-based system the knowledge is represented by domain specific rules Rules in the form of IF-THEN statements encode judgemental knowledge which can be 'fired' or 'activated by an initial query or by other rules In this way rules can be chained together So if Rule1 requires the evaluation of Rule2 we can say that they are chained together It becomes clear then that a non-procedural nature is inherent with these types of systems

5.6 Customising measurements for diverse system

Early in this chapter it was stated that measures that exist have been defined primarily for use with third generation languages As we have seen however, the direction that software technology has taken is leading us away from these development methods Fourth generation languages, logic programming, object-oriented programming, relational database languages and even expert systems are providing developers with tools that are abstracting even further from the implementation details of both software and hardware Little research in these areas has been published, although Verner and Tate [Verner et al 92] proposed a process involving Function Point analysis, which provides data for COCOMO, which is applicable to 4GLs A proposal for a suite of metrics for object-oriented languages has recently been published by Chidamber and Kemeerer [Chidamber et al 91] Other areas have been slower to attract the attention of researchers in the field of metrics. This chapter presents a generalised process through which measures may be defined for a wide range of software development methods

5.6.1 General measurement definition process

The following steps are intended to be a generalised process for obtaining a list of useable software measures which are directly applicable to a specified development processes

Step 1. Analyse the software

Step 2. Decide what components can be measured by existing techniques

Step 3. Investigate how (if possible) all other components can be measured

Step 4. Define a quality model unifying these measures

Step 5. Perform measurement and validation

Step 6. Re-iterate / modify

These steps, while appearing quite simplistic, attempt to integrate existing scientific measurement technology with new developments in a comprehensive practical fashion It is a guide for deciding if a product can be controlled by the use of measures, and if critical elements in the software cannot be measured It is as much a waste of time to perform inadequate measures as it is to not performing them at all As we have seen, a substantial number of measures do exist for specific stages in the software life cycle and without doubt some of these will be applicable to development methods with similar constructs The definition of measures for software components where none exist is not always a straight-forward task, much depends on what was revealed in the analysis of the language and what constructs are considered more critical to development success than others. Given below are further descriptions of the steps already outlined.

Step 1. Analyse the Software.

The success of this entire process is firmly based on the results of this step. If our analysis is incomplete then all future decisions will not be based on sound principles. It is a fundamental concept that a comprehensive understanding of software and its components is required before rational decisions regarding measurement are feasible.

By the use of the term software we refer to both the documentation and the source code. The list of available software components should be itemised. Typical components include:

- product specification
- product design
- program language or languages
- program source code
- user instructions
- development test plan and results

The stage of the development process will greatly influence the available software components, but usually the development language will be decided upon by the end of the specification stage.

The object of analysis is to clearly indicate what components exist in order to allow decisions to be made regarding applicable measures. For example, not all developers produce specifications, and those that do may not have structured it in such a way that Function Points would be applicable. Also the production of design documentation may not be in a standardised form, eg. SSADM (Structured Systems

Analysis and Design Methodology), and hence measures such as Design Weight which rely on basic counts such as *number of modules* and *number of control tokens* may not be computable

If more than one programming language is used then each should be analysed in turn, and the specific areas in which they are to be used should be determined The analysis of the development language involves decomposing it into its various development tools and determine what strategy has been employed for each component If we take a commercial 4GL language, typical components would be a Forms Management System, Report Generator, Query language, Database and perhaps a Data Integrity Manager Each of these components employ diverse implementation strategies For example, a token based language (Pascal-like instructions) would be procedural, containing textual elements and relations, whereas a forms manager would have a non-procedural nature So one method of classification for each component would be to identify procedural and non-procedural parts Further detail relating to these classifications could also be provided, and below are some preliminary suggestions

- Textual components
- Module structure
- Calling relationships
- Data coupling
- Control structure

Upon completion of this step, a table containing each component of the software, along with a comprehensive description of each, should be produced

Step 2. Decide what components can be measured by existing techniques

Based on the first step, each of the components should be examined and a decision made relating to its applicability to existing measures The approach taken in this research has been to present measures based on the stages of the software life cycle With the identification of components which are also stage dependent the task of reviewing measures is simplified The reason for the presentation of such a wide range of measures was to give some indication of the diversity of measures available Using the categories in section 3 1 figure 5-2 identifies the main categories of measures, a systematic process of relating measures to software components should be initiated Those components for which no measures are available will be examined further in step three It is worth noting that although many development methods are not strictly procedural, if they have a similar development life cycle then existing measures may be applicable For example if a specification and design stage are present, measures are already defined Also, the presence of textual components, or procedural text may lead to the possible use of Halstead's measures

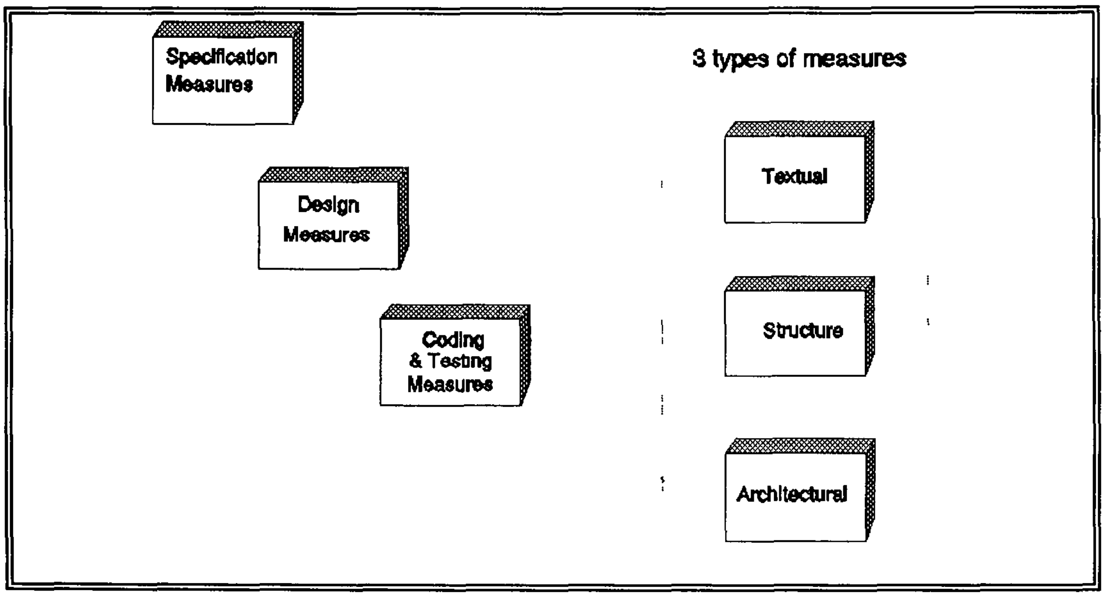


Figure 5-2 Associating measures to software life cycle stages

This step should associate measures with analysed components of the software, and highlight components where measures are lacking These components are then analysed in the third step

Step 3. Investigate how (if possible) all other components can be measured

Little help is available for this section Often it is just valuable to be aware of what factors are <u>not</u> being measured and that are outside our prediction or assessment models. What is required, if measures are to be developed, is a further detailed description of the software components. This should help to identify aspects of importance within the software development process. It is also a good idea to confer with people who have experience with these tools in order to gain some practical understanding of what are typical maintenance problems, complexity factors, and other relevant issues.

The focal point of this research has been to develop measures for a rule-based system. This will be described as a practical application of this stage in the following chapter.

Step 4 Define a quality model unifying these measures.

In chapter 2 the concept of defining product quality models was introduced. Measures are provided as a tool to provide some 'proof' about the 'quality' of attributes with which we are interested. Keeping this in mind we should take the measures we have identified and/or defined and ensure that they are helping us to obtain some required form of verification of particular attributes. It is quite wasteful to collect data upon which no actions can be made or for which we have no use.

We have already described the ISO9126 model of software quality. This model however, is just one of many such models in existence. Very often a company will produce its own version of 'quality' by identifying external attributes which are most interesting to their customers or quality controller.

Let us break down the notion of quality and look at two proposed models by Boehm and McCall. Both approaches identify a set of characteristics which are listed below.

- Sensitive to the environment (user) -> Quality Factors
- Decidable from within (developer) -> Quality Criteria

- Measurable (controller) -> Quality Metrics

In this system factors are characteristics of the software, seen as black boxes, while criteria are characteristics of the software, seen as a white boxes. In Figure 5-3 we reintroduce the quality tree concept. Using this decomposition a software engineer may monitor software quality.

It is most likely that a new quality model will be defined (or at least an existing one modified), when new measures are developed. This could possibly involve identifying new criteria or replacing specified metrics.

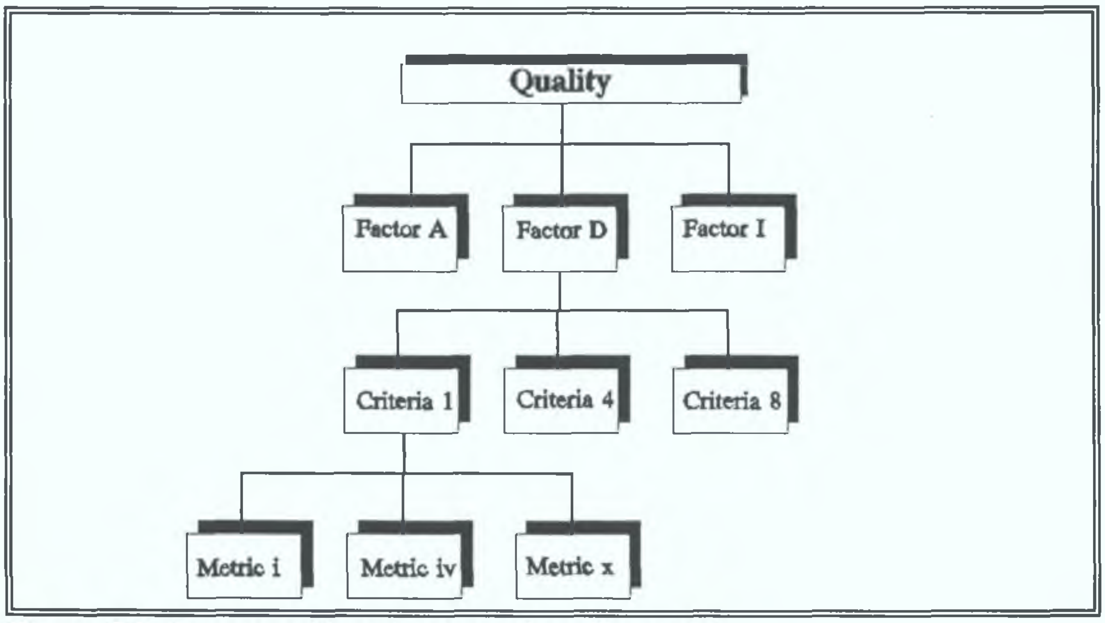


Figure 5-3 Decomposition structure of quality model

Step 5. Perform measurement and validation.

