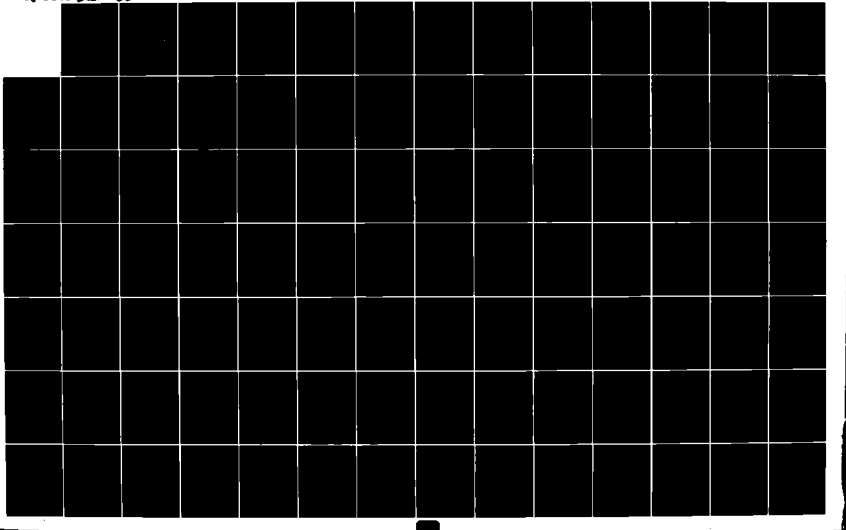
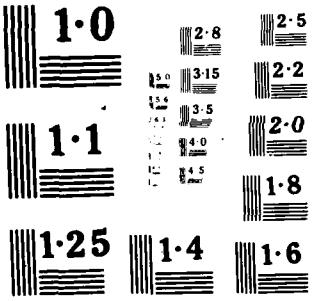
AD A158 184 PRACTICAL ASPECTS OF FUNCTIONAL TESTING TECHNIQUES(U) 1/2
LIVERPOOL UNIV (ENGLAND) DEPT OF STATISTICS AND
COMPUTATIONAL MATHEMATICS M A HENNELL ET AL. 31 JAN 85
UNCLASSIFIED DAJA37-81-C-0736 F/G 9/2 NL




1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
2·5
2·2
2·0
1·8
1·4
1·6

(2)

AD

Practical Aspects of Functional Testing Techniques

Final Technical Report
by

M. A. Hennell

M. U. Shaikh

S. Presland

P. Fairfield

Dept. of Statistics and Computational Mathematics
University of Liverpool
Liverpool L69 3BX
U.K.

United States Army
EUROPEAN RESEARCH OFFICE OF THE U. S. ARMY
London, England

Contract No. DAJA37-81-C-0736

Approved for Public Release; distribution unlimited.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER

AD-A158 184

4. TITLE (and Subtitle)

Practical Aspects of Functional Testing Techniques

5. TYPE OF REPORT & PERIOD COVERED

Final Technical Report
17 Sept 81-16 Jan. 85

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

M.A. Hennell, M.U. Shaikh,
S. Presland, P. Fairfield

8. CONTRACT OR GRANT NUMBER(s)

DAJA37-81-C-0736

9. PERFORMING ORGANIZATION NAME AND ADDRESS

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

61102A 1T161102BH57

11. CONTROLLING OFFICE NAME AND ADDRESS

USARDSG-UK
Box 65 FPO NY 09510-1500

12. REPORT DATE

31st January, 1985.

13. NUMBER OF PAGES

137

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Functional Testing, Requirements Analysis, Design Methods, Software Functions, Testing Methods.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document reports an investigation into the processes involved in functional testing. The objective was to determine criteria by means of which the process can be quantified and hence controlled.

The report demonstrates that insufficient knowledge about functionality exists for general metrics to be constructed. Indeed it clearly indicates that any method dependent on a knowledge of functionality is unlikely, except in special application areas, to be successful.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

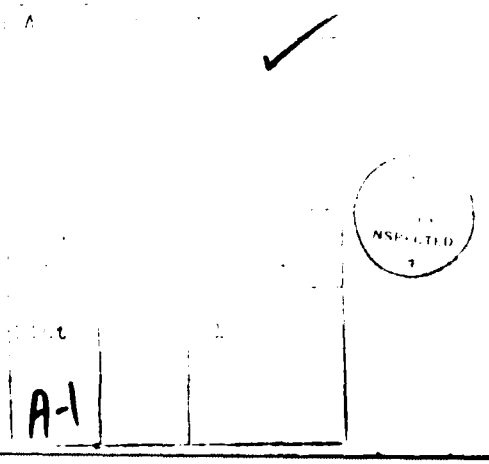
Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

The main problems lie in the lack of criteria for identifying functions in requirements documents.

CONTENTS



Abstract

This document reports an investigation into the processes involved in functional testing. The objective was to determine criteria by means of which the process can be quantified and hence controlled.

The report demonstrates that insufficient knowledge about functionality exists for general metrics to be constructed. Indeed it clearly indicates that any method dependant on a knowledge of functionality is unlikely, except in special application areas, to be succesful.

The main problems lie in the lack of criteria for identifying functions in requirements documents.

Keywords: Functional Testing, Requirements Analysis, Design Methods, Software Functions, Testing Methods

Investigation into Practical Aspects of Functional Testing Techniques

Introduction

The origins of Functional Testing are buried in the mists of time. Nowhere in the literature is the technique explicitly described. Nevertheless, the use of functions in describing software attributes is all pervasive, from widely used terms such as functional requirements through to functional maintenance.

Given that a software system does have a degree of functionality, then it is desirable to demonstrate that the functionality possessed is indeed the functionality required. It may even be argued that this is the only demonstration which is of interest to the "users". The concept is therefore of primary importance to the role of acceptance testing, in which the users or their agents assess the software.

In the context of this report the term Functional Testing will refer to the black-box approach in which the user perceived view of functionality will be explored.

This report will explore both the basic concepts involved together with the practical realities. Further, it will explore techniques for systemising Functional Testing. This will be accomplished by:

1) a very careful definition of terms,

2) the formation of a model of software development (a generalised life-cycle),

3) exploration of common 'design' methods to determine their impact both on the life-cycle model and more particularly the detection of functions,

4) a critical analysis of natural language to determine whether various paradyms for the detection of functionality actually work,

5) experimental analysis of the successes and failures of conventional functional testing techniques,

6) some proposals for systematic functional testing in limited areas.

It will be shown that attempts to establish clear-cut validation criteria expose severe weaknesses in all current software development processes. In particular it will be shown that there is no consensus as to what constitutes a software function. This is particularly interesting because implementation language designers have clearly expected such entities to be of high significance since they have in general provided powerful facilities (procedures and subroutines) which express such a concept. Indeed the higher the level of the language the more powerful these facilities become. Ultimately this culminates in the resurgence of functional programming languages.

Within a mathematical framework the concept of function is well understood and this understanding has spread into computing in many different ways, such as the development of functors, or the use of functions in VDM [45] and HOS [46]. All these developments however are bottom-up in the sense that if one has an idea of function then here is a method for expressing that idea. What is missing are techniques and criteria for identifying functions in the first place. This effect is even more traumatic in situations where the software developer might feel secure, for instance if a project is to implement a program to solve a system of mathematical equations (a nice set of non-linear partial differential equations for example). Here the functionality is clear and unambiguous. Nevertheless, the detailed examination of a program which offers a solution (or approximate solution) shows that there may be a lack of

distinct functionality in the code, and an attempt to test this code fares little better than for other classes of software. In this work the view is taken that the lack of clear-cut functuality in the software is not the result of carelessness, or incompetence in the programmers, but is a manifestation of a real intellectual problem.

One area which leads to confusion regarding the functions of software, is the relationship of the users view to the developers view. The distinction between these two views becomes blurred as the application becomes nearer to the software implementation environment. If the application area is totally divorced from the software implementation, e.g. a financial control package, then the software developer can easily distinguish between his or her view and that of the user. On the other hand, if the application area is software development (e.g. an operating system), then great confusion can arise as to which view is being examined, indeed very rarely are the different views even distinguished. The software developers cannot even see a problem because they relate easily to both views, after all, potentially they are both the user and developer. However a tester may have a more difficult problem. The tester must show that the user requirements are met, not that the software performs such tasks as perceived by the developers. The difference is subtle, but can have important consequences. Consider a user-friendly editor. To the developers user-friendly might mean having single character commands. These are user-friendly because they minimize effort. To other potential users, not involved in the development process a more meaningful command is more desirable, at least in the short term. This document emphasizes the difference between these two views.

The objectives of the project are to isolate where the problems of identifying functions lie, to attempt to improve the identification of these functions and finally to produce metrics which will enable a meaningful assessment of the success of the Functional Tests. This objective has been

gravely handicapped by the appalling use of terminology endemic in the literature. Terms are either not defined or are re-defined in ways which do not comply with common use.

This report considers the issues in the inverse order to which they were investigated in this research project. The approach actually taken, was to look at ways in which the current functional testing activities proceed. From an investigation of the achievements of these testing activities it was hoped that various defects would emerge. New techniques to rectify these defects could then be developed. In practice, defects were discovered but the problems of rectification proved troublesome. The reason is that the defects are easy enough to state in terms of the internal structure of the software, but no mechanism has been discovered to transmit this knowledge back to the class of function or function combinations which leads to this class of structure being tested.

The major cause of difficulty is that current design and analysis methods do not consider functions with sufficient care. Indeed, this report shows that almost nothing is known about the detection and specification of software functions.

The structure of the report consists of a definition of a generalised Life-cycle in Appendix A. Chapter 1 discusses the general principles and problems of functional testing. Chapter 2 investigates the current "design" methods for definitions of functionality and criteria for the detection of functions. Having failed to find anything satisfactory in the literature, the chapter then includes a new definition of function which collects together the bits and pieces discerned in the literature. This is followed by an attempt at general criteria for functional decomposition. Chapter 3 takes up the clues recorded in chapter 2, which state that functions may be associated with verbs. The structure of English is examined to see how the concept of function is expressed. It is shown that there are numerous ways of expressing functionality other than by verbs so that the simple paradym fails.

Chapter 4 looks at the experimental assessment of functional testing activities and points out some explanations for failure. Chapter 5 describes one general method which we have devised to improve functional testing. Chapter 6 is a general discussion of other observations made in the course of this work and attempts to summarise the findings of this project.

CHAPTER 1

FUNCTIONAL TESTING

1. General Issues

The principal objective in functional testing is to verify that a software system satisfies the requirements. This is achieved by constructing test data which in some way explores each of the possibly many functions which the system is required to perform.

Success in this activity does not imply that the system is wholly free from errors since the combinations of functions arising from actual use may cause interference and hence errors. Functional testing does not usually attempt to explore the (sometimes multitudinous) combinatorial problems. Furthermore design and implementation issues may cause subdivision of each function.

Such a testing strategy is extremely attractive, particularly to potential customers. If customers can see that the system really does perform these functions in the required way then their confidence in the usability of the system is greatly increased. Functional testing techniques therefore form the major techniques used in acceptance testing.

In general it is arguably an approximation to functional testing which most programmers attempt when no particular testing strategy is specified, i.e. the 'do the best you can' strategy. The test data sets in this case usually consist of an amalgam of:

the most obvious functional tests,

historically troublesome tests which appear relevant,

intuitively interesting data sets,

and, in desperation, sheer bulk.

Whether this strategy succeeds depends largely on the complexity of the new software, the experience of the implementors with similar systems, and luck.

The principal problem with functional testing is that it is extremely difficult to systemize. That is, one does not know what proportion of the functions one has successfully isolated, what the overlap between them is and to what extent they contribute to the implementation as a whole. For example, the principle functions of a system as viewed by the users may be implementable in a very small amount of code; the less obvious features contributing to the vast bulk of that code. This point will be explored in more detail later in this report. The problem of counting the functions and providing convincing evidence that all have been found is a convincing demonstration of the authors' thesis.

The difficulty stems in part from the lack of an adequate practical definition of the term 'function' as applied to software. This can be seen in an examination of modular programming where it is widely agreed that a module should implement a function or sequence of functions [33]. In practice this is replaced by *'modules shall contain more than 50 lines of code'*, or *'60 lines'* or perhaps *'as many lines as can be displayed on a screen'*, and so on [42]. The adoption of such simple paradyms in practical situations is indicative of a deep seated conceptual problem.

2. Principles

A partial solution to any fundamental problem is to proceed

b. System Development in Shared Data Environment: D2S2 [28]

This method defines functions as business processes which are carried out for the enterprise to perform and satisfy its objectives or goals. Functions are like business activities i.e. what the enterprise does; and are performed as a result of an event and change some input into an output. Function does not show "how" things are done or who does them. Success in identifying such functions clearly depends on knowledge of the business processes. Presumably innovative projects are precluded.

c. Structured Systems Analysis and Design: SASD [4,12,18,20]

" . . . . . , the hole left unplugged by any truly adequate definition of function is a structural defect in the theory through which camels and MACK trucks could rapidly pass".[20]

There is no explicit definition of the term function. From their literature it is clear that they are using this word informally and inconsistently, in general terms which are used in finance or business e.g. "accounts receivable", "accounts payable", etc.. They also use the term function for task, e.g. generating monthly payroll, performing a mathematical function (e.g. calculation of square root, logorithm and Basel function), which takes a value e.g. particular activation of SQRT takes on a value that is used in evaluation.

A function is described by an imperative sentence, ideally having an active verb such as produce, compute, verify, delete, extract, save, store etc., followed by a single, i.e. non plural, object clause e.g.