The first part of this step is to collect data obtained from performing the measures defined for use in the specified quality model. The process of data collection was introduced in chapter 2 where typical methods were described. This activity has been described as follows:

"Data should be collected with a clear purpose in mind. Not only a

clear purpose but clear idea as to the precise way in which it will be analysed so as to yield the desired information " [Moroney 50]

The immediate result of data collection is *raw* data To obtain direct measures from this data we need some form of *extraction* Our first step is to decide what to measure, then decide how the indirect measures will be calculated, and hence what direct measures are needed for analysis

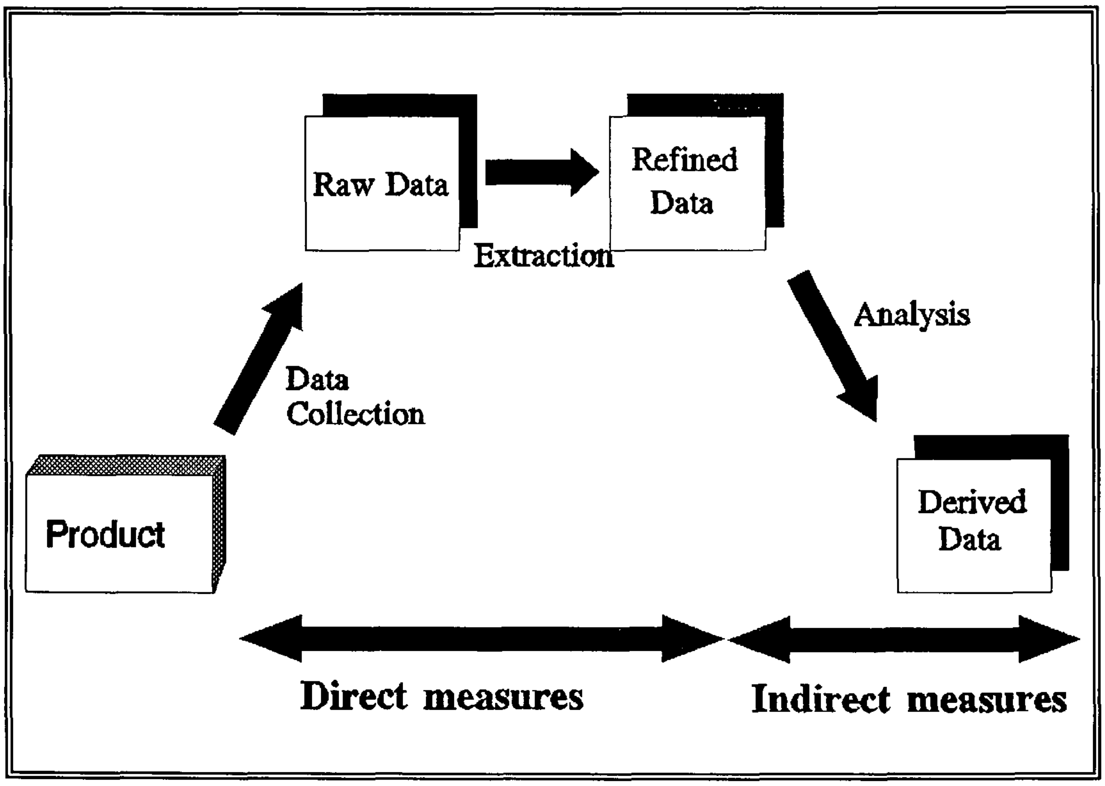


Figure 5-4 The role of data collection in measurement

The collection of data will typically span several phases of the development life cycle It is an on-going process and not a fixed step, as some would believe As this raw data is collected it should be stored in a database which will allow easy analysis when the collection phase has been completed

When we refer to 'analysing software measurement data' we assume the following

- We have a number of measurements of one or more attributes from a software entity, which can be referred to as a *dataset*.

- The software items are comparable (modules from the same product, similar projects in the same company, etc.)

- We wish to establish characteristics of the dataset, and/or relationships between attributes.

Details of analysis techniques for data collected is beyond the scope of this thesis, however Fenton's book *Software Metrics* provides comprehensive statistical techniques for software measurement validation. Below is a summary of some of the more standard methods of data analysis.

Datasets of software attribute values must be analysed with care because software measures are not usually normally distributed. It is advisable:

- To describe a set of attribute values using the box plot statistics based on the median and upper and lower fourths rather than the mean and variance. [Hoaglin et al 83]

- When investigating the relationship between two variables always inspect the scatterplot visually.

- To use robust correlation coefficients to confirm whether or not a relationship exists between two attributes. [Siegel et al 88]

- In the presence of atypical values, to use robust regression to identify a linear relationship between two attributes, or remove the atypical values before analysis [Sprent 89].

- To use Tukey's ladder to assist the selection of transformations when faced with non-linear relationships [Tukey et al 77]

- To use principle component analysis to investigate the dimensionality of datasets with large numbers of correlated attributes

Further recommended reading material is [Dobson 90] and [Draper et al 66], which provide good introductions to the principles of generalised linear models and classical regression analysis

Step 6. Re-iterate / modify

It is traditional to include such a step in most processes to allow for corrections in the original method As more data becomes available since our initial definition of measures, validation and analysis may indicate required changes in our proposed models Such 'fine-tuning' is required to ensure that measures are indeed useful and are associated with specified attributes

5.7 Summary

The process for defining new measures as stated in this chapter is not intended to be a formal methodology, but rather a guide to utilising existing measures and identifying when and where new measures are required The approach to measurement within software engineering should be scientific and not haphazard, which is too often the case When measurement is required it should be well understood what is to be measured, and how these measures are to be implemented

Now that these steps have been presented, and are based on standard approaches to software assessment and prediction, they will be used in the next chapter to develop a new set of measures for a rule-based language

6 Defining metrics for a rule-based language

6.1 Introduction

The requirement for the existence of measures for software systems has already been stated The last chapter gave a general set of stages which could be used to develop these measures for systems other than 3GLs These steps however were not too specific in the area of actual metric definition, which is more an intuitive problem than a simple analytical one To define measures for a language requires some insight into the languages structure with an understanding of how that language is used in software development To demonstrate this definition of metrics, a case study using a rule-based fourth generation language called RULER has been used The stages outlined in the previous chapter will be used where applicable Although the initial stages are relevant, further data collection and research will be required to complete the validation of new measures as required by later stages.

6.2 The RULER language

Ruler is a 4GL for application development on Digital's VAX range of computers using Digital's record manager RMS, and also for use on IBM PCs This language has been designed and developed by an Irish software firm and it is only available at present in this country [Ruler 87] Many of the standard claims made by 4GL vendors such as increased productivity, high level and non-procedural characteristics, integrated data dictionary and reduction in maintenance requırements, can also be applıed to thıs language

RULER allows the programmer to record wıde-rangıng knowledge related to applıcatıon data ın a *data dıctıonary* Thıs knowledge ıs recorded as *ıntegrıty rules* usıng a specıally desıgned procedural langauge These rules define, *valıdatıon constraınts*, *reverentıal constraınts (joıns)* and *computatıon of derıved fields.*

6.2.1 Program Development Methodology

The waterfall development lıfe cycle descrıbed ın sectıon 3 1 stıll applıes to thıs language, however specıficatıon and desıgn phases become the focus for actıvıty.

The analysıs actıvıty usually results ın an ınıtıal Entıty-Relatıonshıp model and an ınıtıal specıficatıon of requıred busıness functıons to be ımplemented Then a prototype ıs developed based on the desıgn and specıficatıon to allow users to verıfy whether the developer's understandıng of the system ıs the correct one Contınued expansıon and refinement of the prototype ıs requıred untıl the system ıs completed.

The functıons ıdentıfied ın the analysıs phase are ımplemented as programs. RULER offers five dıstınct program types, On-lıne, Report, Batch, Menu and Chaın. Each of these programs ıs specıfied usıng a consıstent easy-to-use form fillıng ınterface for standard programmıng attrıbutes, and a full screen WYSIWYG edıtor for more complex attrıbutes

Usıng the Entıty-Relatıonshıp model records and files are defined The data dıctıonary supplıes ınformatıon on record relatıonshıps, ensurıng record and field names are valıd, and also supplyıng ınformatıon about record layouts and field formats so that default forms can be generated In addıtıon the data dıctıonary contaıns ıntegrıty rules for types, fields and files

6.3 Defining measures for a rule-based language

The main focus of this research is to provide a set of measures for a rule based language The steps provided will be used where appropriate, but the main effort will be based on the third step This should provide a practical example of how measures can be defined for languages that are not necessarily procedural Later steps such as validation require a great deal of data and only initial statistical analysis for measures defined will be provided.

6.3.1 Analyse the language

The analysis of RULER will be primarily focused on the identification of product measures These measures will be based on the implementation phase and not on the earlier stages which are to some extent implementation independent

As we have described, RULER is a rule-based 4GL which runs on the VAX under VMS It use the VAX/VMS Record Management System (RMS), and its own Dictionary Management System (DMS) The DMS is considered to be the core of the system and it is here that data types, fields and files are described and stored along with related rules, especially those related to integrity However, before we provide detailed descriptions we must first identify all of the components within a typical RULER product

A. Design specification

For applications to be developed using RULER it is important to ensure that the initial stages of specification and design are performed with more enthusiasm than is typically shown for 3GLs SSADM is often used to ensure a more rigorous specification of the system design The following are typical components within SSADM

Entity Relationships - This model depicts the system entities (or data stores) and specifies the relationships between them It can also specify the numerical relationships between connected entities This model can be easily translated into a physical data base implementation

Data Flows - Data flows define the data requirements of the system and highlight the required system functionality. Identified processes can be implemented as modules within the system, and the data required for each process is also specified.

Entity Description - This is intended to provide a complete description of the data to be implemented in the system Required data is specified by the data flows and it is then decomposed into individual data items These can be further specified with respect to its format (for example char, numeric, alphanumeric) and its length This information can be directly entered into the data dictionary, along with comments and rules (for example: specified allowable ranges for numeric data items)

B. Dictionary management system

The data dictionary is the 'core' of the system as it is where all data is described Note that data is <u>stored</u> in the RMS files Once data has been defined programs may be generated which automatically include these data descriptions The type, field and file entities are described in the data dictionary and have the following relationship

"A *file* has many *fields*, each field is on only one file (if a similar field is on another file, then a new field is created), each field refers to only one *type*, a type can be referred to by many fields "

The types are the primitive data items required for applications Once a type has been defined, along with any rules for validation, it can be referred to by many fields; each field will automatically inherit the specification of its type, but can also have further rules associated with it

Files describe the details of an RMS file record structure and its indices RULER treats the terms *file* and *record* synonymously since each file is assumed to have one record type for each application

As stated the rules are allowed within the data dictionary which ensures the inclusion of specific knowledge to be contained within applications These rules are textual, and similar in syntax to many traditional procedural languages (such as Pascal)

C. Program generators

There are five program generators used to create specific types of programs, On-line, Menu, Chain, Batch and Report Details of each are given below

On-line Program Generator - This allows you to specify the program details and will either generate a default, or use a previously edited form A *forms editor* is used to change this form layout, both background text and data fields The executable 'program' is then generated which includes all relevant dictionary rules for validation, derivation, and file-lookup This final 'program' can then be run.

Menu Program Generator - A default menu form is created and can then be edited using the standard Form Editor The menu actions are then specified, as a set of choice/action pairs, the choice being what the user would enter on the menu form and the action being the name of the program to be run

Chain Program Generator - Chain programs are intended to link together a suite of related programs, so that when the chain program is executed it initiates an environment within which the full suite of programs available can be executed.

Batch Program Generator - This is very similar to the report program except for minor difference in syntax and file related options.

Report Program Generator - This involves specifying a primary file for reporting, choose required 'linked' files, specifying other report requirements such as sequence fields, width, page size and any processing and/or selection criteria using a textual rule based language. The system can generate default report layouts.

Table 6.1

Component	Details
Design Specification	ER model
	Data Flows
	Entity Description
Dictionary Management System	Data descriptions and relationships.
	Textual/procedural based rules.
Program Generators	On-line: Forms/procedures
	Menu: Forms/procedures
	Chain: Forms/procedures
	Batch: Forms/procedures
	Report: Forms/procedures

All of the program generators above are <u>form based</u> with default values applicable to some fields. Because of the specific nature of each, little except the creation of file and field relations, screen layouts and addition process commands are required. Each of these programs contain an optional procedure section where rules, similar to those in the data dictionary, are used. Variations in syntax between programs exist but these rules represent a procedural textual component similar to those in 3GLs. Table 6.1 summarises this information which will be used as the basis for step 2 to help identify those components which may be measured using existing techniques

6.3.2 Decide what components can be measured by existing techniques

This step would typically involve an extensive search of publications for articles relating to measurable attributes of software Chapters 3 and 4 gave a comprehensive description of existing measures and at what stage of the system life cycle these may be used Using these measures the components identified in Table 6 1 will be categorised into those than can be measured, and those which cannot

6.3.2.1 Design specification

This component contains a standard method for program and system specification Structured design and specification measures have been described in chapter 3 and relate directly to the type of designs produced for RULER systems.

DeMarco's "Bang" metric

This is based on the specification documentation, and has been defined for data strong systems which involve large database applications This would be suitable as it is an implementation independent functional measure indicating system size

Function points

Another measure for use in early cost models is FP, also described in section 3 2 2 This measure can be used as input to the COCOMO cost model However, an FP to LOC expansion ratio for RULER would be required to provide an estimate for the lines of code parameter of COCOMO

Research by Verner and Tate provide a process for estimating size and effort in 4GLs using FP and COCOMO [Verner et al 92] As we have seen Function-Point analysis is used for sizing, and COCOMO for effort and schedule estimation, so to make these tools 'fit' a 4GL development process estimates need adjusting

Design weight

DeMarco also provides a design measure (see section 3 4 1) which requires basic counts obtainable from Data Flow diagrams and would be applicable to documentation specifying RULER programs/ systems This measure provides further information related to the effort implied in the design and hence could be part of a cost model

6.3.2.2 Dictionary management system

This component contained two distinct sections, the database and its relationships, and the textual rules which may be associated with the three identified entities types, fields and files

The data description and relationship portion of the database have no directly related measures Indeed there is little research available for this area of measurement It will be required to identify some measures relating to the database structure in the third step

The textual rules which may be incorporated into the data dictionary may be viewed from two perspectives The first is that they are just textual tokens consisting of operators and operands, and as such, standard Halstead textual based measures may be applied The second is to view this text as a representation of knowledge relating to the data, in the form of rules or integrity checks Again little documentation of measures relating to rules or integrity in systems is available and this will be discussed in the third step

6.3.2.3 Program generators

All program generators are similar in the method through which programs are generated. Most of the relationships between files and programs are set by defaults but can be modified. If further computation of the data is required then a procedure may be written. These procedures contain a specially defined language which is similar but more extensive, than the one provided in the data dictionary. Comments relating to the textual rules in the DMS are also relevant to these procedures.

The structure of the modules or programs, although simplified by RULER is similar to that of 3GL languages. A calling relationship can be identified and architectural measures discussed in chapter 4 are applicable, although recursion is not permitted.

Structural measures can also be used in Batch, On-line, and Report programs where there is a often a high use of procedural code, however the other two program generators do not contain typical complex structures of procedural languages as defined in section 4.4.2. Decomposition of directed graphs into prime subgraphs may be performed within instances of programs, however unique decomposition of rules would be more useful. This issue will be addressed in section 8.2.

We can conclude that within the Program Generators, nearly all of existing procedural measures are applicable, however they fail to measure the behaviour of <u>rules</u>. Measures relating to rules within programs and the data dictionary will be discussed in step 3.

With relation to the rules, dynamic measures may also be defined to help estimate the test coverage of test cases. Such measures are usually identified for test coverage of source code, but could be expanded to include rules. This will also be discussed in the following step.

Table 6.2 summarises this information which will be used as the basis for step

3. Here we can see the components which we can measure using existing software measures and also identify those which as yet require some form of development.

Table 6.2

Summary of measurable components of RULER

Component	Detail	Measures
Design Specification	ER model	BANG for data strong systems.
	Data Flows	
	Entity Description	Function Point and COCOMO model
Dictionary Management System	Data descriptions and relationships	To be defined
	Textual/procedural rules	Textual measures Rule measures to be defined
Program Generators	On-line: Forms/Proc.	Textual measures
	Menu: Forms/Proc.	Structural measures
	Chain: Forms/Proc.	Architectural measures
	Batch: Forms/Proc.	
	Report: Forms/Proc.	Rule & Dynamic measures to be defined

6.3.3 Investigate how (if possible) all other components can be measured

RULER can be considered to contain four layers of rules, three within the DMS and the other at the module (program) level, with each layer associated with one of the four entities (types, fields, files and modules). All measures identified relating to rules may be applied to these four entities. We can view the first three as the dictionary integrity rules that specify, in more detail, the contents or accessability of the data.

One primary difference between these layers is the permissible level of complexity of the rules. Figure 6-1 illustrates the relationship between each entity showing an optional addition of rules of entity

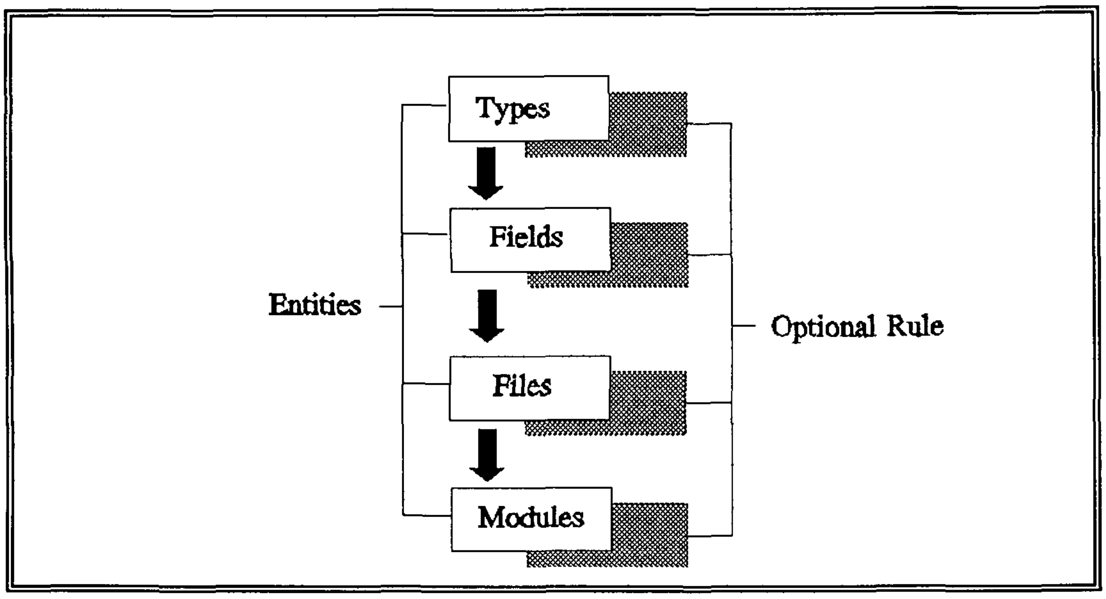


Figure 6-1 Relationship of rules within specified layers

The results from measures applied to each of the four layers may be plotted in an attempt to characterise the relationship between these layers Figure 6-2 graphically represents the possible tolerance values for our proposed measures An average result of the measure is obtained over the four levels Deviations from this average are then plotted to identify differences in these measures for each layer, this characterisation is indented to provide an indication of the typical results obtained from measurement Overlays of standard measurement characteristics with obtained ones could offer similar information to a kiviat graph [Logiscope 90], and help identify possible problematic areas within the software

6.3.3.1 Metrics associated with rules

Before measures are presented for use with RULER the format of rules within each entity are provided to indicate the difference between each In general rules consist of one or more statements which differ for each entity by the number of primary and subordinate statements possible Example 6 1 provides the format that rules on all layer take

```
Primary statement
Primary statement
        Subordinate statement
        Subordinate statement
Primary statement
```

Example 1: General format of rules within entities

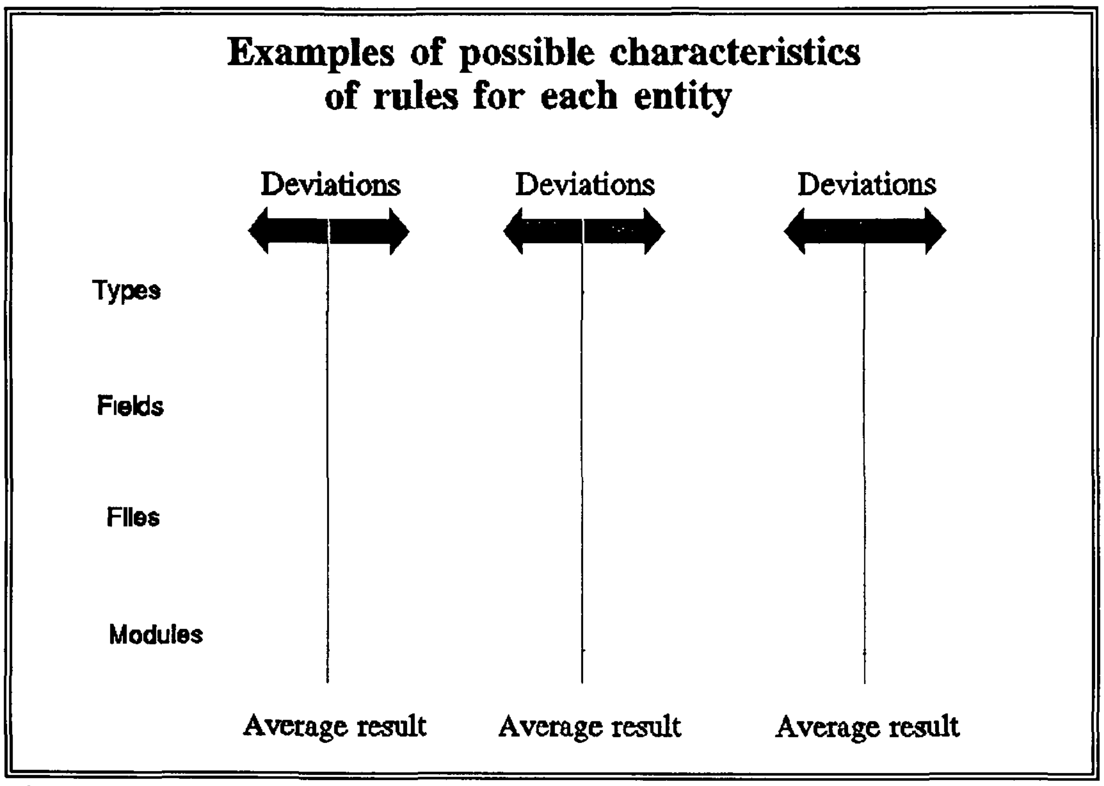


Figure 6-2 Graphing characterisations of measures for each entity

Table 6 3 provides the set of allowable primary and subordinate statements for each entity

A. Type rules

Rules at this level are provided to allow the program to define customised data types These rules will be automatically activated when a data item of that type is manipulated in any way

Table 6.3

Statements within entities

Primary Stm.	Entity	Subordinate	Entity
STORE AS	Type	PRINT REJECT ASSERT IF	Type Field File Module
ASSERT IF	Type Field File Module	Assignment NULL Function/ routine call	Field File Module
DEFAULT SOURCE EXTERNAL Function/ routine call	Field File Module		

```
Assert CUST_NO eq 0 or CUST_NO in all CUSTOMER.CUST_NO
        Else    Print "Invalid Customer Number",
End Assert;
```

Example 6.2 Typical *type* rule

The rule in Example 6 2 ensures that a field of type CUST_NO is accepted only if its value matches with some record in the file CUSTOMER, or has a value of zero, and the match is performed against the CUST_NO field of the CUSTOMER record

B. Field rules

Rules at this level are used to control values of record fields These rules will be automatically activated when a data item is entered or altered

```
Derive TOTVAL = VAL1 + VAL2 + VAL3;
        Assert TOTVAL < 1500
        Else
                Print "Total value exceeds 1500";
        End assert;
```

Example 6.3: Typical *field* rule

The rule in Example 6 3 ensures that the value of the field TOTVAL is derived using three other values VAL1, VAL2 and VAL3, and that this value does not exceed a specified range

C. File / record rules

Integrity rules which are not logically associated with just one field can be specified at the file (record) level in the Dictionary rather than the field level Field rules are active when data is input whereas file rules are only activated when the record is committed

```
If deleting order
        Assert NO_SHIPMENTS eq 0
        Else
                Print "Cannot delete order, shipments exist";
                Reject;
        End assert;
```

Example 6.4: Typical *file* rule

This rule ensures that before the ORDER record can be deleted we must assert that the field NO_SHIPMENTS is zero If this is not the case then a message is returned to provide the reason for not completing the deleting instruction

6.3.3.2 Module rules

Module rules are contained within the procedure section of a program form This procedure is an optional addition to the function of program, and usually involves a further set of rules relating to the data Rules within the Menu and Chain programs are primarily for organising program calling structure and will not be included in the analysis of rules for procedures The number of primary and secondary statements applicable to rule within modules is too great to present here, for further information regarding RULER syntax, refer to the RULER manual Example 6 5 provides an typical statement within such modules The syntax reads very similar to most structured 3GLs

```
Derive EXCH_RATE_TO = CURRTO.EXCH_RATE_PL;
Derive VALUE_IR_TO using
        If CURRCODE_TO = 1
                VALUE_IR_TO = VALUE_CURR_TO,
        Else
                VALUE_IR_TO = VALUE_CURR_TO/EXCH_RATE_TO;
        End if;
End derive;
```

Example 6.5: Typical *module* rule

6.3.3.3 Defining measures for rules and data

Having looked at the structure of RULER and the syntax employed within both the DMS and modules we can now provide a set of primitives (p-counts) relating to RULER from which a set of composite measures may be defined These measures can be collected as soon as the source code has been completed Although more primitives may be conceived, the following represent the most essential ones

As previously indicated there are four entities within RULER, and since many of the measures we shall define are apphcable to these entities, a special naming convention for has been developed for p-counts Those p-counts listed below are all measures obtainable from either of the four entities To demonstrate this the letter x has been inserted, where x is an element of the following set {T, F, R, M}, which specify type, field, record and module respectively

Table 6.4

Primitive counts (p-counts) measurable in RULER

p-counts	Description
Dx_{ij}	count of the number of rules containing data item j in the ith *entity*
FDx_{ij}	count of the number of rules fired for data item j in the ith *entity*
Rx_i	count of all rules in the ith *entity* (including sub-rules)
DIx_i	count of all data items in the ith *entity*
$UDIx_i$	count of all unique data items in the ith *entity*
Cx	count of all *entity*
SRx_i	count of all sub-rules for the ith *entity*
CRx	count of all *entities* containing rules

Where the value of x is explicitly stated there it is not possible to perform this measure any other entity other than the one specified To ensure a clear understanding of the terms used the following definitions are provided