Display - overdrawn
Extract - weekly income
Enter - new books purchased

"Where did knowledge of the properties of the functions come from?"

It is in the attempt to check that the stated functionality is correct that the unsatisfactory knowledge of the functionality comes to light.

1. Definition of Function

The first step is to abstract the definitions of function from the methods and then compare and contrast the differing viewpoints. In the following, for each of a number of methods, explicit definitions will be reported and definitions of other terms which probably refer to function will be discussed.

a. JSD [21]

This method defines function as an action or set of actions performed by the system and resulting in the production of output. The term action is defined as an event in which one or more entities participate by performing or suffering the action. The term event is not defined. It is claimed that a model of the real world can provide a conceptual as well as a computational basis on which various functions and an information model can be built. The JSD model makes an abstract description of reality and determines how abstract descriptions can be realised in working computer models. The model defines a set of words to be used in functional specifications. From this set (dictionary) of words functions can be specified. Any function which appears in that dictionary is then specified. The result is that the system's functions are expressed in words whose meaning depends upon the users view of reality. It does not state how the functions may be derived other than to suggest that functions might be related to verbs.

CHAPTER 2

THE ROLE OF FUNCTIONS IN DESIGN METHODS

In this chapter, the concept of function is explored by means of some of the established design methods. The term design referred to here is used in the coarse sense in which such methods are described in the literature. The choice of methods covers the more widely used methods and a smaller number of the more specialised methods. Most of the methods referred to cover a much wider span of the life-cycle than purely the design phase. Indeed many claim to cover the whole of the life cycle. Their great value is that they have been widely used and developed over many years. Considerable knowledge and experience have been built into them and this should not be lightly discarded.

The objective of this chapter is to discover how these methods direct the discovery and use of functions. Firstly the definitions of function will be investigated, then directions for producing functional specifications and performing functional decompositions will be analysed.

The approach taken in this chapter is the highly critical approach of a validator who must ask questions, such as:

"Is this really the function that is intended?"

"Are there any other related activities missing?"

"What evidence is there that the functions stated are really wanted?"

"Are there any wanted functions which are missing?"

constraints introduced within the design phase affect those user perceived functions. This effect manifests itself for each example as follows. For the numerical problem we are unaware of the class of function for which the program will work. For the compiler, the class of wrong programs which that compiler can successfully process is not known.

In this chapter a number of well known application areas have been examined and it has been demonstrated that even within these areas the identification, characterisation and testing of functions is not straightforward. It is clear that there are areas where the functionality can be established and other areas where it cannot. There are no criteria to help identify in which class an arbitrary problem lies.

programming may be banned with more than a certain number of users on-line.

The interesting question which arises from this analysis is how a particular requirement can be characterised when it does have a distinct functionality. It is desirable to be able to determine the situation where functionality is uniquely derivable from the requirements. One such criterion might be the situation where no input data is specified and the output is loosely specified. An example would be the solution of a set of mathematical equations. The functional decomposition could proceed top-down purely on basis of mathematically determined functions. The output would be the computed solution. Provided only that the solution method chosen satisfied the condition that the corresponding algorithms were completely understood then this decomposition can be (and has been) succesful. In such a situation the inputs (being dependent on the selected algorithms) and their ordering is determined by the nature of the decomposition.

At the opposite extremum there is the situation where a given set of inputs are to be used to produce a defined set of outputs. The global transformation which yields the required outputs from the set of inputs is not known and does not need to be known. The Jackson method accomplishes this by, first deriving the program structure from the structure of the inputs and outputs, and then produces the computational processes as a set of actions, each of which is a basic system function. The user perception of function is never considered.

From these arguments it may be seen that for any given problem, the knowledge of functionality may vary from the case where only the functionality is known to the case where it is wholly unknown. Worse still, it may happen that initially the functionality is known but the introduction of algorithms prevents the understanding of functionality proceeding through the design stage. What we have is a knowledge of the user perceived functionality without the additional knowledge of how

constructs. Neither is there any concept of completeness or convergence. The difficulties here are compounded by error states, error recovery and special cases. In our example of a compiler we would also have to add data consisting of all examples of wrong programs to the input data.

Clues for input data classification must be obtained from knowledge about the software implementation. Organising inputs according to some size criterion is a technique derived from knowledge that the software under test has internal structure dependant on size, not just tables, etc., but also loops and choices. So far the discussion has considered software which has one discernable (albeit complex) function. Consider next, an equally familiar problem which has a much more diverse functionality, namely an operating system. Firstly take a single user system. The Global Function is to service a set of distinct subfunctions, hiding system structure from the user and simplifying the users tasks. The set of functions could be

editing a file.

running a program.

creating or deleting files.

sending messages to another machine.

Yet it may be possible to perform all these functions whilst editing a file (even editing a second file). That is, these functions may be performed separately or as further subfunctions. The subfunction role may be slightly different from the primary function role, e.g. deleting a file as a subfunction may be additionally constrained by not permitting the deletion of the file currently being edited. The functionality is therefore dependant on the context. When next a multi-user environment is considered the problem becomes much more complex. The context may in fact enforce the abandonment (temporarily) of certain functions, e.g. interactive

introduced in order to permit communication between the fragments. The nature and type of both the housekeeping and method of communication will depend strongly on the properties of the functions and their environment. The inverse, however, is not necessarily true, that is, if a number of functions communicate via some housekeeping functions, it does not necessarily mean that they therefore constitute fragments of a more complex function. The techniques of communication may be the same in both cases.

4. Testing Functions.

The grossest descriptions of software functionality are embodied in the Global User Perceived functions. Each of these functions expresses a relationship between some class of input and corresponding output. The relationship will not be unique, indeed it may be a complex relationship. For example consider a compiler again. At the grossest level, submitting a source code program which is successfully translated into machine code which, on execution, delivers the expected result, can be said to be a succesful demonstration of the functionality of a compiler. The problem is that this demonstration does not imply that all source programs will be succesfully translated.

For the compiler, it might be said that the Basic User Perceived functions are that every conceivable source program should translate successfully. Since generating every example of a user program is not a realistic proposition, it is necessary to decompose the source code programmes into classes. This implies that the functionality of the compiler is to be decomposed e.g. into classes of size, starting with the smallest possible program, then by adding the constructs systematically to obtain the largest (in which each construct is repeated up to a given number of times). Note, however, that the decomposition notion is actually derived from the structure of the data, not from any explicit decomposition of the Global Function. There is no way to decompose the notion of compiling for all programs into compiling programs according to size or

example the syntax charts do not show the relationship to the lexical analysis. The criteria for taking one flow in the graph as opposed to another are not necessarily simple, one may need a look-ahead facility.

The point of the foregoing analysis is to show that in general, even for the best understood problems, techniques for design are not well developed. Worse still, even when functionality is clear at a global level, if may be totally unusable at a lower level. This problem has been well known in the field of numerical software. Here one starts with a well-defined, completely specified problem. The specification is a set of mathematical equations and boundary conditions which ensure unique solutions. The problem is that in transforming the problem, firstly to one for which a numerical solution can be found (a mathematical model) and, secondly introducing numerical algorithms (design) for solving the equations, knowledge of the functionality is lost. This is because particular properties of the equations require special attention, and the corresponding treatment of these properties requires the introduction of special functions (algorithms). The result is often a hotch-potch of algorithms tenously held together. That the program will solve particular problems is established experimentally by testing and it is then inferred that the program will solve similar problems. Even the definition and meaning of the word 'similar' is not clear. At the present time, techniques for determining the class of problems for which the programs produce valid solutions are not known [43].

Returning to the development of the compiler, the concept of a lexical analyser component is well understood and accepted. If the language design is such that a degree of parsing must be accomplished before all lexemes can be identified uniquely then the lexical analysis function will become distributed. In more complex systems, this problem of function fragmentation and distribution can become extreme. Whenever a function suffers from this effect, special housekeeping functions must be

between the functions are now missing. The essential detail is a knowledge of the classification of languages which relate to the techniques which can be used in compiler construction. An experienced compiler writer may have little difficulty but an inexperienced analyst would almost certainly guess wrongly.

Taking an alternative viewpoint we can say that the structure of the compiler is almost wholly determined by the structure of the source and target languages. This suggests the use of the Jackson method. Unfortunately, here too things are not satisfactory. Firstly the Jackson method requires the structure of the data to have rather simple properties. Thus recursive definitions cannot be handled and these are widely used in language definition. If one takes the view that all recursive structures are actually implemented as finite trees, the number of structure clashes (see [21]) will probably be overwhelming. Nethertheless if these clashes are examined closely it will be seen that many can be resolved by the classic Jackson solutions of intermediate files. The solution is then equivalent to the standard multi-pass practices familiar to compiler writers. With the Jackson method, however, the number of intermediate files (or equivalently, program inversions) is likely to render the method unusuable at present.

The third attempt is to try a data flow model. Here there are no rules and no methods. The input entities can be viewed at any level e.g. characters, lexemes, statements, modules, etc.. The relationships between the entities are closely defined but how is this translated into a data flow diagram. Examining language definition documents one may see a number of graphical representations of the syntax e.g. Pascal. These clearly can be used to design the early stages of a compiler. The further problems of semantic analysis and code generation are missing but presumably could be added. Missing in turn is the large amount of housekeeping to handle names, modes, checking, block structure, etc., but this could possibly be added as a second stage. What is missing is a set of rules, criteria or methods which will enable the additional components to be added. For

If then Functional Testing is seen as the process of executing user perceived functions to the satisfaction of the users or their agents then the goals can be classified along the following lines:

i) Execute every Basic User Perceived function.

ii) Execute all combinations of up to n Basic User Perceived functions.

iii) Execute all primary (high level) functions.

In order for such goals to be meaningful it is necessary in cases i) and ii) to be able to identify each Basic User Perceived Function. This is currently beyond technolgy. In case iii), the goal is only meaningful if the relationships between functions are clear and the dependant functions can be identified. There are an enormous number of papers and books which refer to functional decomposition but, as shown later, none of these actually give the rules which enables the task. That software developers actually succeed, at least to some extent, in this task is a tribute to their skills.

3. Analysis

In the search for functionality consider the following well-understood problem."A compiler is to be written for the language X for a target machine Y, where X is defined by a language definition document and the machine code of Y is defined in the hardware manual". The global functionality of the compiler is immediately clear, it is the transformation from X to Y. Morever we can now consider a second level decomposition of this function too: lexical analyser, parser, code generator. Unfortunately this second level decomposition may not be valid, for instance the lexical analysis may not be unique without a degree of parsing, e.g. if keywords are not reserved we may need a multi-pass compiler. The criteria which will inform us of the necessary extent of this interaction

unfortunately this may be extremely difficult. Design methods themselves are not designed to make this possible.

At the completion of the implementation phase the software will be composed of the Basic System functions. These Basic System functions can be classified as either housekeeping functions or implementations of algorithms relevant to the application area of the software. Housekeeping functions are those which are introduced post requirements analysis in order to be able to implement the required system on a computer, examples are functions to manipulate data structures, deal with errors, etc.. The proportion of these Basic System functions which are purely due to housekeeping can be of the order of 50% in typical systems. This point will be discussed later in Chapter 4.

Some conclusions can also be drawn from the previous discussion. Firstly, executing every Basic User Perceived function in a test need not necessarily result in every line of implemented code being executed. For example, housekeeping functions may not be invoked. Hence even the simplest possible check on test completeness is denied. Also the converse is not true even under the assumption of system completeness. That is, if the system is complete in that every user perceived function is implemented, then exercising every line of executable code does not guarantee exercising every user perceived function. This is because each Basic User Perceived function is implemented as a combination of basic system functions and this combination can be arbitrarily complex. It is this factor which prevents structural tests from being optimal for acceptance testing.

Exercising every line of code will, however, cause every basic system function to be exercised, providing only that there is no optimisation i.e. function overlap or redundancy. This can be seen as follows. Each line of code performs some part of a specific system function, hence exercising every line of code ensures that every part of each system function (and hence each function) is executed.

searching a table,

clearing a buffer,

queueing an interrupt.

The system design in turn considers the mapping of the system functions onto the computer hardware. This process may lead to restrictions from hardware constraints being imposed on the valid input data space. In turn, the transformation of the design into code can lead to further restrictions being imposed. This process can be seen as follows. Consider a single integer input problem for which the valid input data space from the requirements is $(-\infty, +\infty)$. The restriction imposed by a particular machine reduces this space to (-M, +M) where M is the largest integer representable in the machine. However, if the program implementation is taken into account where the program is:-

```
READ x

Y = X * X

WRITE Y
```

then the valid input data space reduces to $(-\sqrt{M}, +\sqrt{M})$. The full input data space of (-M, M) can of course be restored by testing the value of X and taking special action such as the use of multi-length arithmetic. This structure would, however, not be visible to a tester using functional testing techniques.

In a multi-dimensional input data space for a complicated algorithm with many inputs, considerations of this type may lead to a fragmented valid input data space where, (possibly) whole domains have been excluded and others extensively subdivided into sub-domains due to additional fine structure. It would be convenient if it were possible to relate this fine structure back to the user perceived functions, but

In general it is probably not sensible or even possible to test these Basic User Perceived functions individually so that they must be tested in various combinations. This is because there would need to be as many runs of the software system as there are functions. However as a comprehensive test even this still leaves much to be desired as will be discussed later.

From the requirements it is possible (in principle at least) to determine an input data space. This is the space determined by all the permitted inputs and their constraints. This space is then partitioned into various domains. All input values in a given domain require the same sequence of Basic User Perceived functions to be performed. Different domains may, however, require the same sequence of functions to be performed. The determination of these domains and their corresponding functions can, for certain classes of problems, be accomplished by Case Analysis as in [34].