Rule A statement within either the DMS or the module that is constructed by the programmer in the form of procedural code In the general case an assignment is considered to be a rule, however, more complex rules such as If-Else statements containing a comparison and up to two assignments, are counted also as one rule If another rule is nested within an If-Else pair this is considered to be a sub-rule (see below)

Data Item· A vanable named withn a rule in either the module, record, field, or type definition For the purpose of measurement vanables which appear more than once in a rule are only counted once

Sub-rule. Rules below the first level of nesting, such as those within nested IF statements

Below is a set of nine initial composite measures based on the p-counts in Table 6 4, which may be evaluated for all four entities

A. Data criticality

This measure gives the average number of rules that a data item appears in This measure can be recorded for either individual instances of an entities, given by ADx_i, or for all instances of the entity, given by ADx This should give an indication of how many rules are associated with a data item in the same entity

Using the Equation 6 1 below we can observe a standard level of criticality on an individual instance of an entity which can help identify anomalous instances For example, if we determine that 95% of modules result in a value within a reasonable range (determined using historic data), we can investigate further the remaining 5% of modules to determine the reason for the higher level of criticality

$$ADx_i = \frac{DIx_i}{UDIx_i} \qquad \textbf{(Equation 6.1)}$$

A variation of this measure is to obtain the same information except to ignore instances of the entity, given in Equation 6 2 Using this formula we can determine the criticality of data items within a specified entity This provides us with a wider indicator of the sensitivity or criticality of the data

$$ADx = \frac{\sum_{i=1}^{n} DIx_i}{CF}$$ **(Equation 6.2)**

To help identify possible criticality problems in data items it is best to set tolerance ranges which act as indicators to specific areas of code. When average values are used it is quite easy to have isolated anomalies undetected due to the weighting associated with a large volume of 'acceptable' results The presence of a maximum value can pinpoint individual cases that are beyond the acceptable ranges set (again these ranges are set based on historical data) Equation 6 3 provides the formula for obtaining this maximum level of criticality

$$MDx = \max\ (Dx_{11}, \ldots, Dx_{nm})$$ **(Equation 6.3)**

One adjustment which could be made is that the tolerance ranges indicated could be adjusted for the volume of rules and data items within the system. A volume measure is proposed in section 6 3 3 4

B: Rule complexity

The complexity of the rules is an indication of how difficult it will be to maintain the system Yet again we can record this measure for either individual instances of entities, given by ADIx$_i$, or for all instances of the entity, given by ADIx This gives an indication of the average complexity of rules within the same entity, based on the number of data items they manipulate

Using Equation 6 4 we can observe the level of complexity on a individual instance of an entity This can help identify specific entities that are more complex and hence possibly less maintainable than others

$$ARx_i = \frac{DIx_i}{Rx_i}$$ **(Equation 6.4)**

A variation to this measure similar to that given for data criticality is to obtain the same information for all instances of the entity This formula is given in Equation 6 5

$$ADIx = \frac{\sum_{i=1}^{n} DIx_i}{\sum_{i=1}^{n} Rx_i} \qquad \textbf{(Equation 6.5)}$$

The more data manipulated within rules the more errors likely at coding, and the longer it will take to modify the code if required This measure will need a tolerance range which may be used to indicate when rules are too 'complex' within an entity, different ranges will be required for different entities

Yet again we must ensure that in the process of measurement we do not miss rules with unacceptable levels of complexity because we are using an average measure The presence of a maximum value can help isolate specific rules with suspect levels of complexity

$$MDIx = \max\left(\frac{DIx_1}{Rx_1}, \ldots, \frac{DIx_n}{Rx_n}\right) \qquad \textbf{(Equation 6.6)}$$

C. Descriptive measures

A set of measures primarily defined to provide a general description of the system can be defined quite easily Trends may be observed to exist in systems and the deviation of systems from those trends may indicate added complexity within the system Further investigation would then be required An example of four such descriptive measures are given below

The formula for determining the percentage of entity x containing rules is given in Equation 6.7. This measure can be used to indicate the use of rules within dıfferent entıtıes An example of trends wıthın the module entıty, would be that rules are more scarce ın On-lıne programs than Batch programs

$$\hat{Px} = \frac{CRx}{Cx} * 100 \qquad \textbf{(Equation 6.7)}$$

Equatıon 6 8 evaluates what percentage of rules wıthın the ıth entıty are ın fact sub-rules Thıs measure ıs sımılar to the depth of nestıng measure defined for 3GLs The dıfference however ıs that rules wıthın RULER are not often nested Such nestıng ıncrease the complexıty of the system as ıt effectıvely creates another 'layer' of rules assocıated wıth a data ıtem

$$PNx_i = \frac{SRx_i}{Rx_i} * 100 \qquad \textbf{(Equation 6.8)}$$

We can present thıs last measure ın another format Thıs tıme we refer to the complete system and we are not focus on ındıvıdual ınstances of an entıty Equatıon 6 9 calculates the percentage of rules wıthın all ınstances of entıtıes that are sub-rules

$$ANx = \frac{\sum_{i=1}^{n} SRx_i}{\sum_{i=1}^{n} Rx_i - \sum_{i=1}^{n} SRx_i} \qquad \textbf{(Equation 6.9)}$$

Now that a set of composıte measure have been defıned a example ıs provıded to demonstrate theır applıcatıon Usıng an extract of code from a batch module, gıven ın Fıgure 6-3, we can demonstrate the applıcatıon of these measures to entıty module. Sınce we are only usıng an ınstance of the entıty there are only four measures lısted above whıch can be ımplemented The measure to be evaluated are ADM_i, ARM_i, PNM_i, and ANM_i, whıch need the followıng p-counts, $UDIM_i$, DIM_i, RM_i, SRM_i The results of all these are presented ın Table 6 5

```
if T_RATE = 1
        WORK_RATE = EXCH_RATE_PL;
else
        WORK_RATE = EXCH_RATE_BS;
end if;

NUM_CNT = NUM_CNT + 1;
NUM_DAYS = T_AGING_DATE - NUM_CNT;

if NUM_DAYS in 0 to 30
        CURR1 = OS_BAL_CURR * -1;
else
        if NUM_DAYS in 31 to 60
                CURR2 = OS_BAL_CURR * -1;
        else
                CURR3 = OS_BAL_CURR * -1;
        end if;
end if;
CURRT = OS_BAL_CURR * -1 + NUM_CNT;
```

Figure 6-3 Example of a *Batch* program procedure

Table 6.5

Results of Applying Module measures

P-counts	Results	Metrics	Results
$UDIM_i$	12	ADM_i	1.5
DIM_i	18	ARM_i	3
RM_i	6	PNM_i	16.6
SRM_i	1	ANM_i	0.2

All of the p-counts used to obtain these measures may be collected automatically which would mean that a static analyser could be designed to generate measures for RULER objectively and efficiently. A prototype tool which attempts to perform some of these p-counts is described in the following chapter. The values obtained for the example in Table 6.5 could then be compared to other modules to help determine it's relative complexity. Using a large sample of data, acceptable ranges for results are determined. Only when these have been defined can be begin to make some decisions based on the values obtained in our example above.

One way to provide both a visual representation of the rules and a diagram from which p-counts can be measured is to construct a Rule/Data (RD) relationship graph. Nodes on the graph are either *data items*, *rules*, or *sub-rules*. Arcs represent the relationship between nodes. Typical relationship exist between data items and rules, data-items and sub-rules, and rules and sub-rules. Using the extract rules obtained for Figure 6-3 a RD graph has been constructed in Figure 6-4.

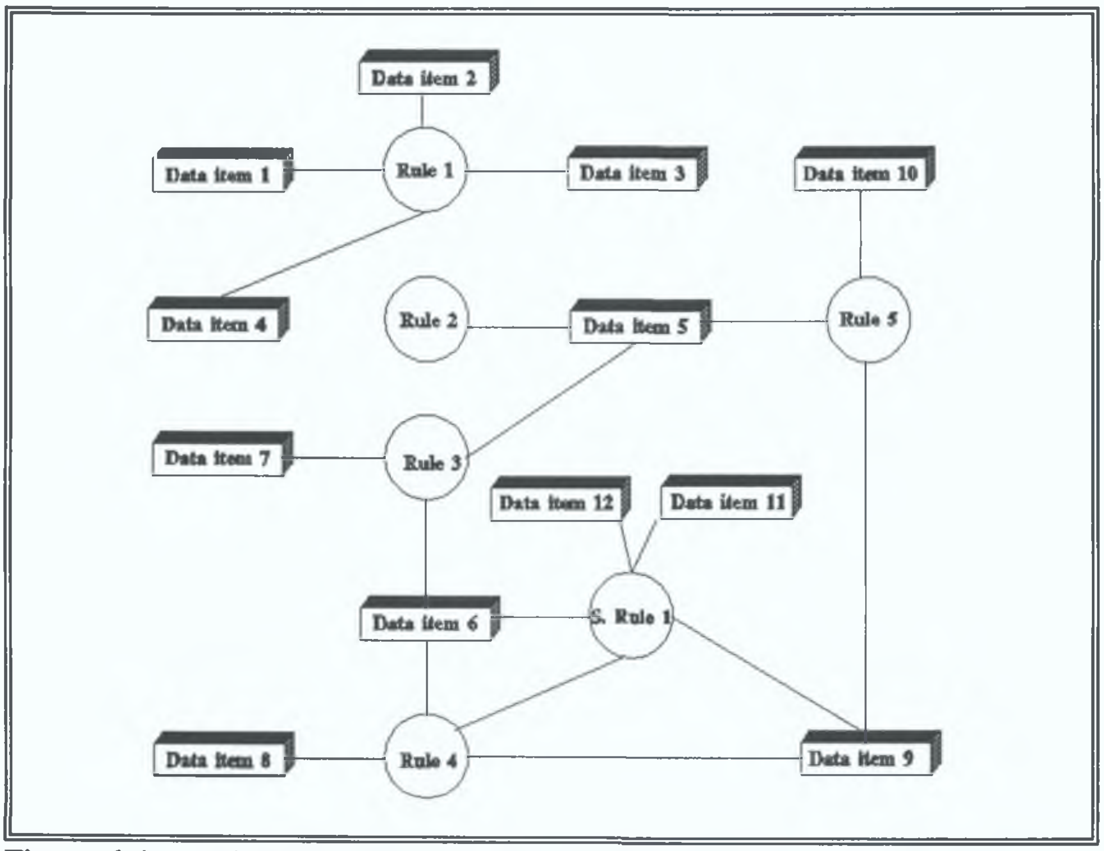


Figure 6-4 A rule-data relationship diagram for a batch program procedure

To provide a full description of the graph a symbol table is required. This table, shown in Table 6.6, is a key for the contents of the graphs, containing a description of all the symbols. As we can see in the diagram rules and data items are not always interconnected, rule 1 and its related data for example are not connected in any way to the rest of the code. The arcs (relationships) are nondirectional and only provide an indication that a relationship exists and not what form that relationship takes

Table 6.6

Symbol table for RD diagram

Symbol	Description	Symbol	Description
Data item 1	T_Rate	Data item 10	CurrT
Data item 2	Work_Rate	Data item 11	Curr2
Data item 3	Exch_Rate_PL	Data item 12	Curr3
Data item 4	Exch_Rate_BS	Rule 1	IF-ELSE
Data item 5	Num_Cnt	Rule 2	Assignment
Data item 6	Num_Days	Rule 3	Assignment
Data item 7	T_Aging_Date	Rule 4	IF-ELSE
Data item 8	Curr1	Rule 5	Assignment
Data item 9	Os_Bal_Curr	Sub-rule 1	IF-ELSE

This RD relationship diagram can also be used to calculate a measure of coupling that exists between rules and data (i e the connectivity of the graph). Using standard graph theory an adjacency matrix may be constructed from which we can determine which nodes are directly connected [Sedgewick 84] This information is usually presented in the form of an adjacency list Further levels of coupling (i e which nodes are indirectly connected via one other node) may be obtained by multiplying the matrix by itself. Within software it is believed that lower levels of coupling are more desirable as it facilitates the process of maintenance Similarly the lower the level coupling for a node then the easier it is to maintain that node, without diversely affecting other nodes

6.3.3.4 Data measures

Using the p-counts identified in Table 6.4, measures may be identified for use which relate to the storage of data within the DMS Examples of an initial set of possible composite measures relating specifically to the data are presented below.