The next step in the software cycle after completion of the requirements is the production of the system specification. This is also developed in layers of abstraction, presumably, but not necessarily, different to those used for the requirements. In this case the User Perceived functions are mapped onto the system functions. For example, in an aeroplane seat reservation system the user perceived functions might be:

an availability enquiry,

a purchase option,

a reservation,

a cancellation option.

The system functions might be:

creating a file,

systematically. In an advanced software implementation environment a 'requirements definition' document should exist (see Appendix A for definitions). This document is constructed by both users and analysts working together (hopefully) and summarises at least a global view of the functions which the system is to implement. More precisely it contains at least the Global User Perceived Functions: these are the principal functions which the user (guided by the implementor) conceives that the system should perform.

Difficulties may arise from grossness in the requirements, i.e. the level of detail may be insufficient to be able to isolate the fine structure which is needed for a comprehensive test. For instance, a single global function may require slightly different actions to be taken within the system depending on some minor change in an input. In an invoicing system a particular customer account may have to be processed slightly differently from the mass because the customer requires special transportation facilities. A global view of the invoicing system may not be sufficient to display this possibility.

In principle the more detailed the requirements definition, the more useful it is likely to be to the tester. There is, however, no reason why the requirements should be expressed in terms of functions, so that in extreme cases the requirements can be useless to the process of functional testing. This can arise for instance with the use of the Jackson design method [21] in which a purely data view of the system is taken. When the requirements are expressed in terms of functions, then it is possible (in principle) to expand the detail until the requirements are expressed in terms of Basic User Perceived functions. These Basic User Perceived functions are those in which all other functions can be expressed. Termination criteria for this process are difficult to specify, the point is that there should be no further fine structure as perceived by the users. One task of functional testing is to take each of these Basic User Perceived functions and supply test data to exercise them.

Delete - names of students who
have left the university
Compute - overtime

Unless the text is written this way in the first place this is not helpful.

Function is expressed like an order, in a way that it can easily be understood, and hence can be performed in an easy way. However there are no explicit criteria to select a function.

d. Nijessens Information Analysis Method: NIAM [14]

Here function means the capability to transform information flows, such capability is often represented by a verb. The function may be:

i) Formalised i.e. whose performance is known, called a formalised function.
ii) Unformalised i.e. functions for which we do not know the in- and outgoing information flows, and their mechanism for achieving the transformation is not known.

Only formalised functions can be a part of an automated information system in the NIAM method.

In an existing system an inventory of all the functions (represented by verbs) which the information system is expected to support is made. These functions are decomposed into subfunctions to a level of detail where information flow and transformation achieved by the function become clear.

It is claimed that an information flow (i.e. stream of messages which represents a communication between two certain objects such as users) always has a function as its origin or destination. Information flow gives an

information structured diagram which shows functions and constraints formally.

No criteria for detecting functions, other than by identifying verbs are given.

e. Sysdoc [27]

The term function is used as a business function or mathematical function; and is primarily described by its processing rules in a high level language. This is similar to b.

f. SADT [11]

No clear definition of function is available from their literature. Probably they use function in terms of activities which are represented by verbs in natural language. No criteria for detecting activities are given.

The identification of the main activity, which is further decomposed at some level, depends entirely on the intuition of the analyst/designer. Functions are diagramatically represented by boxes in the activity diagram. These boxes represent collections of related activities. A box may perform different parts of its function under various (different) circumstances, with different combinations of its inputs and controls, giving distinct outputs as a result of different activations.

In the data diagram which presents a different view of the same subject, its decomposition is based on classes of data not on classes of function. In this type of diagram, activities i.e. functions expressed by verbs are represented by arrows entering or coming out of the boxes. There is no mention of any method to select a sub-function such as might be needed if one arrow splits and terminates

at two boxes.

g. SDL [47]

The term function is not explicitly determined by SDL. However, in SDL, the term function mainly deals with tasks or jobs concerned with switching systems where the action of a function is to alter the current values of local variables of a process such as

- call processing and
  (e.g. call handling, routing, signal sending, signal recognition, metering, call charging, etc.),
- maintenance and fault treatment
  (e.g. alarms, automatic-fault clearing, configuration control, routine test, etc.),
- system control
  (e.g. overload, control, modification and extension procedures, etc.).

SDL is also applicable to a range of other systems.

From this search of some of the major design methods it can be seen that considerable problems exist with current definitions of function and criteria for detecting such functions.

The IEEE [52] defines function in two parts, firstly as

"the specific purpose of an entity or its characteristic action".

The other part of the definition refers to the specific use of the term in programming languages. Note that entity is not defined and nor is action; these therefore have their wider meanings. The word purpose is ambiguous being either an aim, or an effect. The IEEE definition is essentially useless.

2. Functional Specification

Even though methods may not be explicit with a definition of a function, it may nevertheless be possible to deduce the characteristic properties of functions from the way in which the term is used. The term functional specification is widely used and hence this aspect is explored in the hope that more light will be shed on the problem.

The IEEE defines a functional specification as

> "a specification that defines the functions that a system or system component must perform".

Function has its previously mentioned definition and perform is not defined. This definition of functional specification is therefore meaningless.

a. JSD uses the term functional specifications in the general sense of specification. There is no clear difference between the two terms. The abstract description of the real world (term not defined explicitly), which is a partial description, and its realisation both provide the context for functional specifications. The basis for:

   deriving an abstract description,
   developing the dictionary of words,
   knowing the meaning of each word in the real world,

   is not stated. However functionality is not particularly emphasized, and furthermore they believe that a system design methodology based on function will always produce systems which are difficult to maintain.

b. D2S2 uses slightly different terms called

   1) the function logic model. They build for each of the most important (not defined) elementary (not defined) functions, a logic model showing entity types and relationships necessary to support that elementary

function. A list of attributes possessed by each function is also compiled. It does not state how the entity types and their relationships, nor how the function attributes may be obtained.

2) the function dependancy diagram, which shows dynamic pictures of function hierarchies. This diagram portrays functional dependencies by depicting which functions must proceed others in time. Again, how this information is to be obtained is not stated.

c. SASD uses functional specification in the sense of

1) the functional requirements. They define functional requirements as "a precise description of the requirements of a computer system". This includes a statement of the inputs to be supplied by the user, the outputs desired by the user, the algorithms involved in any computations desired by the user, and a description of such physical constraints as response time, values, etc. [20].

It seems that by this term they mean a detailed statement of what the system is to do, and it is free from physical consideration of how it is to do them. The term functional requirements (or as they mean functional specification) is the re-statement of the problem in a manner which emphasises the data flow, and ignores procedural aspects of the problem [20]. These specifications are expressed by using decision trees, structured English, data dictionaries, etc.. They stress that the functional specification should not be developed on a machine dependent basis, and should give an idea of the end product which is to be achieved. Furthermore it can be used as evidence as to what the user prefers to have.

2) the classical product of analysis, a description of a system to be implemented [5]. In general it is not possible to deduce the characteristics of a function

from the use of the functional specification. The adjective functional appears to be without content.

d. NIAM does not use the term implicitly or explicitly.

e. SYSDOC methodology is basically data-oriented and therefore not suitable for describing functionally oriented systems. The term functional specification is not used by them. However they do use the term requirements specification.

f. SADT does not specifically define the term functional specification. However they say that the result of functional analysis is the functional specification, and functional specifications serve as one of the inputs to the design phase of SADT. The major components of functional specifications are provided by:

i) Activity diagram (describing the system in terms of activities).

ii) Data diagram (describing the system in terms of data).

In SADT the requirements are converted into a functional specification and are expressed as a functional model which shows cross-referenced activity and data aspects of the system and represents "what" the problem is. They also use the term functional architecture to express the layout of the activities performed by a system. In general activities approximate to functions. It does not appear possible to deduce the characteristics of a function from the description of the functional specification.

g. SDL defines the term specification of a system as a description of the required behaviour. This term is used by them to define the requirements of the system and can be specified (i.e. consists of) by

i) set of general parameters required by the system,

ii) functional specifications of its required

behaviour.

By functional specifications of a system they mean the specifications of the total functional requirements of the system from all significant points of view. The functional specifications can be portioned into a number of functional block specifications. Each functional block contains one or more processes which performs a related group of functions e.g. call charging, signal recognition, maintenance, etc..

In SDL functional specification of a new system or new facilities and functions can be expressed by using a formalised method called SDL/GR which is more compact and has a predefined and well specified set of concepts, rules, symbols and diagrams. They claim this method is easy to understand.

Whilst there is thus a method for representing functions there are no criteria for identifying functions.

The above set of definitions and descriptions appears wholly inadequate in order for someone to actually produce a functional specification. It is almost impossible to frame a set of questions which would determine whether or not a functional specification has been produced let alone whether it is correct or not.

3. Functional Decomposition

This is the third term connected with functionality which appears widely in the literature. Again our purpose is to attempt to deduce the characteristics of functions from the knowledge that they can be decomposed.

The IEEE defines functional decomposition as "a method of designing a system by breaking it down into its components in

such a way that the components correspond directly to the system functions and also sub-functions".

The terms "system functions" and "sub-functions" are not explicitly defined. A system is defined as:

1) A collection of people, machines and methods organised to accomplish a set of specific functions.
2) An integrated whole that is composed of diverse interacting, specialised structures and sub-functions.
3) A group or subsystem united by some interaction or interdependence, performing many duties but functioning as a single unit.

None of which is greatly enlightening. Analysing the design methods reveals the following:

a) JSD considers functional decomposition as an activity of decomposing the system function which is organised as a hierachy of functional procedures. No criteria are given.

b) D2S2 uses the term functional decomposition to analyse an existing system (if there is one) to determine which of the elementary functions is to be supported, either in whole or in part. Functions are decomposed to about three levels of detail. The entities involved with each of these functions are listed and an entity-function relationship matrix is developed. This decomposition is stopped when the analyst is dealing with a level which is concerned more with the mechanics of the function rather than its purpose. The objectives of functional decomposition are to find out the business activity of an establishment independently of its organisational structure and to represent the elementary functions as transactions for future design. No criteria are given for the decomposing process.

c) SASD does not use the term directly. There is no criterion

for the decomposition of a function, but the criteria for measuring cohesion (the functional relatedness within a module) are given, so that poor decomposition can be recognised after it has been performed.

In SASD, which is based on the concept of modularity, a logical module is a defined function, having a name which expresses the purpose of the function.

The term functionality as used in SASD represents the concept of module strength, i.e. how tightly the processing elements within a module are related to each other and how strong is intramodular functional relatedness. The term cohesion is used for the same concept. Cohesion shows the structure of the problem i.e. how the problem is defined, because processing elements can be functionally related in any number of ways, in other words the structure of the problem can be decomposed in different ways by different analyst/designers. They have specified the following seven distinct levels which show the strength of cohesion, starting from the least or weakest cohesion.

1. Coincidental association,
2. logical association,
3. temporal association,
4. procedural association,
5. communicational association,
6. sequential association,
7. functional association.

Each logical function can be broken into a more detailed data flow diagram when it is no longer useful to subdivide the logical functions. Their external , business logic is expressed by using decision trees, structured English, etc..

A module whose processing elements are designed in a way

that they are elementary in nature, and every element of processing is an integral part of, and is essential to, the performance of a single module, is called a completely functional module.

SASD claims that a functional module can be identified by a process of elimination and examining the design for any potential defects in the form of low cohesion. This elimination is preferably repeated until the module functions can be described fully, and accurately, in a single English imperative sentence with single transative verb and a single object. So if the module is functional in nature, it should be possible to describe its operation fully in the above manner.

With this criterion, which appears to have no experimental or theoretical backing, it is possible to recognise a function only after it has been fully designed and possibly even implemented.

d) NIAM Here functional decomposition means the decomposition of information flow, in order to reduce the complexity of the transformation. This is achieved by dividing the transformation capability of one function into more than one function (called sub-functions). These sub-functions possibly can be further decomposed until functions are obtained for which it is possible to describe in full detail both its transformation and its information flow. It appears that the decomposition is not functional but data decomposition.

e) SYSDOC does not use this term being a data oriented methodology.

f) SADT uses the term functional analysis for functional decomposition nearly in the similar sense. functional analysis starts with a statement of systems requirements

and delivers functional specifications as output. During the phase of functional anlysis emphasis is given to analysing and documenting "what" the system is supposed to do rather than "how" it will do it. However in some cases, considerations about "how" also take place. The boxes in the activity diagram which represent a function can be broken down, when needed, into a number of more detailed boxes. This means each function can be decomposed into simpler functions. This process of functional decomposition is done in a top-down structured hierachy, with not less than three per level, and an upper limit of six. This top-down approach avoids providing too many details at once, and depicts uniform and systematic information. Again no criteria are given for the actual process of decomposition.

g) SDL. In SDL a functional block (which is like a module or subsystem) can be decomposed into sub-blocks and sub-channels by forming a hierachical multi-level description of the system. Each sub-block represents a subsystem. The information contained in the upper levels should be contained in the interfaces of the lower levels. This can be described by a tree structure of blocks such as

During functional block decomposition it is only necessary to make sure that the most detailed sub-blocks contain processes. It is claimed that this makes the description more meaningful.

One of the users of "A System Design Methodology, based on SDL" [48] mentions that, "functions are generally decomposable into sub-functions, all of them will have the same behaviour". There are no particular criteria in this methodology, for functional decomposition, and it is mainly based on the designer's decision. If a function cannot be decomposed then it must be a process. This functional decomposition stops when all the functions cannot be further decomposed, and at this stage all are processes, and as such can be implemented.

The conclusion reached from this analysis of the major design methods is again that the term functional decomposition is used lightly. No criteria are developed and yet the audience are expected to know what is meant.

4. Definition of Function

As an aid to future analysis a definition of function is vital. To this end the following is the best which the authors can construct from analysis of the literature and their own pragmatic knowledge.

With respect to a software system, existing or proposed; a function is a process, activity, event or task which performs some perceptable action, set of actions or causes a change of state. A function has the following properties:

1. *It produces by virtue of its actions, at least one distinct output. This output may consist of one or more, data items, messages, conditions, function descriptions, or changes of state, in any combination.*

2. *It may have zero or many distinct inputs. These inputs may consist of data items, messages, conditions and function descriptions, in any combination. Such inputs may be transformed, combined or otherwise manipulated by the actions of the function to produce the outputs.*

3. *The actions of the function may be dependant on the state of the system,*

or on the value of the inputs, or both.

A distinct input or output will be loosely defined as, an individual item, a class or set of items; such an item, set or class of items being deemed distinct by the relevant beholders.

To this definition, for mathematical convenience is added the null function which has no outputs. This function has no practical use since it would never be known whether it was working or not. Note that a validatory function, say, checking the ordering of data, must have one output through which to report failure. The above definition also includes the possibility of the identity function.

There are a number of precise definitions in the literature, e.g. "A function maps inputs onto outputs" is a typical exemplar. Such definitions are meaningless in the search for functional requirements. Neither users nor analysts find this helpful. Nevertheless the underlying meaning of the formal definition is captured in the above general definition.

The previous definition of a function covers at least the following cases.

i) A mathematical package which produces the solution for one complex equation. No inputs, the "solution" is the output.

ii) A switching system which merely takes "messages + address" structures and routes them in some way. The outputs are identical to the inputs.

iii) A transformation system, such as a compiler, which takes one representation of an entity (a program) and outputs another.

iv) An operating system in which the response (outputs)

depends on whether one is logged on, editing, etc..

v) A commercial D.P. system which takes input records and produces analysis tables.

vi) A program which takes input and produces another program to perform a particular function or class of functions. An example would be the YACC [53] system.

If the function has at least one of the following properties:

a) it produces more than one distinct output;
b) it has more than one distinct input;
c) its distinct actions are dependant on the state of the system;
d) its distinct actions or consequent changes of state are dependant on the values of the inputs;

then the function might be decomposable into sub-functions. A distinct action is one which a beholder perceives as being indivisible, inseparable or otherwise unique.

The precise mechanism which can be invoked will be highly dependant on a number of factors. One factor which will not be discussed here, is the dependancy on the mechanism by means of which the actions will be performed.

That a function can be decomposed into sub-functions does not necessarily imply that it should. Reasons, such as efficiency or resource limitations, may motivate towards composition.

Functions may also be decomposable if there is knowledge of the specific actions which are to be performed.

The action of decomposition may add to the number of distinct system inputs. This can happen, for instance, in a process whereby the mechanism to be used in the decomposed function requires the use of tables of data, or parameters, etc..

The converse is however not true. That is, if two functions are referred to in conjoined phrases, or separate sentences etc., it does not imply that they are independant. For example:

"Before driving adjust the mirror and check
that no traffic is approaching" (17)

does not imply that the two functions (to adjust and to check) are independent. It is pragmatic knowledge that links the two. One cannot check until the adjustment is complete. The two functions are also in temporal order but this also is not obvious. The inclusion of "then" after the conjunction would have made this clear.

There is also no reason to assume just from ordering in text, that any two functions which appear should be evaluated in the order of their occurrence. For example:

"emptying the mixer will be simplified if the motor
is revved and water is added" (18)

Here the water must be added before emptying starts.

The relationships between functions will frequently arise in prepositional phrases

"Printing the paper will be performed
after the checking of spelling" (19)

Note here that both functions (to print and to check) are not described by a verb, but by a noun (nominalised verbs). The above relationship is temporal as indicated by "after". Other relationships can be confusing, e.g. prepositional phrases such as:

"together with...."
"at the same time"

and the verbs

Relative clauses are used to modify nouns and as such are essentially selecting from the class of all exemplars of that noun. In the case of iteration, the selection process is usually reasonably clear but for embedding considerable confusion can arise. This is particularly true in situations where the pronoun "that" is omitted such as in the example above.

However consider

> "the table that stood in the house that Jack built"
> and
> "the table stood in the house that Jack built".

Did Jack build the table or the house? In this case the ambiguity is greater in the sentence containing the pronoun "that".

It appears that the binding order of relative clauses is not defined and hence this is a source of ambiguity.

3.2. Relationships between Functions

In this section is is assumed that some functions have been detected and the relationships which exist among them are to be found.

If two functions are completely independent then they might be described in a single sentence (with a conjunction between the fragments describing each function), two separate sentences, or two separate paragraphs (or however much text is needed). For example:

> "Insert the key and adjust the seat".

The action of inserting the key and the action of adjusting the seat are completely independent.

"the same difficulty" (15b)

To say that "the traveller is commercial" is nonsense, as it is to say "the difficulty is the same". It is not clear whether these adjectives can be deemed to define functions.

From this it can be seen that not all noun groups necessarily imply functionality. For example:

"a plane tree" (16)

is not a subset of trees which have the characteristic of being plane.

Relative clauses are also a source of problems. These clauses are sentence-like structures which usually start with the pronouns,

"who", "which" or "that".

These clauses can be iterated or embedded. Examples are: for iteration,

"This is the dog that worried the rat that ate the malt that lay in the house that Jack built".

and for embedding,

"the birds the woman who painted fed died".

Here the "woman who painted" describes a particular woman, "(that) the woman fed" describes a particular set of birds. Finally, these particular birds died.

There are no limits to the possible extent of either the iteration or the embedding other than the limits of human understanding.

of subject, or object of the verb form. Consider:

"company purchases" - purchases by a company
"machine calculations" - calculations by a machine

where purchases and calculations are nominalised verbs which are now subjects whereas for;

"machine purchases" - purchases of machines
"weather calculations" - calculations about the weather

the purchases and calculations are nominalised verbs which are now objects.

Nouns may also be modified by the use of adjectives. There are two types of adjectives which are of interest to this paper. The first type are the predicating adjectives which work in the same sort of way as previous noun strings, thus

"a red car" (14)

is a subset of cars with the attribute of being red. Care must be taken with the use of the predicating comparative adjectives, such as "big", "small", "large", etc.. For example, "a big mouse" will generally still be much smaller than " a small elephant". So "a big car" is the subset of cars which have the attribute "big" when this is applied to "cars". Predicating adjectives may be successfully paraphrased as

"a car which is red" (14a)

The second type of adjective of interest to this chapter are the non-predicating adjectives. These cannot be paraphrased as in (14a). For example:

"a commercial traveller" (15a)

in fact combine as a compound noun to form the selector "water meter". This could be resolved by writing "water meter" as "watermeter" or "water-meter". There appear to be no rules concerning the construction of compound nouns e.g. "field-mouse" and "harvest mouse" in the Oxford Dictionary. Why one of these nouns should have a hyphen and the other not is a matter for conjecture.

Continuing the theme of noun-noun modification, that is, when one noun is modified by another, it can be seen that other relationships can be expressed. For example:

"machine maintenance" (11)

expresses the function of "maintenance" being performed on "machine".

"COBOL program" (12)

describes a program written in the Cobol programming language not a selection.

While

"card reader" (13)

is a name for something which reads cards and hence an implied function, but a

"human reader" (13a)

does not read humans. It generally describes a human who performs the action of reading.

Nouns may also be modified by nominalised verbs, that is, by verbs used as nouns. These usually occur with the relationship

telecommunication systems is an example of a method applicable to this sort of problem. The functions which perform the change of state need not be explicitly stated.

Functionality can be expressed in other ways. Consider the phrase:

"water meter cover adjustment screw" (10)

This is a string of nouns but the interpretation refers to a series of subset operations being applied to the head noun. There is initially a set of screws (input data). Selecting the class of adjustment screws results in an output set "adjustment-screws". Further selecting those which hold covers, results in a further output ("cover-adjustment-screws"), and so on. A requirement specification may refer to a set of screws and then state what should be done with each (or some) of the sub-sets. The selection function, or functions is/are only implicitly mentioned. One of the more interesting properties of functionality when presented in this way is that temporal ordering of the functions is clear. One cannot have

"cover water meter adjustment screws" (10a)

"adjustment cover meter water screws" (10b)

"adjustment water meter cover screws" (10c)

although

"cover adjustment water meter screws" (10d)

might be acceptable provided that cover-adjustment is one of the selection functions. The relationship functions will be returned to later. Note that there are lexicographic problems because "water" and "meter" are not two separate selectors but

is, at least, that of stopping the production line. (The reader is expected to visualise these sentence fragments in a larger technical context).

The word "stationary" is an adjective and is used in its own adjectival phrase. The use of adjectives can often infer an action and this observation will be examined later.

Worse still there are the implied actions which cannot be resolved from the text but depend for their resolutions on a larger body of knowledge. This body of knowledge is usually referred to as PRAGMATIC KNOWLEDGE. An example is the problem with the lorry, example (6). No lorry can perform the tasks of going out and collecting anything. There is the implied knowledge that a lorry must be driven to the collection point and at that point another action "collect" will be performed (probably by humans or some other agency) which will cause something to be placed within the lorry. Another example of this type occurs when the word used to detail the action is a label for a defined sequence of actions. i.e.

> "The product is weighed up to mark "A"
> on the weighing machine". (9)

The weighing process referred to here implies the process of continually adding an amount of the product until its weight can be seen to be "A" units on the scale on the weighing machine. This process is what is referred to by the phrase "weighed up to".

Collecting knowledge concerning the states can be just as misleading as collecting verbs, because the information may also refer to the background.

It is interesting to note that it is possible to completely specify a complete system by means of stimuli and resulting states. The SDL design method [47], designed for

All these sentence fragments can be used to describe the process in which a program is run and which consequently produces output. They may even all appear in the same document and hence be classed as different functions when the same function was originally implied. Or conversely they may be classed as the same function when the writers actually intended to emphasize the subtle differences which could be inferred from the different terms.

It is already clear that merely collecting the verbs is not sufficient to determine the functionality implied in either a requirements or specification document.

Let us now examine the second part of the original premise. It would appear initially that if all the noun phrases were collected, a list of all the data entities would be available. This is also not true. In many cases, similar to the case for verbs, nouns are used to provide background information.

Consider the text:

> "A patient-monitoring program is
> required for a hospital." (7)

In this example, "a patient-monitoring program" introduces the subject of the requirements document, but "hospital" is an external entity to the proposed system. It is not mentioned again and has been used for scene-setting.

Changing the direction of the attack, and dropping the original over simplistic premises it must be noted that actions (or functions) need not be stated explicitly. They can, for instance, be inferred from the text. For example:

> "The production line is stationary". (8)

describes the state of the production line. The action implied

a functional viewpoint will probably have a relatively higher number of active verb uses than a system described from an informational (data) viewpoint. Thus classifying the verbs in this way is unlikely to be particularly helpful.

Another problem is that many verbs are also used to describe other features of the system such as data structures. Consider:

> "The words in the telegrams are separated by sequences of blanks". (4)

> "The stream is available as a sequence of letters, digits and blanks on some device". (5)

Neither of the verbs "to be separated" and "to be available" describe actions. Another interpretation of these sentences could be the verb "to be", modified by an adjective in each sentence, but this also does not describe an action.

A bigger problem arises from the fact that verb phrases in natural language text may consist of more than one verb. e.g.

> "The lorry is going out to collect...". (6)

where there are the verbs "to be", "to go out" and "to collect". Ignoring the copular verb "to be" for the moment, then neither the verb "to go out" nor "to collect" describes the action which must be performed. We will return to the problem shortly.

Another problem which arises, is the use of different verbs to describe the same process.

> "The program will be run...."
> "Its execution will yield...."
> "Its output will be...."
> "The process terminates...."

Indeed each sentence may be rearranged and then given in the imperative form, as an active command:

"Produce a grid reference using the program". (1a)

"Operate the mixer for time T". (2a)

Consider, however, the following text:

"For the production line monitor consider
the main drive motor". (3)

There is one verb "consider". If this is a statement within the requirements document then it is clear that the production line monitor (the subject of the requirements document) would not be expected to consider anything. It is only the reader who is expected to do the considering. This is an example of a verb which is used for scene setting and background information. Other verbs of this background type are "seem", "look", etc., i.e. the copular verbs. To emphasize this point table 1 shows a breakdown of the verbs in use in a small sample of simple specifications.

	Spec 1	Spec 2	Spec 3	Spec 4
BACKGROUND	8	3	4	2
FUNCTIONAL	4	7	5	2
TOTAL	12	10	9	4

Table 1

The relative number of active verb uses to passive verb uses will fluctuate depending upon the viewpoint from which the system is being described. For example, a system described from

CHAPTER 3

DETECTING FUNCTIONALITY IN TEXT

The problem which we address here is whether it is possible to detect functions in natural language text. If functions can be so detected then can their relationships be inferred from the text? Basically the problem is examined from the point of view of a validator, but it should be noted that precisely the same problems will be encountered by someone writing a specification from a natural language requirements document. The discussion firstly considers how functionality is expressed in natural language text, then the sources of relationships between functions are analysed. The next section seeks ways in which constraints upon the working of functions may arise. The final sections cover the problems of textual terseness, through ellipsis, substitution and informality, and some of the sources of ambiguity.