A: Volume measures

It was indicated earlier that some of the criticality measures could be related to the size or volume of the system Thus the more data there is within a system the more likely that higher percentages will be associated with greater numbers of rules Equation 6 10 formulates the volume of data in the system as a direct measure of the amount of data within the DMS

$$V = CF \qquad \textbf{(Equation 6.10)}$$

This is a count of all the unique data items defined in the system This measure cannot be used alone, but it could be an indicator of system complexity when related to other p-counts, examples of which are given below

Equation 6 11 gives the ratios of records to volume, or put another way it measures the average number of data items per record, while Equation 6.12 gives the ratio of modules to volume, or a measure of the average number of data items per module

$$FV = \frac{V}{CR} \qquad \textbf{(Equation 6.11)}$$

$$MV = \frac{V}{CM} \qquad \textbf{(Equation 6.12)}$$

These last two measures provide more information relating to the distribution of data within the DMS and the modules Higher values of FV and MV imply a possible large set of unpartitioned data

6.3.3.5 Dynamic measures of rules firing

For dynamic measures to be calculated a tool must be created that is able to collect data while the application is running Most dynamic measures are used to determine the reliability of a system or to determine the test coverage The measures defined in this section are similar to dynamic measure currently in existence which relate to the level of test coverage The only difference is that coverage is related to rules and not statements The test coverage of the system can also be measured using the following general formula

$$Ratio = \frac{OT}{NO} \qquad \textbf{(Equation 6.13)}$$

Where *OT* is the number of objects tested, and *NO* is the number of objects in the system

Objects usually identified for structured code are BI (block of instructions), DDP (decision to decision path), and LCSAJ (Linear code sequence and jump), all of which may be applied on a small scale to rules within procedures However, to ensure that rules are being tested then we need to identify rule related objects. Below are two objects for use with RULER

RI Rule Instance:	The number of rules in the entire system A basic object that can be used to indicate whether or not all rules have been executed at least once
RB Rule Block:	The number of rules associated with a data item irrespective of where that rule is stored

Using these objects test cases can be defined for the system and measures of the rules fired noted Using the coverage ratio formula from Equation 6 13 we can determine the percentage coverage of rule blocks, or the percentage number of rules associated with a data item that were implemented In this case *NO* is calculated as the number of rules associated with a data item for all entities, and *OT* is number of rules associated with that data items that were activated in a test period

$$RBC_j = \left(\frac{OT}{NO} \right) * 100 \qquad \textbf{(Equation 6.14)}$$

Where:

$$OT = \sum_{i=1}^{n} FDT_{ij} + \sum_{i=1}^{m} FDF_{ij} + \sum_{i=1}^{p} FDR_{ij} + \sum_{i=1}^{q} FDM_{ij}$$

$$NO = \sum_{i=1}^{n} DT_{ij} + \sum_{i=1}^{m} DF_{ij} + \sum_{i=1}^{p} DR_{ij} + \sum_{i=1}^{q} DM_{ij}$$

To demonstrate the computation of RBC for a data item j the results of a test case have been presented in Table 6 7 The number of rules which reference the data items in all four entities are given along with the number of rules which were actually activated during the *test case alpha*

Table 6.7

Result of Test Case

Entity	**Number of rules a data item appears in: Dx_{ij}**	**Number of rules activated during test case**
Type	3	2
Field	10	7
Record	2	1
Module	30	10

Using this data we can see that the percentage coverage for a data item j is calculated below as 44 4%

$$RBC_j = \left(\frac{OT}{NO} \right) * 100 = \frac{20}{45} * 100 = 44.4\%$$

The percentage coverage of just one data item is not very helpful if there are thousands in the system, so a measure of coverage for rules activated relating to <u>all</u> data items in the system needs to be defined, shown in Equation 6 15 below Since each declared field is an entity, the complete list of unique data items is calculated using the number of field entities in the system

$$RBC = \frac{\sum_{j=1}^{m} RBC_j}{CF} * 100$$

6.3.4 Complete definition process

The final three steps are briefly described in this section As previously stated the primary aim of this research is define a set of measures for a rule based language The last steps in the process described in chapter 5 were included for the sake of completeness but are not entirely relevant to this thesis

Step 4 Define a model unifying these measures

Existing quality models are applicable to all measures defined so far As described earlier ISO9126 provides a set of top level attributes which are most desired in software The sub-attributes (or internal attributes) of these external attributes are related to issues such as complexity, which many of the measures defined provide data on Step 2 in this process indicated that many components are already measurable and that additional measures relating to rules and data were required, so these measures are additions to the tools available to assess the maintainability of software using this rule based language

Step 5 Perform measurement and validate data

Difficulties relating to the validation of measures have been addressed in section 5 5.1 It has also been stated that validation is a non-trivial task that typically evolves the presence of large volumes of data collected over a period of time. This research does not have access to the considerable amount of data required, so a less than rigorous validation is proposed

There have been three different methods employed which offer some indication of the validity of the measures proposed

A. External review

All measures proposed have been reviewed by other members of the metrics community, notably Richard Bache who has contributed to the development of the software static analyser QUALMS by introducing newly developed test coverage measures based on the program flowgraph structure [Bache 90] Richard has produced many papers relating to test coverage metrics including co-authored papers with Norman Fenton and is currently involved with SCOPE [SCOPE 90], an Esprit II project which is assessing the feasibility of a software quality assurance scheme in Europe

B. Publication

A paper based on the proposals for metrics for rule based systems was accepted and read at the IEEE Fifth International Software Engineering Knowledge Engineering conference in Capri, Italy [Doyle et al 92]

C. User observations

A range of statistical analysis of RULER modules has been published as part of an evaluation of case study performances in SCOPE [Neil et al 92], it was found that these traditional structural measures failed to identify critical parts of the system that the case study provider considered affected the maintainability of the software.

Based on these observations, the rules presented in this chapter have been defined.

D. Trend analysis

Since complete validation cannot be performed an initial set of trend analysis will be presented To collect the data for these analysis a prototype static analysis tool has been developed concurrently with this research Using this tool large volumes of module rules have been analysed and statistical results have been extracted to attempt to provide descriptive information and tentative correlations between p-counts Although further analysis is required to ensure that conclusions are valid these results will help set initial tolerance ranges for the measures defined. Both the description of the analysis tool and statistical analysis are presented in chapter 7

Step 6 Re-iterate / modify

Modifications to our model may be required based on historical data Correlations may not be made between measured values and software performance If such a case arises, modification or 'fine-tuning' of the measures may be required This will only be known when the process of validation is complete

6.4 Summary

This chapter provides a set of measures which attempt to capture relevant information regarding the maintainability of RULER Components for which measures already exist have been identified along with their related measures, while new primitive counts were defined for components for which no measures existed Using these p-counts, composite measures were formulated along with a rule-data relationship diagram from which most p-counts can be obtained The validation of these measures has still to be completed and this issue will be addressed further in the following chapter

7 Results and analysis

7.1 Introduction

In order to provide data from which analysis can be initiated, a static analysis tool has been developed. This tool has been used to collect data for applications developed using RULER. It also demonstrates the practicality of data collection within the RULER environment. This prototype tool, described in section 7.2, demonstrates the ease in which a complete tool could be developed. A modular development process has been used to allow changes to be made as data requirements are modified. Requirements for data collection based on defined p-counts have altered, and although alterations to the tool have not been completed, such changes should require little difficulty.

Using the data collected, analysis has been performed to further describe the relationships between data and rules within the system. Section 7.3 graphs this data and provides tentative conclusions and tolerance ranges for measures defined in chapter 6.

7.2 Tool development

R-DAT (Rule and DATa static analyser) is a prototype tool which calculates some of the measures described in chapter 6 and has been developed primarily to demonstrate the feasibility of producing rule and data related measures using a static code analyser, and to provide a comprehensive foundation which would facilitate further development

R-DAT was developed on a SUN 4 workstation and was written using C, YACC and LEX An initial problem exists in taking RULER code from the VAX and transferring it onto the SUN However, within the development environment of this tool direct connections between these two hosts exist

To maximise the flexibility of the tool, R-DAT was developed as a set of tools that incorporate the use of 'pipes' which serve as the connection between each tool. Figure 7-1 provides a general representation of R-DAT's functionality as described in section 7 2 1 and 7 2 2, while Figure 7-2 shows in detail the process of measurement

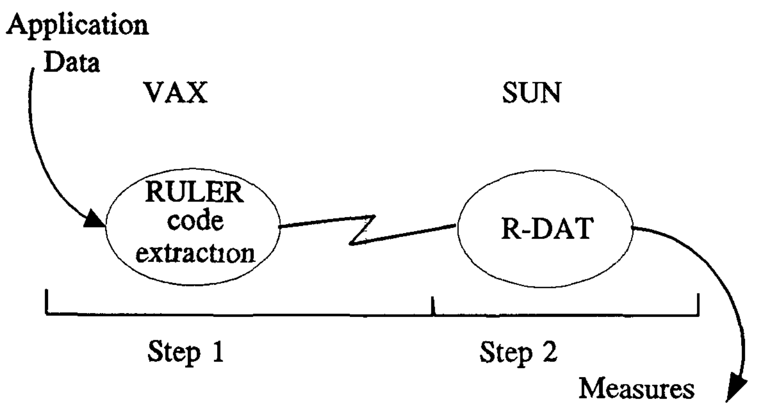


Figure 7-1 Overview of R-DAT and RULER integration

7.2.1 Initialising for R-DAT

A setup procedure is required before R-DAT can be used, consisting of the following two steps

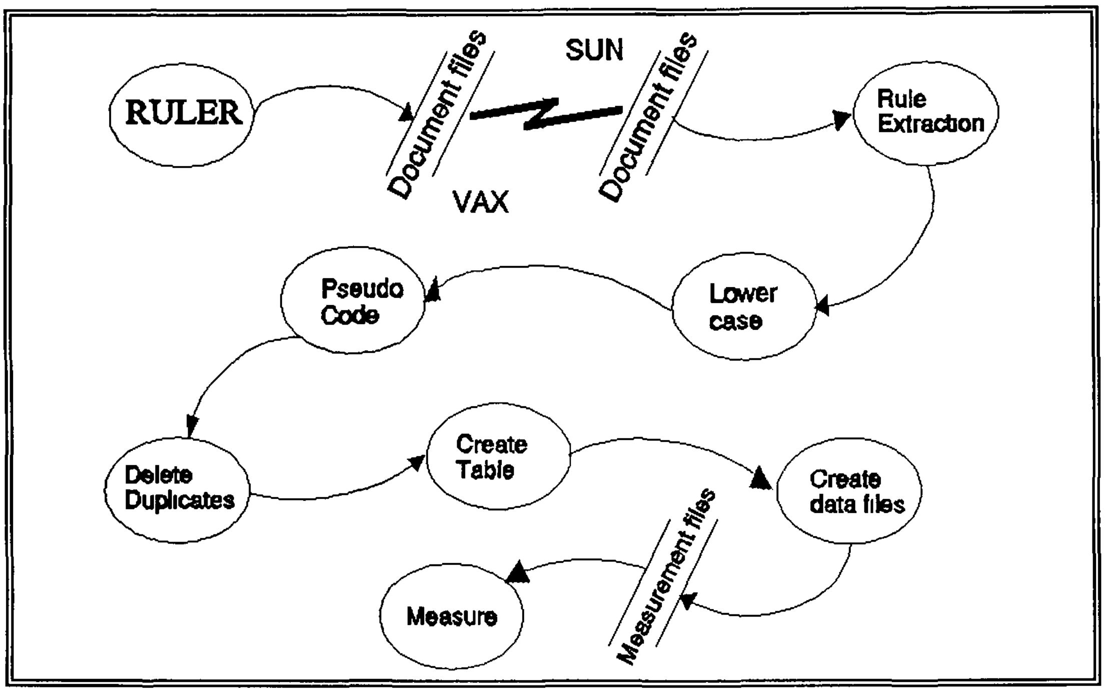


Figure 7-2 Detailed process of measurement using R-DAT

A. Code extraction

This function is provided by the RULER system All code within the modules can be extracted as part of a documentation process These files however contain other non-standard 'statements' which are part of the documentation text and are not module rules Similarly a documentation process is provided for the contents of the DMS Rules associated with types, fields and records are also extracted using the RULER documentation feature Four files are created, one for each of the four entities Each file contains a description of each instance of the entity along with the rules associated with that instance

B. Code transfer

The transferring of files between different systems is typically well documented and easily implemented In this case FTP provided the required facility

The result of these two steps is to provide four files on a SUN workstation which can be used as input to R-DAT

7.2.2 Implementing R-DAT

A. Rule extraction

This procedure removes all documentation information within the RULER files. Initially this was performed using editor macros, however this could be performed using a small program written in LEX This program has not as yet been developed, but would be a significant advantage when large volumes of code are to be processed

B. Lower case rules

This program was written using LEX and merely reduces all characters in a file to lower case The main reason for this is that RULER is not case sensitive, unlike Unix, so the work required in identifying recurring instances of the same token is reduced

C. Pseudo-code generator

This program parses RULER code and produces a simplified abstraction, or pseudo-code equivalent Using this new representation it is easier to calculate the required measures All reference to the assignment operations between data items is removed A possible extension of this program would be to take this pseudo-code and provide a graphical interface giving the Rule-Data relationship diagram representation of the code The form of this pseudo-code is given in Figure 7-3

D. Duplicate deletion

This program eliminates the duplicate data items within the same rule, as required by the definition of a rule in section 6 3 3.3 Data items between RULE and END-RULE blocks are sorted and duplicates eliminated

```
Module 1
RULE1
                Data item1
                Data item2
END-RULE1
RULE2
                Data item2
                Data item3
END-RULE2

Module 2
RULE1
                Data item4
END-RULE2

        etc
```

Figure 7-3 Pseudo-code structure produced for RULER code

E. Measurement table

This program produces a table in the format described in Example 7.1 Each row contains the details for each instance of entity so in the case of more than one rule, four bits of data are added to the row for every extra rule All subsequent measures are then performed based on the data in this table.