3.1. Discovering Functions

It is claimed in a number of books [12] [21] that the functions of software will be described by the verbs found within the requirements document and that the data structures in turn will be described by the nouns. This premise will be examined first and then the search will be widened to look at further issues.

At first sight the premise appears to work for verbs. Consider the following texts:

"The program produces a grid reference". (1)

"The mixer operates for time T". (2)

Each of these describes a particular aspect of an action.

be in the form of an input, a set of constraints, a corresponding action (function) and the resulting outputs. The whole system is then an amalgam of all these fragments. In the inverse problem, where a complete system exists, knowledge may be required as to which components are relevant to a particular input-constraint set. The input-constraint sets and corresponding component relationships are known as "threaded links". This idea of taking the threaded links and building a system was one of the early ideas of the SREM system [15]. It involved the building of R-nets from these fragments.

The rules for composition are also not well formed, thus a validator is faced with a severe problem.

Summary

In this chapter a number of major design methods have been analysed to find whether they indicate the steps which lead to a knowledge of functionality for a given software project. Our conclusion is that they do not specifically state such steps, although interestingly such a knowledge must be gained in order to use some of the methods.

In the absence of suitable criteria the authors have attempted to define such criteria, basing their definitions on the snippets of insight which were discerned in the literature. In the next chapter some of the indicants of functionality referred to in this chapter are pursued in detail.

stores are frequently passed as parameters to sub-functions but are defined globally.

One of the problems encountered in decomposing functions can be seen in the SADT method. In SADT each process (function) has four sets of arrows associated with it. There are the input arrows (from the left), the output arrows (from the right), the constraint arrows (from above) and the mechanism arrows from below. The constraint arrows carry with them the criteria which must be satisfied in order for that process to perform its task. The problem referred to above concerns the separation of the data streams from the constraint streams. It is frequently found that the output from one function appears to act both as a constraint to another function and an input data stream to further functions. The decomposition then appears difficult because the reader is unsure of the significance of the data and constraint inputs. The confusion between data and constraints appears fundamental. It probably arises from difficulties encountered in expressing ideas in natural language. These difficulties arise because sufficient care is not taken to express the ideas in a way which makes the distinction between data and constraints clear. This topic is dicussed in detail in the next chapter. Basically the constaints should be boolean expressions. The data however may also have boolean values.

5. Functional Composition

In many projects it might be necessary to perform a composition of functions instead of, or as well as, a functional decomposition. What proportions of projects fall into one class or another is not known or indeed whether there is a need for both composition and decomposition in a significant number of projects.

The need for composition arises from the way in which the user perceived functions can arise. Each user perceived function may

The actual decomposition rule will depend on many factors. Two such factors are:

1) If the structure of the input data and also the output data are corresponding, then the hierachy can be a Jackson tree structure.

   a) A special case of this is when the inputs consist of a sequence of clusters of independant data and the output relates directly to the clusters. Then the function representing the whole can be decomposed into a sequence of functions.

      There are two further sub-cases:
      i) the functions do not communicate i.e. inputs of each are independant of the outputs of others.
      ii) the outputs of each function in turn is also an input to its successor.

2. In general any functions resulting from the decomposition of another function must have a method of communication. That is, there must be a mechanism by means of which the output of one function can become (part of) the inputs of another function. There are two main methods:
      i) transient values,
      ii) data stores.

In computer software the transient values are locally declared variables which transmit the values generated by one function to another (other) function(s). In some languages, for technical reasons, these variables may be defined globally. Generally, however, such variables are passed as parameters.

The data stores are global repositories of data and they have a lifetime exceeding that of the individual functions. They may even be permanent, residing for example, in a long term data base, library or archive. In computer software these data

"takes place"
"happens"
"occurs"

may imply simultaneity or not depending on context.

"If you go to town could you visit the library at the same time" (20)

does not imply being in two places at once.

If the prepositional phrase contains "by" and also a noun group containing "ing" then the phrase contains an implied mechanism for performing the action, rather than a condition.

Compare

"the mixer is emptied by tipping" (21)

where the prepositional phrase describes how the mixer is to be emptied.
With

"the mixer is emptied by the wall" (22)

in which the prepositional phrase is locative.

The problem of detecting functions which are the subfunctions of others is extremely difficult from graphically unstructured text. If graphical structuring is present, e.g. indenting and assigning hierachical numbers to paragraphs, the problem is greatly eased. Nevertheless there does not appear to be any well developed, generally accepted convention either for expressing such relationships textually or by graphical structure.

3.3. Constraints

When a function is detected it is necessary to determine the

conditions for the performance of this function, i.e. the constraints. Some of these constraints (e.g. temporal ordering) have been briefly discussed previously. The detection of these constraints is hindered by the lack of rules for the scope of such constraints. It is, for instance, possible to qualify the function within the sentence which defines or introduces that function. Such a constraint may be specified by a prepositional phrase.

"The lorry must be parked by the wall".

This specifies that the operation of parking the lorry is restricted to a place by the wall (it is locative).

Unfortunately the constraints (pre-conditions) may have been explicitly introduced earlier in the text, by sentences such as;

"in the following it will be assumed that....."

where the extent of the following text is not defined. Occasionally a specific indenting notation is used, in which case there is the implicit assumption that the scope is that of the indentation. More frequently the constraints are introduced implicitly, that is they arise in the text without necessarily being highlighted, and these constraints are cumulative. They are cumulative because there is no natural language mechanism for dropping constraints other than explicitly stating those which will cease to apply. Usually pragmatic knowledge and content are used to discard these constraints.

A further level of difficulty arises from the use of backwards directed constraints. These are frequently in the form of afterthoughts, e.g.

"Of course, this only holds if....."

where the context of "this" is not explicitly stated and which can be inferred to relate either to the last statement, or part (even the whole) of the preceeding text.

There is also no mechanism in natural language text to discover whether the constraints are contradictory. For this the constraints need to be formalised in some way in order to permit analysis with a mathematical basis. Inadequacy of the constraints is unlikely to be discovered by any mechanism except considered thought by those involved.

The types of constraints can vary widely, e.g.

temporal,
locational,
possessional,
communicational.

The sources of knowledge concerning these constraints are again diffuse. They occur in prepositional phases,

"....under manual control",

in adverbial phrases,

"....move the records on Monday".

Through the use of modal verbs such as,

can, must, will, shall,....

where these modal verbs are normally used to qualify other verbs, as in,

"the software must control...."

Sometimes, the constraints introduced are strong enough to

imply a "mechanism" whereby the function can be implemented.

> "The program extracts keywords and deduces the exact location by scanning the input text near the keywords".

Here the verb "to deduce" introduces a function and the preposition "by" introduces a mechanism "scanning" whereby the program will "deduce" the exact location.

The introduction of constraints into natural language text can also be achieved by the use of conditional sentences:

> "If a then b else c".

Such a form is recommended in a number of texts [12] [5]. However, the common use of such structure can vary dramatically. Consider the following text in which the condition is spread over two sentences:

> "For order with payment or good credit, inventory is then checked to see if the order can be filled. If it can a shipping note with an invoice (marked "paid" for prepaid orders) is prepared and sent out with the books".

Here a condition is introduced (the order can be filled). This condition is substituted in the second sentence by "it", which makes the document more informal and more readable. This technique may be extended over many sentences. There is then the problem of the alternative, that which happens when the condition does not apply. As discussed later, an ommission of an alternative leaves unresolved ambiguities. If an order cannot be filled is nothing to be done, or is an invoice to be sent without the shipping note and books?

The alternative can be introduced explicitly, implicitly, or by the introduction of another condition.

"Otherwise a confirmation of order will be sent".

"Confirmation of order notes will be sent with estimated delivery dates".

"If no books are available a confimation of order will be issued".

Failing to recognize the relationship between the conditional in this last alternative with that of the original statement will lead to the possibility of infeasible branches in an implementation.

It is clear from formal methods for proving or partially proving programs [45], that pre- and post-constraints for functions are a powerful weapon. This is particularly true in a data flow model where the data links between functions (processes, activities, etc.) are modelled. In such a model there is the implication that the availability of data is sufficient to trigger the process. This may not necessarily be true, there may be conditions deriving from the state of the system, or from other processes. In SADT each process is governed by the imposition of conditions. However, in this method confusion between data and constraints occurs frequently. This problem appears to be resolved if pre- and post-conditions are introduced explicitly.

One extremely useful post-condition, is to derive the condition whereby the completion of the process can be established. Often such a post-condition can be trivially true because it is not possible for the process to terminate any way other than after having completed its task.

Distinguishing between constraints and mechanisms is also difficult:

"When the data is sorted...."

can imply either a post-condition, or it can state that a sorting process must be invoked.

3.4. Ambiguity

It is widely recognised that natural language text contains many sources of ambiguity. It is frequently claimed that this is a major disadvantage of using natural language. In this section some of the causes of ambiguity are investigated. Consider the sentence

"The lorry must be parked by the car park attendant".

The cause of the ambiguity here is the preposition "by", which can be interpreted in its locative form (specifying the place to park the lorry) or in its subjective form (specifying who is actually to park the lorry). The consequent information on the parking function is dramatically different. In the first case the sentence describes a constraint on the function. In the second case the sentence describes how the function is to be performed. This ambiguity cannot be resolved without reference to some further relevant knowledge.

Throughout the discussion preceding this section, a specific interpretation of the sentences or fragments has been taken. In many of them other interpretations are possible.

Some sources of ambiguity are:

A) LEXICOGRAPHIC AMBIGUITY

This occurs because many words have many different common meanings e.g.

"dog, cow, badger, fly, horse".

This list of nouns could also be a list of verbs!

In the computing environment confusion can arise with the use

of words such as "process, effect ...."

Probably the worst examples are words such as "cleave" which has two opposite meanings. One is to split apart, etc., the other is to adhere to, etc.. A list of such problem words would be useful.

B) LOGICAL AMBIGUITY

There is no universal agreement in natural language as to the semantics or meaning of certain syntantic structures. For example, the construct

IF a THEN b

has two interpretations:

INTERPRETATION 1

If a is TRUE, the content of b is relevant
If a is FALSE, the content of b is irrelevant

Example: If the sun is shining then dig the garden.
This does not omit the possibility of digging the garden if the sun is not shining.

INTERPRETATION 2

If a is TRUE, the content of b is relevant
If a is FALSE, the content of not b is relevant

Example: If a person is convicted then they are a criminal (and if a person is not convicted then they are not a criminal).

Clear documentation of the two interpretations can be found in legal histories where both possibilities are observed [49]. Computer specialists, by their training, always take the first

interpretation.

The problem is compounded by the many different ways in which a conditional sentence may be expressed: For example

(i) c UNLESS a, IN WHICH CASE b
(ii) c. HOWEVER, IF a THEN b
(iii) c, BUT b WHERE a
(iv) - ONLY WHEN a SHOULD b
(v) IF a THEN b. OTHERWISE b.

Each of these conditional statements reduce to the logical structure:

IF a
THEN b
ELSE c

It should be noted however that conditions are not always expressed within one sentence (examples (ii) and (iv)). Similarly, it should be appreciated that in general use structural words and punctuation marks might be omitted. Example (ii) could be written:

c. HOWEVER, IF a, b

When a condition is expressed in more than one sentence, the alternative clause (the ELSE part) might not be expressed. This will generally result in "loose-ends" in a specification.

Clearly great care is needed in the use of such logical constructs.

The conjunctions "and", "or" and "not" are sources of ambiguity when more than one of them appear in a condition expressed in English. Consider:

"packages which are being sent abroad and weigh less than 20Kgms., or are marked urgent are to be sent airmail".

What happens to a domestic parcel weighing 10Kgms. marked urgent? If one interpretation is used then it will be sent airmail, if the other interpretation is used then it will not be sent by air mail. It is not clear whether the 'and' includes one or both of the adjectival phrases following. The problem is that the scope of the logical implications are not defined. For example, the scope could be the next phrase or the whole of the remainder of the next sentence. e.g.

"No person shall be Senator who shall not have attained the age of thirty years and been nine years a citizen of the United States, and who shall not when elected, be an inhabitant of that state for which he shall be chosen".

Does the first not negate the phrase "attained the age of thirty years", or "attained the age of thirty years and been nine years a citizen".

In problems of this type a list of the predicates seems preferable.

c) GRAMMATICAL AMBIGUITY

There are four main areas of grammatical ambiguity. We shall examine each of them in turn:

(i) The bracketing of constituents within sentences can sometimes cause ambiguities. We have already seen one example with compound logical statements in the previous section. The wider problem occurs when a modifier can be attached to more than one of the preceeding constituents of the sentence. These

modifiers can occur as relative clauses or preposition groups. For example:

"The man saw the woman in the park"

Who was in the park? This type of ambiguity can sometimes be resolved using either contextual knowledge or pragmatic knowledge.

(ii) The bracketing within constituents of a sentence can cause ambiguities. Consider the sentence:

"We want to attract more intelligent students".

Here, without reference to further knowledge the sentence is ambiguous. The ambiguity can only be resolved if it is known whether the general level of intelligence is to be raised or whether a greater number of bright students is required.

(iii) Interpretations of nominallizations can cause ambiguities. Consider the sentence in which the verb "to shoot" has been converted to a noun:

"The shooting of the hunters was disgraceful".

The ambiguity arises from the fact that "the shooting of the hunters" can have a subjective interpretation (they were doing the shooting) or an objective interpretation (they had been shot). Again, this type of ambiguity might be resolvable given appropriate background knowledge.

(iv) Ambiguities can be caused by words which belong to more than one syntactic group. For example, words such as "flies" and "process" which can be both a verb and a noun. If more than one of these words occur in a

sentence then many interpretations are possible. Consider the sentence:

"Time flies like an arrow",
which has at least three interpretations.

D) TERMINOLOGICAL AMBIGUITY

The problem here is that in different technical areas particular words have different technical meanigs. e.g.

"Buy a car with two wheels and a banana".

In grocery circles, a banana is an item of fruit; in hot-rod car circles a banana is a fancy chromium plated exhaust-system. In motor-cycle circles, the banana might be interpreted as a fancy type of seat! No doubt other interpretations exist.

Resolution of these problems requires a careful definition of terms, for example the construction of a data dictionary.

Other problem areas leading to possible ambiguity.

The casual way in which people use punctuation marks can create misinterpretation. It is not strictly an ambiguity but does cause problems. Compare the sentence:

"The Prime Minister called for an end to violence and internment, as soon as possible".

With the similar sentence:

"The Prime Minister called for an end to violence, and internment as soon as possible".

Worse still, punctuation marks are often omitted. Consider:

"Drink ye all of this"

This could mean either

"Drink ye, all of this"

or

"Drink ye all, of this"

Legal documents avoid these problems explicitly by avoiding the use of punctuation. The ambiguities however seem to remain implicitly.

Humans are also very casual in the way in which they communicate numbers. The phrase "twelve and a half" is usually taken to mean the numeric value 12.5. However, the phrase "one million and a half" is usually taken to mean 1,500,000 NOT 1,000,000.5.

Whilst on the subject of numbers it is worth noting that humans often omit the qualification of numbers (i.e. what they actually refer to). Consider:

> "A basic practice allowance of £1000 a year, with a standard capitation fee of 15p a year for each patient under 65 and 20p a year for patients over 65. For each patient after 1,000 there would be a supplementary capitation fee of 30p a year".

The phrase "over 65" would generally be taken to mean "over 65 years of age", whilst "over 1,000" would be taken to mean "over 1,000 patients". This example demonstrates the vagueness with which humans use the terms "greater than" and "less than". In the above, a doctor would strictly get no additional allowance for a patient exactly 65 years old.

Numbers can be used to demonstrate the terminological ambiguity claim made earlier in this section. What does "negative" mean? In mathematical circles, a "negative" number is a number whose value is less than zero. In laboratory circles, a "negative" sample is a sample that does not contain any trace of a particular feature looked for. The police, however, consider a

breathalyser test to be negative if the reading is less than 80 mgm/100ml.

From this it is clear that great care must be exercised in documents involving numbers.

Another area which leaves the opportunity for misinterpretation occurs when certain operands are omitted from a sentence. Consider (ignoring other ambiguities):

> "The program extracts keywords and deduces the exact location by scanning the input text near the keywords".

What are the keywords extracted from? This information has been left implicit.

There are many ways in which ambiguities can arise. It is essential that before any semantic processing of the text can be performed, ambiguities must be removed. This in itself leads to two problems:

1) The identification of ambiguities, and
2) the resolving of the detected ambiguities.

If humans perform the analysis the major difficulty lies with 1), whilst with automated systems it is 2) which poses the major problem.

3.5. Ellipsis, Substitution and Informality

These processes produce sentence fragments, incapable of standing alone as sentences and which rely for their interpretation on the context and the ability of the reader to reconstruct the "understood" sentences to which they relate.

1) Ellipsis is a process whereby a word or many words can be omitted from a sentence, the complete meaning

being obtained by inference e.g.

"The colour is unsuitable".

This sentence by itself is meaningless, what is needed is knowledge that it is a room being referred to. This knowledge can be obtained either by reference to the previous text or by inserting the missing words e.g.

"The colour of the room is unsuitable".

The danger with ellipsis is that there may be more than one interpretation based on the previous text.

The removal of ellipsis is not usually desirable but attempts to ensure that it is unique and obvious what is being replaced is worthwhile. For example this last sentence would be better written as

"but it is worthwhile to ensure that what is being replaced is unique and obvious".

ii) Substitution is the process whereby a pronoun is used to replace a noun or noun phrase, e.g.

"It is a widely used technique".

Here "it" substitutes for the noun "substitution". The main use of substitution is to reduce the length of text. In this role it is extremely powerful but it can also be highly dangerous. The problem is that it is not clear which noun phrases are those which are being substituted for by a given pronoun and hence ambiguous sentences can result. e.g. (paraphrased from the definition of Pascal)

"Any text enclosed in curly brackets is called a comment. It may be removed from a program without altering its meaning".

There are two substitutions in the second sentence.

a). An analysis of 22 Cobol programs from a D.P. environment [7].

	Average per program
Cobol source cards	988.40
Comment cards	73.40
Cobol statements	528.09
Unexecuted Cobol statements	27.54%

b). Results from a few samples of commercial programs [8].

Case A: 10 programs from an accounting suite.

size: 181 - 1321 procedure division statements

Ter_2 = 72.4%

Case B: 3 programs from a large applications suite

size: 789 procedure division statements on average.

Ter_2 = 47.3%

Case C: 150 module system

size: approx 20K lines of code

Ter_2 = 53.2%

Figure 4.4

Results for the structural testing metrics when used to evaluate to conventional functional test data. Note that in general the statement cover (Ter_1) is usually about 10% better than branch cover (Ter_2).

CATEGORY	PROGRAM 1	PROGRAM 2	PROGRAM 3
MODULES	115	70	160
DECISION OUTCOMES	730	490	1040
FUNCTIONAL TEST COVERAGE	70%	75%	85%
ERRORS DISCOVERED	8	*	8
FINAL COVERAGE	90%	90%	90%
ERRORS DISCOVERED	4	*	2
ACCEPTANCE TEST ERRORS	1	1	0
ADDITIONAL ERRORS	7	5	0

Figure 4.3

Results from [6] showing 'coverage' of branches (Ter_2) achieved by a functional test, together with the number of detected errors. The final cover was achieved by utilising structural information.

	Ter 1 %	Ter 2 %	Ter 3 %	
TD1	78.56	63.56	40.55	Functional test data set.
TD2	96.41	88.89	63.98	Structural test data set.
TD3	97.80	91.36	65.84	Error exit test set.
TD4	98.00	92.59	67.08	Inialisation sequence.
TD5	98.60	93.00	67.39	Error exit test set.
TD6	99.00	93.83	68.01	Error exit test set.
TD7	99.40	94.65	68.63	Special case.
TD8	100.00	97.12	71.74	Special case.
TD9	100.00	97.53	72.36	Special case.

Figure 4.2

Results from [5] showing 'coverage' of statements, branches and LCSAJs attained firstly by the best functional test data and then showing how other tests were added to give an acceptable cover.

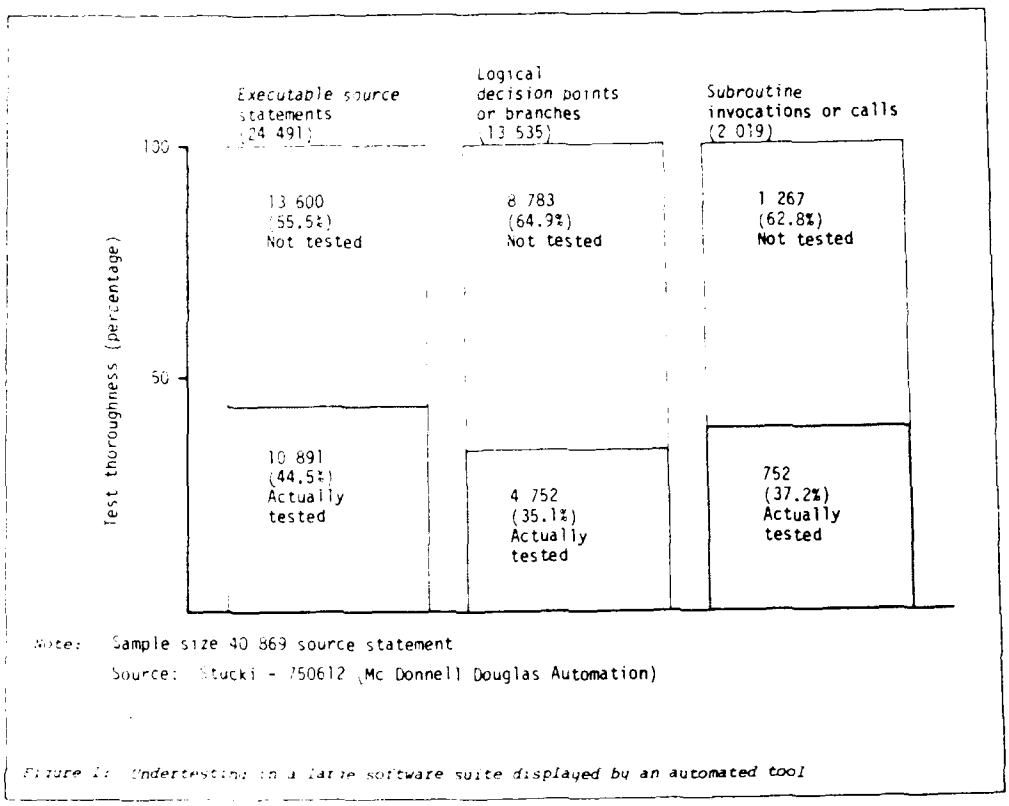


Figure 1: Undertesting in a large software suite displayed by an automated tool

Figure 4.1

specifications of these programs in order to classify the failures the authors have not been successful. There do not appear to be any general principles which can be established in order to ensure that these test cases are not missed. The only patterns which were observed have been incorporated into the method advocated in the next chapter.

Some of the test case omissions observed fell into the same category as that described above for the triangle program. That is, within a clear functional class it is possible, with only a minor degree of pragmatic knowledge, to deduce that special cases or subcategories might be worth exploring. The specification does not however explicitly indicate them.

In conclusion, the analysis of this chapter demonstrates that functional testing strategies do not achieve significant coverage of the program code. The code that is missed is largely implementing features introduced at the design phase. Some of the housekeeping code is currently successfully tested, more could be tested by organising the functional tests in a manner outlined in the next chapter and finally there is a body of code for which the authors see no external mechanism for executing (other than chance). In this latter case there is no necessary cause for concern provided a structural test strategy has also been performed. On the other hand this means that functional tests alone will not demonstrate the absence of errors.

knowledge of its presence. Note that if an error is introduced into the isosceles triangle component of this algorithm, four test cases will be unlikely to discover the error and yet the functionality will now differ from that intended.

Notwithstanding this problem, which is not resolvable from an external view, there is the thought that perhaps a functional approach should consider the overkill possibility. That is, the functional approach should seek potential sub-functions and provide test data for each of these, even though there is the possibility that these sub-functions may not have been implemented in the code. Thus in the triangle problem there should be six test data sets:

one equilateral,
one scalene,
three isosceles,
one not a triangle

This still will not approach the error detecting power of structural testing which would require of the order of thirteen test cases. Knowledge about these possible subfunctions does not come from the specification but either from a knowledge of the design or pragmatic knowledge.

Returning to the analysis of functional tests by means of the structural test metrics it is now worthwhile to differentiate between, these cases where the testers could have deduced missing test cases from the specification, and those where they reasonably could not. This is not an easy task because in general, for the major samples of software examined, the specifications are no more than descriptions. However for the mathematical software the situation is more clear cut because a precise specification is available. In this case 62% of the unexecuted code could have been executed by constructing test data deducible from the specification and combined with pragmatic knowledge. The remaining code was associated with design functions not discernable externally. In studying the

isoscoles triangle,
equilateral triangle,
scalene triangle,
not a triangle.

From this analysis it is reasonable to assume that there are only four paths within the program. It is indeed possible to write such a program. However, assume that our software developers choose an alternative proposal, perhaps for reasons of efficiency (figure 4.8).

From the specification there are four verb phrases:

1) to be written,

2) reads,

3) classifies,

4) forms.

The first is an action to be performed by the software implementers. The second is a trivial action of the program. The third is the principal action of the program. The fourth describes the data structures. Judging functionality from this viewpoint the function performed is that of classification and there are four such classes.

If the implemented code is that of figure 4.7 then four test cases, one exemplar for each class will not only execute every line of code but also every branch. However, if the implemented code is that of figure 4.8, then every line of code is not executed. Both programs are correct, they differ only in design, one chooses to use complex predicates and the other prefers an algorithm which counts the number of equal sides. The algorithm introduced in figure 4.8 is not referred to in the specification and hence a functional tester has no

The systems analysed in figure 4.5 are large, e.g. greater than 10,000 lines of code and it is possible that the proportion of housekeeping code grows at a rate which is faster than the length. If smallish programs are analysed [51] it can be seen that some of the code which is missed in even an exacting environment, can be classified as housekeeping. For example; error exits leading to failure messages and special cases which are dependant on the specific solution algorithm chosen are a significant portion of the untested code.

When the software referred to in figure 4.5 is dynamically analysed, the unexecuted code also falls into similar categories. That is, the unexecuted code consists of:

error exits,
special cases, and also
code associated with the housekeeping functions.

In fact most of the unexecuted code can be attributed to the housekeeping functions and this fact provided the main motivation for developing the testing technique described in the next chapter.

The problems of functionally testing a program can easily be illustrated by a well known example. It is a triangle program, for which the specification is:

> "A program is to be written which reads a set of three positive integers and classifies them into; those sets which can form the sides of isocoles, scalene and equilateral triangles, or which cannot form the sides of a triangle".