Module number	Rule no	No of data items	No of sub-rules	Level

Example 7.1 Table format for data collected from RULER code

F. Measurement files

Using the measurement table described more specific measures may be obtained. This is done by writing 'awk' programs which extract relevant data from this table and produce a file of more specific and condensed information Three such programs have been written which produce files of the following format

Module Number	Rule Number	Number of data items

Example 7.2 Output file - *filename* 1

Module Number	Number of rules in module	Number of data items	Number of sub-rules

Example 7.3 Output file - *filename* 2

Data item	Number of occurrences

Example 7.4 Output file - *filename* 3

G. Calculate measures

Using these final three files eight measures are calculated For additional measures to be provided suitable 'awk' programs need to be constructed Three such programs have been written which are described in Table 7 1

Table 7.1

Measures obtained using R-DAT

Program	Measures computed
Metrics1	Average rule complexity for all instances of an entity, based on the number of data items they manipulate.
	Maximum rule complexity for all instances of an entity, based on the number of data items they manipulate.
Metrics2	Average size of an entity instance, based on the number of rules in an instance of that entity.
	Maximum size of an entity instance, based on the number of rules in an instance of that entity.
	Average size of an entity instance, based on the number of data items in an instance of that entity.
	Maximum size of an entity instance, based on the number of data items in an instance of that entity.
Metrics3	Average level of criticality of data items based on the number of rules a data item is in.
	Maximum level of criticality of data items based on the number of rules a data items is in.

7.3 Analysis of data

The task of validation is by no means trivial. Many existing and popular measures are still applied to software when no significant analysis has been performed to demonstrate the relationship between the measure and the attribute it claims to assess. Many attribute / measure relationships have been accepted as intuitive, but are often based on little, if any, scientific evidence. The process of validation as previously mentioned is a long and detailed statistical task which has been covered in more detail by Norman Fenton [Fenton 91].

This section attempts to provide a description and characterisation of the data collected in order to allow tolerance ranges for measures to be derived and to identify trends within rules that could lead to further research into more composite measures (similar to DeMarco's BANG metrics) which would attempt to provide a single measure of system complexity, maintainability, or size

Two methods of data analysis have been performed The first is provided by the use of R-DAT from which we can obtain quantified results for our measures The second involves using statistical tools to provide more descriptive information regarding the relationships between p-counts defined in chapter 6 These measures have been performed for the module entity due to the restriction of available data.

Table 7.2

Results of module analysis using R-DAT

Measures evaluated	Results
Average number of data items per rule	007
Maximum number of data items per rule	057
Average number of rules per module	002
Maximum number of rules per module	018
Average number of data items per module	013
Maximum number of data items per module	058
Average number of rules per data item	002
Maximum number of rules per data item	022

7.3.1 Results using R-DAT

The use of R-DAT at this stage is more to provide the three files in examples 7 1 to 7 3 than to perform actual measurement However results have been extracted for the measures listed in Table 7 1 Typical use of this tool would not include the use of statistical analysis and so these results would be vital to the assessment of the software Both the measures below and statistical analysis in section 7 3 2 are based on the same three files Section 7 3 2 will however contain far more descriptive information from which ranges for measures can be determined Table 7 2 contains output from running R-DAT but the significance of the information will be discussed further in the next section

7.3.2 Graphing relationships

The three files created using R-DAT as described in section 7 2 (*filename* 1, *filename* 2 and *filename* 3) can be input to any standard statistical package from which descriptive information of the data can be obtained MINITAB was chosen for this task primarily because of its accessability A mimtab batch program was written which performed analysis of these three files These results were then incorporated into a Lotus 123 style spreadsheet and graphically presented In this section we examine those graphs and discuss the setting of initial ranges for a set of measures

Rules were observed in only three of the five module entity types, online, report and batch modules The graph in Figure 7-4 represents the histogram of data items to rules (ignoring sub-rules i e rules nested below the first level) within modules with maximum level indicators included to show the difference between the three module types If a rule contains many sub-rules then it is likely to contain a high number of data items which accounts to some extent for the larger values on this graph We can confirm this by looking at Figure 7-7, where we see that batch modules have more sub-rules than the other two This is compensated for in Figure 7-9 where rules and sub-rules are included

We can conclude however that 89% of rules have a complexity rating of less than or equal to 20 However 100% of rules in online and report modules have a maximum value of 6 Larger values are probably caused by the use of sub-rules within batch modules which will be considered later in this section So an initial range for rule complexity as defined in Equation 6 5, is 20 for batch modules and 6 for the other two

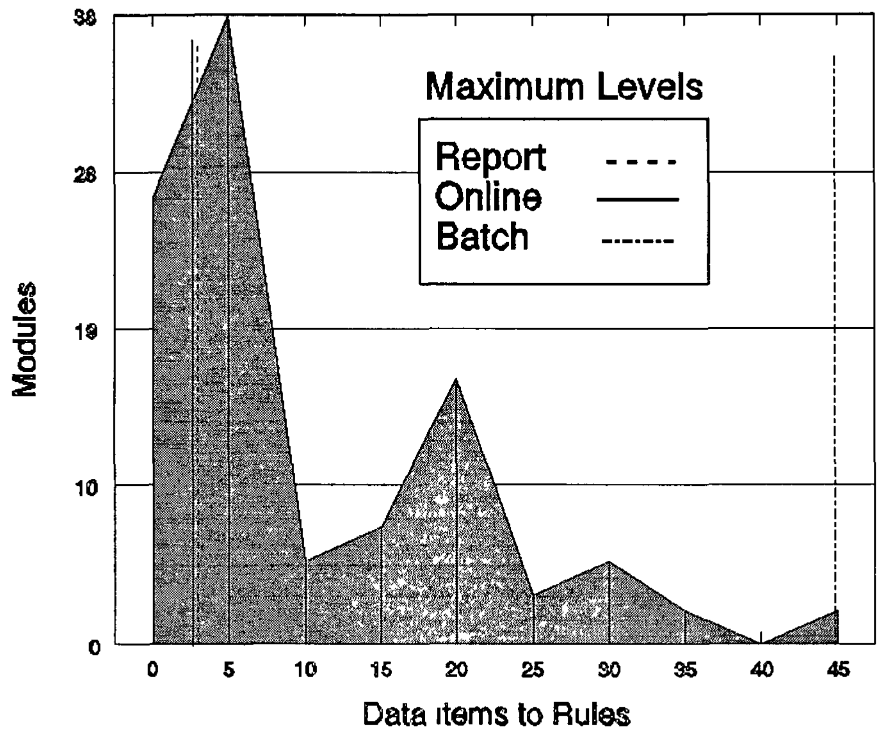


Figure 7-4 Rule complexity for all modules

Table 7.3

Basic statistics relating to Figure 7-4

Minimum	Maximum	Mean	Standard Dev.
0	47	10 1	10 74

The graph in Figure 7-5 presents the number of rules in a module to provide a measure of module 'size' 100% of both batch and online modules are below 4, and although the report modules, which tend to be larger are as high as 15, 92% of them have a size value of less than or equal to 4 It is only a small number of modules that exceed this value and as we shall see in Figure 7-6, report modules typically contain a higher number of data items So based on this data we can set an initial range for acceptable module size based on the number of rules as being between 0 and 4, which accounts for over 90% of all modules

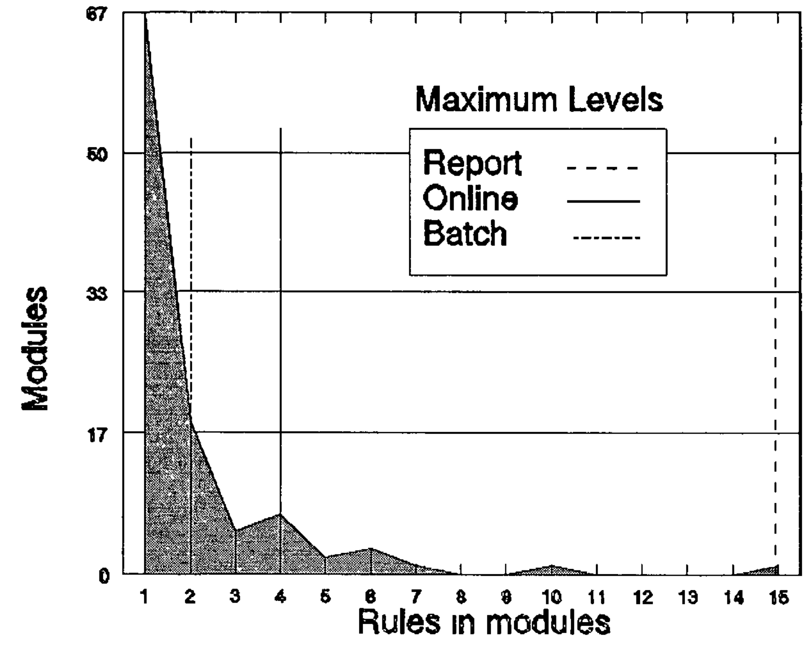


Figure 7-5 Size of a module instance based on the number of rules

Table 7.4

Basic statistics based on Figure 7-5

Minimum	Maximum	Mean	Standard Dev.
1	15	1 962	2 019

Fıgure 7-6 graphs the hıstogram of the number of data ıtems that appear ın a module, to ındıcate the sıze of the module The first poınt to notıce ıs that although batch modules contaın far less rules than report modules they manıpulate almost as much data, whıch explaıns why they contaın more data ıtems to rules, shown ın Fıgure 7-4 It would perhaps be more correct to ensure that sıze measures contaın ınformatıon about data ıtems <u>and</u> rules

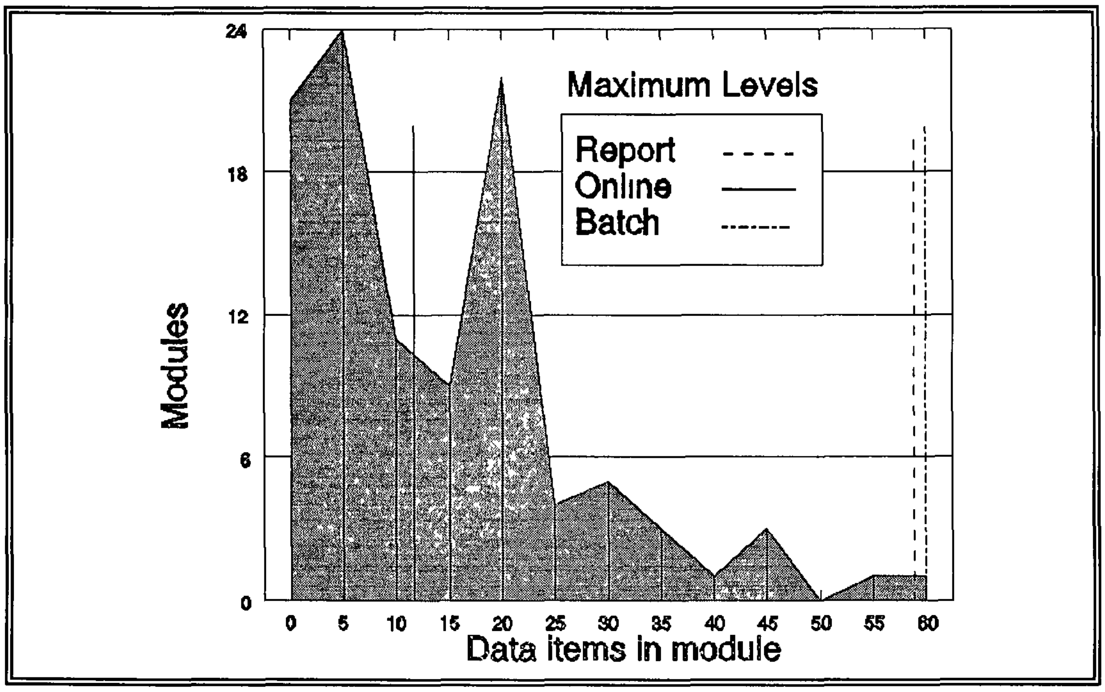


Figure 7-6 Module sıze based on the number of data ıtems ın a module

Table 7-5

Basıc statıstıcs relatıng to Fıgure 7-6

The number of data items that appear in a module for all modules			
Minimum	**Maximum**	**Mean**	**Standard Dev.**
1	58	14 27	12 91