Two programs which satisfy this specification are shown in figures 4.7 and 4.8.

From the specification there are four classes of the integers which are significant.

comprehensively test the software nor remove all the errors.

It appears that the results are essentially independant of the application area. The variations can be quite dramatic but the picture which emerges is that the programmers seriously overestimate their performance. In the following analysis it will be assumed that results such as those presented above are produced by programmers conscientiously trying to do a good job. It is generally accepted that computer programmers are highly intelligent people and the statistics are such that they cannot be wholly explained on the basis of incompetance and lazyness.

An explanation as to why these functional tests provide a rather poor coverage can be made by the following argument. In figure 4.5 the results of an analysis of the type of each module of some software systems are presented. It can be seen that in each case nearly half of the modules are performing housekeeping functions, i.e., functions not reflected in the requirements. These housekeeping functions are purely a design artefact, introduced in order to perform some task determined purely by the design strategy. Without a detailed knowledge of how these housekeeping modules interrelate to the user perceived functions it is clearly unlikely that they will all be executed. The housekeeping functions referred to here are a subset of the design functions described in [32]. In figure 4.6 another analysis is made of two small systems written to the same specification and again it may be seen that housekeeping code dominates over code which is more clearly related to the user functions [41]. The classification of either code or modules into classes such as those discussed above is not a simple clearcut problem. It requires judgement and is subject to considerable error. No attempt has been made to bound this error because insufficient statistics are available. These results should be treated with some caution. Nevertheless the extent of the code implementing housekeeping functions can be considerable.

provide a very detailed functional test. The testers were given an unbounded time to think about the test data, which, since the software was a Cobol testing tool, was well defined (the Cobol syntax). In the event the results were very significantly worse than the testers expected. A figure of around 96% for Ter_1 (as always) was the guess submitted in advance of the measurements. It should also be noted that the numbers quoted in figure 4.2 favour the experimenters for the following reason. The original test data caused the tool to fail, and the test data was only successfully processed after the tool had been significantly rewritten. This process caused the tool to become significantly smaller. Most of the code removed was infeasible.

Figure 4.3 reproduces some results from [38] in which the testing of three software tools is described. Again the coverage attained by the functional tests is poor considering that the test data was well understood. The attainments of the functional tests was clearly unacceptable to the software developers because testing continued in a structural testing mode. In this latter process additional errors were found. The environment was essentially text processing/system development.

Results obtained for standard D.P. programs in a Cobol environment are reported in [40] and also [39], which are summarised in figure 4.4 Again the overall picture is one of poor performance. In both environments the programs were mostly smallish and since the environments were accountancy and standard D. P. the functionality ought to have been simple and well understood.

Investigations with scientific software, specifically numerical analysis have shown improved results but nevertheless the programmers have been seriously concerned with the results [51]. In this case even a complete, unambiguous specification was available and yet the testers were neither able to

There is now a considerable body of functional tests which have been investigated by using structural testing metrics. Results have been obtained from all over the world and cover many different programming environments and implementation languages. Some have been controlled experiments, others have been 'a postiori' measurements where the test data has fortuitiously been available. That is, after the functional tests have been completed, the software and its test data are examined by structural testing techniques to obtain the coverage metrics.

The metrics which have been used are, either the coverage of the statements Ter_1 (the percentage of executable statements actually executed by the test data) or branch coverage Ter_2 (the percentage of the branches executed).

In figure 4.1 one of the earliest and most famous analyses appears [36]. The programmers in this case were not consciously using a functional testing strategy. Since they were using no other strategy it can be safely assumed that the choice of test data will have fallen into the functional test category. The main guidance the programmers will have had being a knowledge of the specification. Their environment was Fortran in the aerospace industry and hence it was probably a scientific calculation. The results they obtained are appalling. Interestingly it appears that in similar cases where the results have also been similar, the programmers upon being questioned were of the opinion that they had succeeded to a much greater extent. This intuitive feeling is confirmed in a small number of controlled experiments where testers have given their estimates of coverage in advance.

A more clear cut case is shown in figure 4.2. Here the investigators were specifically interested in knowing how good the functional tests could be in terms of structural coverage [50]. The environment was Fortran and Cobol in a text process ng/D.P. environment. A considerable effort was made to

CHAPTER 4

EXPERIMENTAL ASSESSMENT OF FUNCTIONAL TESTS

This chapter analyses the deficiences of current functional testing strategies. The method of assessment is the use of structural testing metrics. On the face of it, this is not a fair way to assess functional testing. What is needed is some metric or measure which will provide some guidance as to how well the functional tests are performing relative to the tasks of functional tests. Such a measure might be the percentage of the 'significant combinations' of functions tested. However the previous discussions have shown that currently this is impractical.

On the other hand some aspects of functional tests are worthy of consideration in a non-functional context. The first point is that a functional test ought to be a reasonably comprehensive test of the software, exposing it to many combinations of its working load. This suggests that most of the code should be executed by such a test. It is the fact that this does not happen very often that exposes current functional testing practices to criticism. The fact that a large amount of code is unexecuted by the functional test can be used to infer that there is a high probability of a very large class of errors remaining in the software. This is particularly true if it is the only testing method used. Since the users of the software will be able to make this inference without a significant degree of technical knowledge some other assurance would be needed. Where this assurance has been contractually essential, the technique has been to monitor the functional tests using structural testing techniques and then improve the testing by using structural tests until some chosen metric is satisfied as discussed below. Even in this case, it is accepted that an improvment in functional testing techniques is desirable.

The solution to the problems of informality is to ensure that all knowledge, detail, etc., needed to comprehend the text is present within the text. Some readers will skip this text as trivial and obvious and still make erroneous interpretations. The move towards data dictionaries can be a significant help in overcoming these problems.

3.6 Summary

In this chapter the authors, who are not linguists, have attempted to find rules and techniques for identifying functions. What has been found, in widely diffuse sources, indicates that humans have not found it necessary in the past to develop precise techniques for expressing ideas concerning functionality. No authoritative papers or books have been found which either discuss the problems or offer solutions.

The content of this chapter is entirely negative in the sense that it has demonstrated that a simplistic approach will not work. The problem of identifying functions and their interrelationships is highly complex and essentially very little fundamental research on this problem appears to have been published. The authors have attempted to produce some guidelines and validation procedures but these require vastly more work before being useful.

The first "it" refers to the comment but the second "it" could refer either to the "comment" or to "the program".

The usual signs of substitution are the appearance of pronouns such as

"it, them, that, those, one".

iii) Informality is the process whereby the writer assumes that the reader knows what is being referred to, thereby, utilising the fact that the language is context sensitive.

For example consider the following mathematical problem:

"What is the probability of being dealt a complete hand of spades in a card game".

An examiner setting this problem might not be aware that many students cannot solve this problem because they are unaware of what constitutes a card game or what is the process of dealing. Only those who have played card games with the 52 card pack of four suits are in a position to produce correct answers.

Informality is a very powerful weapon because it provides the mechanism whereby text can be terse. By not defining these terms, vocabulary, etc., with which the audience is familiar the text can be shortened considerably. It is argued that it is this technique which makes natural language text superior to mathematical text.

There is an obvious danger with informality, readers may believe that they understand the text but their interpretation may differ from that of the writer.

	Software tool	Stock control
INITIALISATION	3	1
HOUSEKEEPING	30	28
ALGORITHMIC	27	35
DIAGNOSTIC	6	1
	66	65

Figure 4.5

An analysis of the roles of modules in medium sized systems. A housekeeping function is one which performs a task which is in no way reflected in the requirements. It is introduced during the design stage as an artifact.

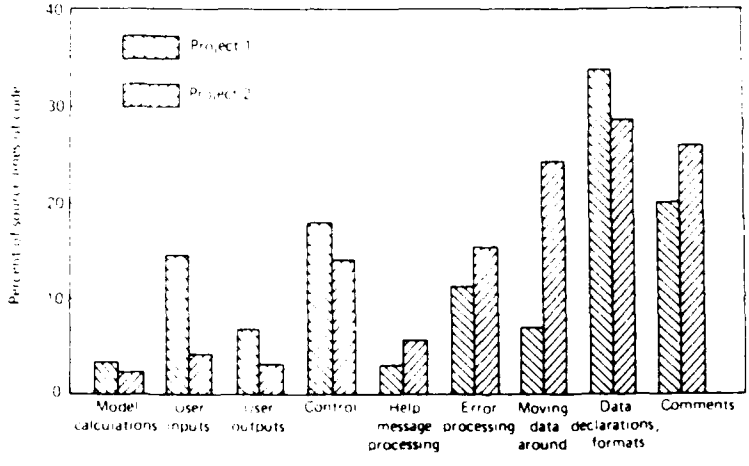


Figure 4.6

What does a software product do? An analysis of two implementations of the same specification.

Source: Boehm, B., "An experiment in small-scale application of software engineering". IEEE Trans. on Software Engineering, SE-7, 482-493, 1981.

```
read (i,j,k);
if j < k + i and i < j + k and k < j + i
then
    if j= k and k = i
    then
        print ("equilateral triangle")
    else
       if j = k or k = i or j = i
       then
          print ("isosceles triangle")
       else
          print ("scalene triangle")
       fi
    fi
else
        print ("not a triangle")
fi
```

Figure 4.7

Triangle program with no house-keeping code.

```
read (i,j,k);
if    i = j and j = k
then
          match: = 4
else
     if j « k + i and i « j + k and k « j + i
    then
         match: = 1;
       if  i = j then match: = match + 1 fi;
       if  j = k then match: = match + 1 fi;
       if  k = 1 then match: = match + 1 fi;
    else
         match: = 0
    fi
fi
if       match = 0 then print ("not a triangle")
else
    if   match = 1 then print ("scalene triangle")
    else
       if
         match = 4
       then
           print ("equilateral triangle")
       else
           print ("isosceles triangle")
       fi
    fi
fi;
```

Figure 4.8

Triangle program incorporating an algorithm to classify triangles.

CHAPTER 5

THE ORGANISATION OF FUNCTIONAL TESTING

The purpose of this chapter is to consider how functional testing can be organised in order to improve confidence in the success of the functional tests. Much of the motivation resulted from an analysis of code not executed by conventional functional testing activities.

Consider the possibility that the user perceived functions can be identified. If one of these functions is executed with appropriate test data, the correct results may be produced, e.g. in an airline reservation system, a seat request may be initiated which results in a seat being booked. The problem is how to assess whether this result was obtained by coincidence. For example, the reservation system may always reserve seats, regardless of whether seats are or are not available.

To resolve this ambiguity, there are two possible strategies; either to monitor the execution of the code by some form of dynamic analysis, or to organise the execution of a sequence of the User Perceived functions so that the success of the sequence indicates that each individual function is probably executing correctly. Note the careful choice of words, there is no certainty in functional approaches. To illustrate the second approach, consider the reservation system again. If a function is executed which defines a particular flight with a particular aircraft, then a specific number of seats will become available. Now, if the number of seats is N, then the first N requests for seats may be correct by coincidence. However, the failure of the N + 1 th. request provides a higher level of confidence that the request functions are working correctly. The fact that this N + 1 th. function execution yields, correctly, a different output to the previous executions

suggests that coincidental correctness is unlikely.

These two approaches to functional testing will be explored in detail.

MONITORING OF FUNCTIONAL TESTING

To obtain the highest confidence in the success of a functional test, the outcome must be seen to be correct, the stored data values must also be correct and the appropriate code segments should have been executed. The possibilities of coincidental correctness then lie solely within the computational processes of the actions. Elimination or reduction of this latter possibility is best accomplished by Structural Testing Techniques [35].

Determining which are the executed code segments can be accomplished at a number of levels:

i) By full dynamic analysis as required in Structural Testing Techniques.

ii) By selective dynamic analysis in which larger units than lines of code or Basic Blocks are instrumented. For example, module-calling sequences may be sufficient.

iii) By using diagnostic trace facilities. This possibility is similar to ii) above but also utilises the output of selected data items from which it is possible to deduce the execution sequence.

Consider the merits of these cases in turn. In the case of i), the cost of the dynamic analysis is higher than in the other cases. However, the detailed tracing which results from this can be used to document the tests to the extent that precise benefits of the test can be evaluated. The value of the test

for validating subsequent software modifications can then be assessed. This knowledge is particularly helpful during maintenance or enhancement activities. The major benefit of detailed tracing is that the corresponding structural testing metrics can be obtained, so that if these metrics are not satisfied to the required level, then structural testing techniques can be used when the functional tests are exhausted. The combination of monitored functional testing, followed by structural testing, is very powerful and has been used by many practitioners. It has the advantage of obtaining the best of both worlds. Both [37] and [38] describe how a poor functional test was enhanced by structural testing techniques with comparatively little extra effort.

The option ii) is significantly more economical, but correspondingly the information available is more sparse. There is greater scope for coincidental correctness and less accurate documentation of the test data coverage. Finally, the tie-up with structural testing coverage metrics is lost. This latter loss means that both techniques must be used separately with the possibility that both kinds of tests could be performing essentially the same tasks. However there is a considerable saving of resources because full dynamic analysis can be very expensive.

The possibility iii) can be used in a situation were dynamic analysis tools are not available. It has the consequence, however, that the software must contain a diagnostic trace mechanism as an integral part, and that the designers of this mechanism must have carefully considered whether the resultant information can be used to deduce which code segments have been executed. Automated analysis of the trace is highly recommended because humans tend to be poor at this kind of analysis.