The dıstrıbutıon of sub-rules to modules ıs well defined As ındıcated before, one of the reasons why there are more data ıtems to rules ın batch modules can be dırectly related to the fact that they contaın more sub-rules A good ınıtıal value for thıs measure ıs 3, whıch account for 92% of the modules Although hıgher values exıst, they are rather unusual and further ınvestıgatıon of these modules ıs requıred to examıne the reasons for these values

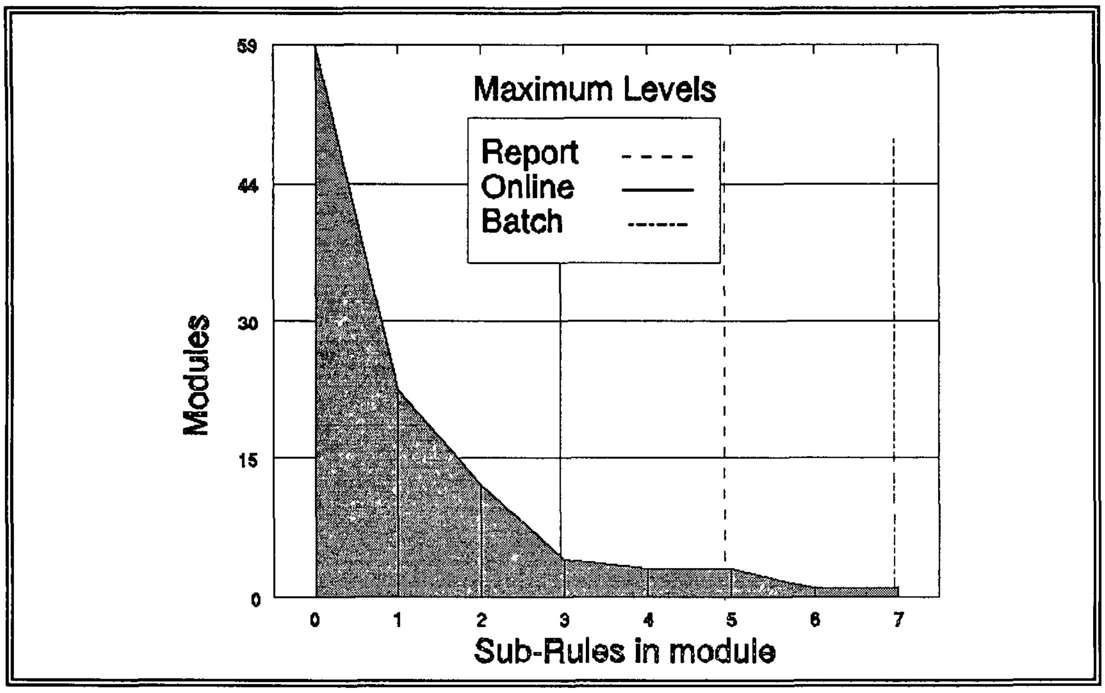


Figure 7-7 Module complexıty based on the number of sub-rules

Table 7.6

Basıc statıstıcs relatıng to Fıgure 7-7

Minimum	**Maximum**	**Mean**	**Standard Dev.**
0	7	0 933	1 463

The hıstogram ın Fıgure 7-8 takes ınto account both the rules and sub-rules as a measure of the sıze of a module Yet agaın we see that report modules contaın more rules than the others, although these exceptıons represent only a small percentage of the modules An adjusted range for the sıze of a module based on the total number of rules could be 8 whıch would account for 95% of the modules

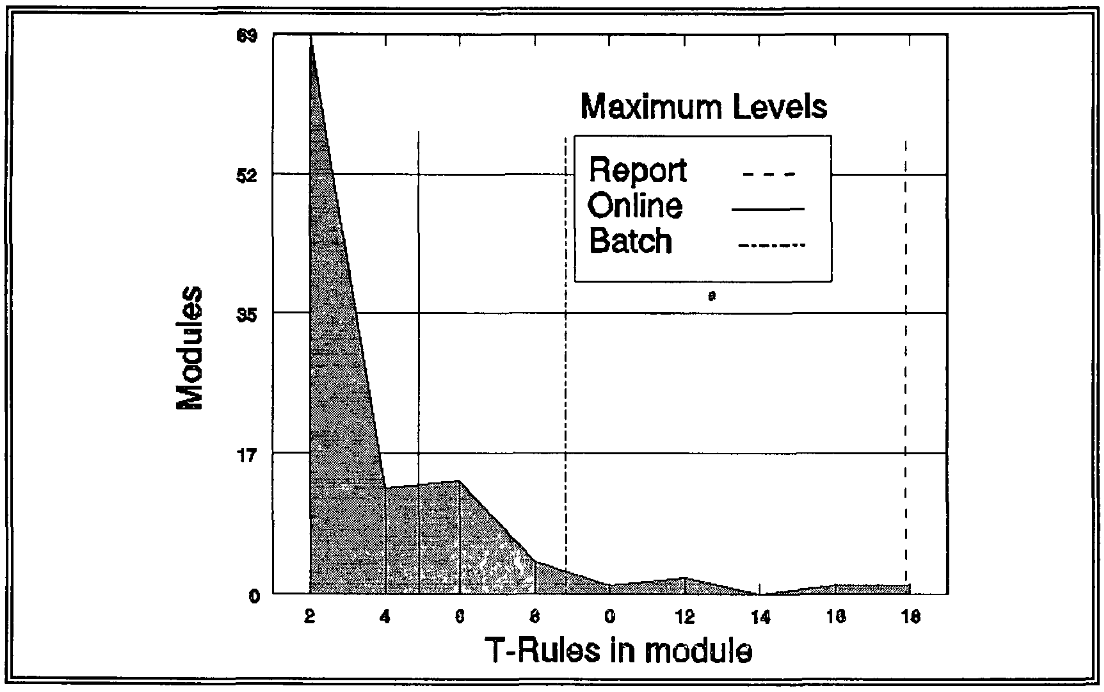


Figure 7-8 Module size based on the number of rules and sub-rules

Table 7.7

Basic statistics relating to Figure 7-8

Minimum	Maximum	Mean	Standard Dev.
1	18	2 895	2 984

The graph in Figure 7-9 is a corrected version of Figure 7-4, to include <u>all</u> rules in modules (rules and sub-rules) Note that there is a slight reduction in the larger values obtained in Figure 7-4 Those that still exist can be related to sequences of assignment statements within rules, which as described earlier are not counted as rules The complexity of rules in both online and report modules is not affected much by the inclusion of sub-rules which is to be expected based on Figure 7-7 It may be necessary to differentiate between these types of modules when setting tolerance ranges A range from 0 - 5 captures all values for online and report modules while 15 covers 86% of batch modules

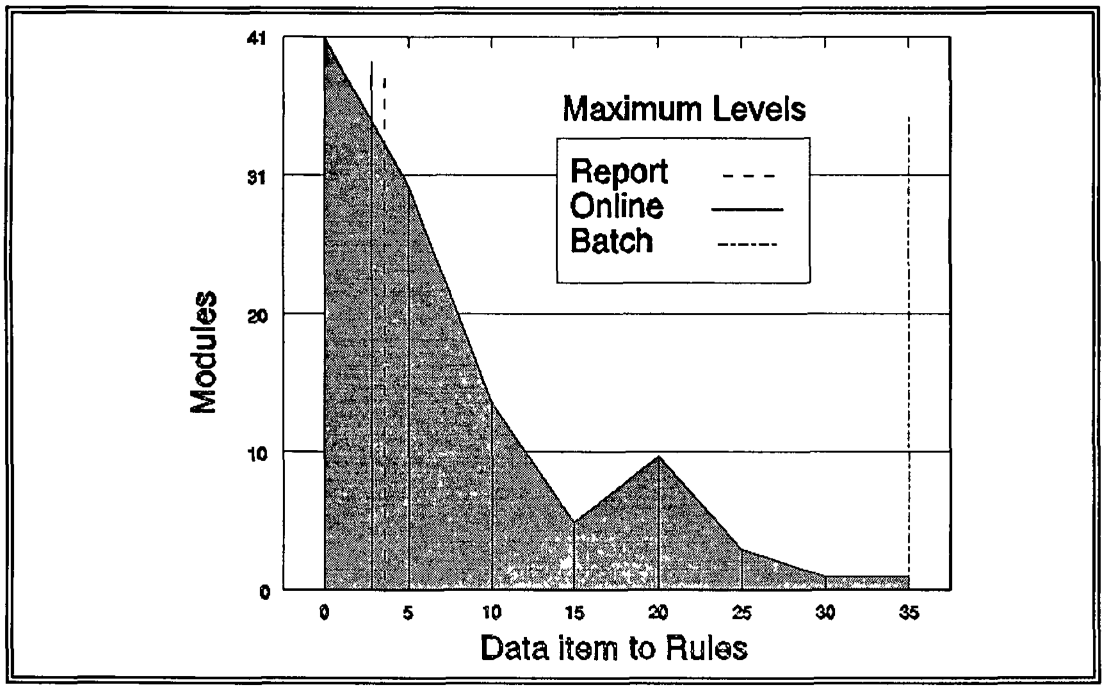


Figure 7-9 Rule complexity for all modules, adjusted for sub-rules

Table 7.8

Basic statistics relating to Figure 7-9

The number of data items to total rules per module			
Minimum	**Maximum**	**Mean**	**Standard Dev.**
1	37	7 061	7 669

Figure 7-10 gives an indication of the criticality of data by graphing the number of rules they appear in It is quite clear from the graph that at least 90% of data items appear in 4 or less rules However a maximum value of 22 exists, which would make the data item highly critical Such high values should be investigated to see if they are constants, or other relatively stable items of information If this is not the case the rules they are associated with should be extracted and examined to determine if such high levels are required This function could be an extension to R-DAT Suggested range for this measure is 0 - 4

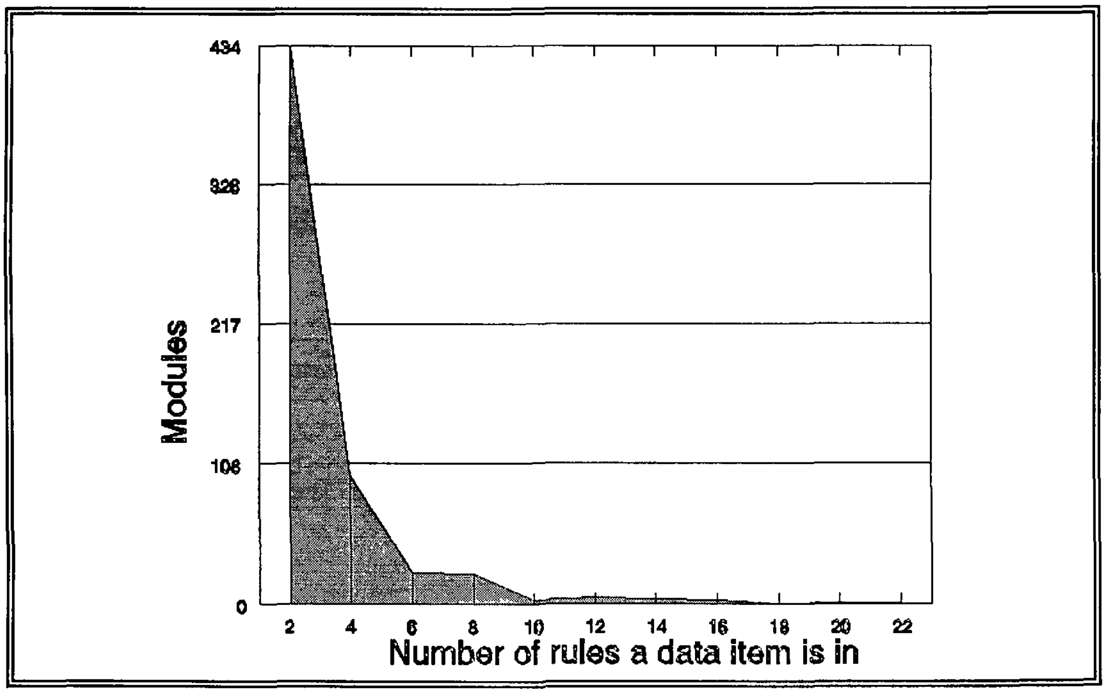


Figure 7-10 Data item criticality based on the number of rules it's in

Table 7-9

Basic statistics for Figure 7-10

Minimum	Maximum	Mean	Standard Dev.
1	21	2 505	7

7.4 Initial measurement ranges

The ranges for all of these measures have been defined for the module entity and new values are be required for the other three entities These values are provided as an initial set of ranges within which a high percentage of data falls, however no correlation between these values and the maintainability of systems that conform to them has been established This will require further data and analysis It is not recommended that these ranges should remain constant As more systems are analysed further evidence of their validity will be obtained and these values should be changed to reflect this

Measure	Minimum	Maximum
Rule complexity (excluding sub-rules)	0	8
Module size based on the number of rules	0	4
Module size based on the number of data items	0	20
Number of sub-rules in a module	0	3
Module size based on number of rules + sub-rules	0	8
Rule complexity (including sub-rules)	0	20
Data criticality	0	4

7.5 Summary

The prototype static analyser R-DAT has been developed to collect enough data to perform the initial analysis presented in this chapter. Although not complete, it demonstrates the ease with which data may be collected. This is one of the most important issues in software assessment - the production of objective and reproducible measures. The data R-DAT provided allowed the graphing of relationships between p-counts and measures to help provide tentative ranges which could be used as an indicator of the level of system complexity, maintainability, and size.

8 Conclusion

8.1 Conclusion

With the increased pressure on developers to produces systems of ever higher quality, measurement has provided a much needed tool with which to quantify attributes in quality modules. Measurement in software also aids in the prediction of resource usage, effort, and cost. Efforts have been made to ensure that when it is performed, measurement is done scientifically and with specific goals in mind (not just to please the Quality Controller). For project management to be effective information obtained from measurement should be accurate, objective and reliable.

Measurement has been used in software engineering to help assess and predict attributes such as complexity, useability and reliability for almost two decades and most measures defined were tailored for procedural languages such as Cobol, Pascal and Fortran. Many of these measures, still in use today, are not universally accepted with questions regarding how they relate to attributes still under debate. The use of Halstead's size measures relating to operands and operators is seen as a vital part of many prediction and assessment models, however the use of these vocabulary measures as a the basis for software science is rather controversial. Assuming that many existing measures do quantify important attributes of software, most have been developed for procedural development methods. Other development methods such as object-oriented, rule based and 4GL systems also require measurement for the same reasons as procedural systems, ie. greater control of system development can only be obtained when increased reliability in measurement is possible. Some implementation independent measures do exist, but are confined to cost and size estimations based on documentation produced early in the software life cycle

This research focuses on measures that can be applied to rule based systems, and provides a set of steps through which measures can be developed for other non-procedural development tools Using these steps a set of measures have been defined for a 4GL rule based language called RULER This set of six steps attempts to help in the identification of components of the software for which no measures exist As much re-use of existing measures as possible is encouraged to ensure that as much standardisation as possible is maintained within the science of software measurement

Using a set of identified primitive counts, composite measures were developed which attempt to formulate attributes such as rule complexity, data criticality, and entity size. All of these measures provide quantified values useable in the assessment of an applications maintainability These measures easily fit the quality model structures defined by Boehm and McCall

8.1.2 Measurement in RULER

The categorisation for which measures were defined are data criticality, rule complexity, descriptive measures, volume measures and test coverage Each of these categories contain a number of measures which are based on the defined set of p-counts To improve the objectivity and reusability of these measures a prototype static analysis tool R-DAT was developed which provided data for some of these measures to allow preliminary data analysis to be performed

R-DAT allows the collection of data for all except the fifth category of measures, which requires that p-counts be evaluated while the application is running No attempt was made to develop a dynamic code analyser to provide this data However little validation is require for test coverage Once suitable 'blocks' have been identified all that is required is the collection of data to ensure that as a high a percentage of the application has been executed as possible This measure based on the identification of a <u>new</u> block conforms to conventional test coverage estimation methods

Chapter 7 provides an analysis of the other four categories from which a set of measurement tolerance ranges were decided Ranges for measures are typically used to ensure that systems conform to acceptable levels of a particular attribute (or sub-attribute) The ranges in chapter 7 were primarily set to ensure that at least 90% of all data collected was within them It is not claimed that they are meaningful indicators of rule complexity and data criticality It is important to note that relationships between the attributes mentioned and the measurement ranges in chapter 7 have <u>not</u> been established This is not the aim of this thesis To set meaningful values for these ranges requires the analysis of larger volumes of data and also a rigorous validation to ensure that relationships between these measures and attributes exist As stated, the aim of this thesis is to demonstrate that measurement theory can be applied to rule-based systems and that measures for these systems can be developed

8.2 Future work

Tool development for measures is considered to be an important requirement for the assessment of attributes in an effective objective way It has been demonstrated that both objective and reproducible measurement may be performed for applications developed in RULER Existing measures have been applied using QUALMS, while R-DAT provided alternative measures Using both of these tools the feasibility of the assessment of the maintainability of RULER applications has been demonstrated However, R-DAT in its present state is incomplete Further modification to it's functionality is required An RD-diagram could be produced, which would provide a visual representation of individual modules, types, fields or records and allow direct access to the coding of rules and data items which contain *out of range* values

Another problem with R-DAT is that at present due to the use of 'awk' programs, decimal values are not possible in measurement results. Implementation of these programs in C would provide more suitable results. Greater functionality is also required to provide values for all measures defined in chapter 6.

With regard to measures defined, those that exist are only intended to be low level indicators for use in more complex measures. The definition of these measures will require a better understanding of what these measures actually assess. An approach similar to the one used by DeMarco in his definition of Design Weight is one possible method for establishing higher level measures such as system complexity, or system maintainability.

Now that measures for rule-based systems have been developed and implemented, other non-procedural development methods should be analysed with the idea of producing measures which quantify their unique properties. Logic programming and object oriented development methods are still badly supported by software measurement. A set of measures should be developed for Prolog and Lisp which may facilitate the standardisation of measures for all rule-based systems. The measures defined in chapter 6 should also be applied to Prolog to determine how applicable they are to different types of rule-based languages.

The decomposition of directed graphs into unique primes should be examined further to determine if unique decomposition of rules within rule-based systems is possible. Such a method could be the basis of a measurement framework for which parameters could be defined in a similar fashion to those measures described in section 4.4.4.

The existence of measures for any system is only useful if those measures are actually used. An article by Lieberherr [Lieberherr et al 89] proposed that one method for ensuring a good program was to encode a set of measures within the language itself. In this way we are guaranteed that a minimum level of assessment is performed. This idea could be incorporated into RULER to ensure that compılatıon ıs preceded by a set of measures whıch could provıde warnıngs to the programmer or even provıde suggested modıficatıons to the source

As development approaches branch out ınto more and more dıverse areas ıt ıs ımportant to contınually update our methods for controllıng theır development Thıs research has shown that such measurement ıs both feasıble <u>and</u> practıcal

References

References

[Albrecht 79] Albrecht A J , "Measuring application development productivity", Proceedings of IBM Applications Development Joint SHARE/GUIDE Symposium, Monterey, CA, 1979, 83-92

[Amble 87] Amble T, *Logic Programming and Knowledge Engineering* Addison-Wesley Publishers Ltd 1987

[Bache 90] Bache R , *Graph theory models of software*, PhD thesis, South Bank Polytechnic, London, 1990

[Basili et al 88] Basili VR, Rombach HD, "The TAME project Towards improvement-oriented software environments", IEEE Transactions on Software Engineering 14(6), 1988, 758-773

[Boehm 81] Boehm B W , *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, N J 1981

[Bohm et al 66] Bohm C, Jacopini G , "Flow diagrams, Turing machines and languages with only two formation rules", CACM 9(5), 1966, 366-71

[Chidamber et al 91] Chidamber S, Kemerer C , "Towards a metrics suite for Object Oriented Design" Proceedings of sixth ACM conference on OOPSLA, October 1991

[Conte et al 86] Conte SD, Dunsmore HE , Shen VY, *Software Engineering metrics and models*, Benjamin Cummins Publishing, inc 1986

[Curtis 80] Curtis B , "Measurement and experimentation in software engineering", Proc IEEE 68(9), 1980, 1144-1147

[DeMarco 88] DeMarco T , *Controlling Software Projects*, Englewood Cliffs, N J Prentice Hall, 1988

[Dobson 90] Dobson AJ , *An Introduction to Generalised Linear Models*, Chapman and Hall, London, 1990

[Doyle et al 92] Doyle P, Verbruggen R , "Applying metrics to rule-based systems", Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering, IEEE computer society press, June 1992, p123-130

[Draper et al 66] Draper N, Smith H, *Applied Regression Analysis*, Wiley, New York, 1966

[Fenton et al 86] Fenton NE, Whitty RW , "Axiomatic approach to software metrication through program decomposition", Computer Journal, 29(4), 1986,329-339

[Fenton 91] Fenton N E , *Software Metrics, A Rigorous Approach*, Chapman & Hall, London 1991

[Gibbins 88] Gibbins P, *Logic with Prolog* Oxford University Press, New York, 1988

[Gilb 87] Gilb T , *Principles of Software Engineering Management* , Addison Wesley, 1987

[Halstead 77] Halstead M H , *Elements of Software Science* New York Elsevier North-Holland 1977

[Hoaglin et al 83] Hoaglin D, Mosteller F, Tukey J, *Understanding Exploratory Data Analysis* Wiley, New York, 1983

[ISO9126] Software Product Quality Characteristics & Guidelines for their Use

[Kitchenham 87] Kitchenham B , "Towards a constructive qualiuty model", ICL Technical Journal, 2(4), 1987, 105-113

[Kowalski 74] Kowalski R, *Logic for problem solving* Elsevier North Holland, 1974

[Littlewood 88] Littlewood B ,"Forecasting software reliability", in *Software reliability, Modelling and Identification*, (Ed Bittanti S), Lecture Notes in Computer Science 341, Springer-Verlag, 1988, 141-209

[Lieberherr et al 89] Lieberherr K , Holland I , "Assuring Good Style for Object-Oriented Programs", IEEE Software, September 1989, 38-48

[Logiscope 88] Verilog S A, *Software development and testing using LOGISCOPE*, March 1990

[Martin 85] Martin J , *Fourth Generation Languages Vol I & II* Prentice Hall, Englewood Cliffs, N J 1985

[McCabe 76] McCabe TJ , "A complexity measure", IEEE Trans software Eng SE-2(4), 1976, 208-230

[McCall 77] McCall J A , *Factors in software quality Vol I,II,III*, US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055, 1977

[Moroney 50] Moroney M , *Facts from Figures* Penguin Books, 1950

[Neil et al 92] Neil M, Carson P, "DCU1 Case Study Data Analysis Report", SCOPE consortium, Esprit II, P2151, March 1992.

[Ratcliffe et al 90] Rathcliffe B, Rollo AL , "Adapting function point analysis to Jackson System Design", Software Engineering Journal, 5(1), 1990

[Ruler 87] Phimac Ltd , *RULER reference manual*, April 1987

[Sammet 69] Sammet F E , *Programming Languages History and Fundamentals* Prentice Hall, Englewood Cliffs N J 1969

[SCOPE 90] SCOPE consortium, *Technical Annexe*, Esprit II, P2151, July 1990

[Sedgewick 84] Sedgewick R, *Algorithms*, Addison-Wesley, 1984

[Seigel et al 88] Seigel S, Castellan N, *Nonparametric Statics for Behavioural Sciences* (2nd Edition), McGraw Hill, New York, 1988

[Sprent 89] Sprent P , *Applied Nonparametric Statistical Methods* Chapman and Hall, 1989

[Stroud 67] Stroud JM , "The Fine Structure of Psychological Time" Annuals of New York Academy of science 138,2 (1967) 623-631

[Tukey et al 77] Tukey J, Mosteller F , *Data analysis and regression* Addison-Wesley, 1977

[Verner et al 92] Verner J, Tate G , "A software size model" IEEE Transactions on Software Engineering, 18(4), April 1992, p265-278

[Walston et al 79] Walston CE, Felix CP , "A method of programming measurement and estimation", IBM Systems J, 16(1), 1979,54-73

[Warnier 76] Warnier JD , *Logical Construction of Programs*, 3rd Ed , trans B M Flanagan New York Van Nostrand Reinhold, 1976.

[Wilson 72] Wilson RI , "Introduction to Graph Theory", Academic Press, 1972

[Wilson et al 88] Wilson L, Leelasena L, "The QUALMS Program Documentation", Alvey Project SE/69, SBP/102, South Bank Polytechnic, London, 1988

[Yourdon et al 79] Yourdon E, Constantine LL, *Structured Design*, Prentice Hall, 1979.