Principles of Confirmatory Functions

In this section the exploitation of one function to provide confirmation of the correct functioning of others is considered. In order to accomplish functional testing by utilising the confirmatory power of the functions, two steps can be taken:

i) the assessment of the confirmatory power of the identified functions and the construction of the appropriate temporal sequence for their evaluation;

ii) the provision of special functions, not reflected in the user requirements, which are introduced purely to provide confirmatory power. An example is a checksum which is introduced solely to confirm that data movement operations are performed correctly. A checksum would not form part of user requirements.

These additional functions are termed 'validatory functions' and, in general, they will have a high confirmatory power. Frequently, the correct functioning of subsystems of the software project can be confirmed by the use of one or more validatory functions. This practice is commonplace with scientific programs, although there is no reason why this should not also be true of most classes of software. At the simplest level a diagnostic print statement is a validatory function, its role is to provide a means for checking internal data stores, etc.

It should, however, be noted that there is an inescapable run-time penalty associated with the use of these extra functions. This price is cheerfully paid for operations close to the hardware, e.g. checksums, but software engineers are wary of using them at higher levels. This is a sad waste of a powerful ally.

The organisation of the functions to determine their confirmatory power is not a simple exercise. Firstly it is worthwhile distinguishing between types of confirmatory functions. In order to be able to confirm the actions of functions executed previously, it is essential that these functions generate some outputs which are accessible to other functions. That is, the functions must have a communication mechanism. In a sequential system this communication will be through the use of data stores. The concepts of data stores are widely used in a number of design methods, such as Yourdon [20], Gane and Sarson [12] and the MASCOT method [22]. A data store is a place where items of data are retained for future use. This should not be confused with the use of temporary variables introduced for optimisation or algorithmic purposes.

These data stores may be internal or external. An internal data store resides within the software system's data storage areas, whilst the external data stores are contained in files, data-bases, etc.. An external data store implies that the temporal sequence of function executions can be across many separate runs (executions) of the software system. An internal data store implies that the temporal sequence of function executions must be within the same run, because with an internal data store the lifetime of the data is purely that of the current execution.

In the discussion it will be assumed that an instance of a function execution has the following properties:

1). A predicate (or guard) which must be satisfied before that instance of a function can be evaluated. This predicate is dependant on the values of the data stores or is null (true). In the airline reservation system this is to perform N + 1 executions before the confirmatory power of the function becomes a maximum.

2). An action which causes the values of the data stores to

be either defined (i.e. changed) or referenced (i.e. used in a computation or an output).

These functions may involve other predicates and actions which have no effect upon the data stores in question (ie. effect output only). It is assumed that these outputs will be checked independently of the testing activity described here.

An exact definition of a function is not needed in this context because it is assumed that the functionality of the software system is resolved before the organisation of testing is considered.

The justification for associating guards with instances of functions follows from the need to order the execution of the functions so as to achieve maximal impact. The guard then is dependant on the contents and properties of the data stores and the particular actions of the functions. This guard is not necessarily reflected in the actual implementation of the function.

An example of the use of guards can be seen with respect to the organisation of functional tests for the airline reservation system as shown in figure 5.1. This figure models the sequences of the following functions:-

f_0 - initiates a flight by creating files and initialising values.

f_1 - is a function which enquires into the availability of a seat.

f_2 - is a function which books a seat. It is assumed that it is always preceeded by f_1 from which vital information (to the booking clerk) is gained.

f_3 - is the function to obtain a standby reservation.

f_4 - closes and dispatches the flight.

f_5 - cancels a booking of either type.

The nodes of the graph contain three fields, one contains the guard (or null where the guard is always true), the second contains the function name and the third contains the action (or null) performed on the contents of the data stores. The arcs of the graph are directed and indicate the temporal sequencing.

In this graph the function f_1 appears twice but each instance has a different guard. The reason for this is that by separating out the two different roles for the function its ability to confirm previous function executions is substantially increased. On the other hand the function f_5 also appears twice, this is purely to simplify the graph.

The property of these functions which is of interest here is their confirmatory power which is defined as follows.

1) Type A functions with zero confirmatory power. These functions do not reference or define elements of the data stores, i.e. they have no guards or access to data stores.

2) Type B functions with weak confirmatory power. The references to the data stores occur solely in the guards.

3) Type C functions with high confirmatory power. They contain references and definitions for the data stores in the actions but not the guards.

4) Type D functions with strong confirmatory power. Both the actions and the guards contain references to the data stores and the actions may also define data store values.

Note that the first node of a graph cannot have any

confirmatory power regardless of its access to the data stores. Thus from figure 5.1,

f_0 is type A,

f_1 is type B,

f_2, f_3 and f_4 are of type C,

and the function f_5 is of type D.

The construction of such a graph will be covered in the next section.

CONSTRUCTING CONFIRMATORY FUNCTION SEQUENCES

To perform a test using confirmatory function sequences it is necessary to obtain the temporal sequences of User Perceived functions which will cause the confirmatory functions to be executed in the appropriate order.

The steps involved in the process are as follows.

1. Identify the various data stores and select those which will form the basis of the test. Find those functions which reference or define values in these data stores. This step is usually fairly straightforward. This is particularly so in the case when the design contains explicit references to data stores. It is also easy to do for an implementation because the information can be obtained with the aid of a cross reference. The functions so detected need to be assigned the appropriate value for their confirmatory power.

2. The temporal sequence of these confirmatory functions

must then be deduced. The construction of the guards gives helpful insights into the number of different instantiations required for each function. The hint to include more than one instance of a function is usually indicated by the number of successor functions. If there is a selection between these successors and this selection depends on the contents of the data store then repeating the function with this condition in the guards is indicated.

The temporal sequence charts such as that shown in figure 5.1 are worthwhile constructing. This is because there will usually be one per data store, since in general functions which access one data store will not access another. These charts are then analysed to produce execution sequences of the confirmatory functions. The rules for producing these sequences are as follows. Firstly self loop entries involving only type B functions can be removed. Then paths connecting the confirmatory functions need to be constructed so as to maximise the confirmatory power of the sequence. Whilst this step may appear difficult, in practice the interrelationships of these functions is not high, even in large software systems.

3. The final step is the hardest. It is to relate the confirmatory functions back to the sequences of User Perceived functions. In the examples examined by the authors this has not proved to be too difficult but in general this may not always be true.

The problem of relating these functions backwards to the User Perceived functions depends enormously on the design method in use. The problem arises because to date the inventors of design methods have given little attention to the needs of testers. It is of course also true that testers have not made their needs clear to design theorists.

USES OF CONFIRMATORY FUNCTIONS

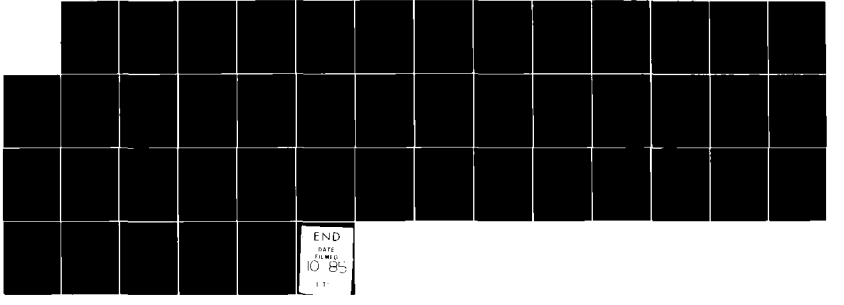
Housekeeping functions which were introduced earlier can now be seen to be closely related to the concepts of data stores. In general but not always these data stores are not reflected in the requirements, they are artifacts introduced at the specification and design levels. The operations performed upon these data stores are usually encapsulated into the housekeeping functions. Frequently however these housekeeping tasks are interleaved into the routines implementing User Perceived functions. This frequently occurs as the result of optimisation.

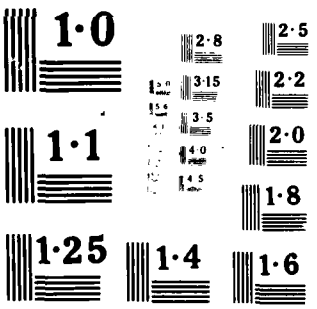
It is clear that exploiting the confirmatory power of functions in general will result in the exercising of the housekeeping functions. Moreover the confirmatory power of a given function can be emphasised by separating out its differing roles as for f_1 in the seat reservation system. In this way the more pathological possibilities can be explored.

When the functional sequence chart has been constructed it may be found that certain data stores have very few functions of high or strong confirmatory power associated with them. Or it may be felt that the accuracy of the contents of these data stores is of particular significance. In this case validatory functions can be introduced. They will perform some analysis of the contents of the data stores which will either yield confirmation or partial confirmation of the correctness of the contents. This may be on a sampling basis or each time the contents are changed. For example a random parity check could be made on a data file, whereas a model problem inserted into a large scientific calculation yields high confirmation over the whole run.

A validatory function should not alter the contents of data stores, it should merely reference the contents either in guards or preferably in its actions. An example of a validatory

AD A158 104 PRACTICAL ASPECTS OF FUNCTIONAL TESTING TECHNIQUES(U) 2/2
LIVERPOOL UNIV (ENGLAND) DEPT OF STATISTICS AND
COMPUTATIONAL MATHEMATICS M A HENNELL ET AL. 31 JAN 85
UNCLASSIFIED DAJA37-81-C-0736 F/G 9/2 NL




1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
1·8
4·5
1·1
1·25
1·4
1·6

function which references the data stores through the guards is error detecting code which prints out an error message when, say, a data store overflows. Here the contents of the guard are explicitly present in the code.

A more powerful validatory function would have been one which by monitoring the data stores was able to predict the overflow and retain a diagnostic trail or execution history from which an accurate diagnosis could be made. The function would have referenced the values of the data stores in its actions.

Another example of validatory functions lies in the use of assertions which are essentially predicates inserted into the code at key points. These assertions are type B validatory functions which report when a violation of their predicate occurs. Such techniques have been used in testing tools for many years [36].

SUMMARY

The technique advocated here is that the design (or code) should be analysed for the occurrence of data stores. The functions (or modules) which access these data stores should be obtained. Then the appropriate guards for instances of these functions should be deduced. This may require an analysis of the properties of the data stores and the successors of the functions.

The confirmatory power of these functions or modules should be considered. If it is found that the contents of individual data stores are not confirmed to the required extent, then additional validatory functions (modules) should be added to the software.

Finally paths through the functional sequence charts should be obtained which maximise the confimatory power. These paths then

indicate the ordering for the execution of the functions in a test run.

The benefits of the method are that the functional tests can be organised so that the risk of coincidental correctness is minimised. Further, the problem of test outcome assessment is minimised because the correct internal functioning of the software is checked automatically. In some sense these functions perform the tasks of an automated oracle [31].

CHAPTER 6

SUMMARY AND CONCLUSION

The general thrust of this research project has changed direction a number of times in response to difficulties encountered.

The original thesis was that functional testing under achieves and that this claim was demonstrable by assessing these achievments by means of structural testing methods. The authors believe that this point has been made convincingly by the extensive results which have been reported both formally (in published papers) and informally (at conferences and specialist meetings).

Using the acceptance of under achieving as a starting point the question to address was what further steps should a functional tester take in order to improve the activity. It was anticipated that a careful study of those areas in which functional testing was deficient would indicate ways in which an improvement could be sought. This has been found to be partly true although perhaps in a wider context than originally thought.

The study of the unexecuted code has shown that a considerable proportion of this code is implementing functions which are not directly related to the external or User Perceived functions and therefore targetting such external functions towards the unexecuted code is unlikely to be successful. There is, however, considerable scope for targetting at least some of the functional tests towards the remainder of the unexecuted code. To this end the report describes the method of confirmatory functions which actually contributes to the execution of code implementing both User Perceived functions and also housekeeping functions.

The next stage in the research was to relate areas of unexecuted non-housekeeping code back to the user perceived functions. In order to do this it is necessary to be able to recognise a function implementation either directly from the code or from its external viewpoint. Our analysis shows that from the directed graph representation of a program there appear to be certain structures which are consistently missed in the choice of functional tests. The next step was clearly to detect the corresponding items at the design level in order to move closer to the external view. At this stage, despite a detailed analysis of the major design methods, no clues have been uncovered and hence this approach has failed.

Despite the failure of the analysis to produce any significant benefits to functional testing the results look very promising for application in structural testing methods and hence are not included in this report.

In studying the design methods it was found that great difficulty arose from confusion with terms. This led us to attempt a systematic analysis of the use of these terms and to produce generalisations which incorporate individual uses. One result of this is the generalised life-cycle incorporated in the appendix. This life-cycle is an incorporation of all the life-cycles which have been found in the literature, it is not a new life-cycle as such. Its relevance to this project is purely that of a useful tool. A detailed classification of which parts of this life-cycle are covered by the design methods is only partly complete and hence not covered in this report.

Many of the design methods examined offered suggestions as to how functions could be detected in requirements documents. Naturally these suggestions have been followed up in detail and the results of our investigations are fully reported in this document. The conclusion reached, however, is extremely disappointing. It does not appear that functionality is

derivable from natural language text in a simple way. That is not to say that functionality cannot be so deduced, we believe that it can, but that either the techniques for obtaining this functionality must be highly sophisticated, or very great care must be taken in the writing of the documents. Our investigations have indicated that certain possibilities might be worth investigating and these will be pursued under other research contracts.

In summary then this research has produced three substantial contributions to knowledge:

1. The development of confirmatory function techniques. This method can be developed to a much greater extent when design methods are adapted to yield a suitable viewpoint.

2. The evidence that current concepts of functionality are completely inadequate and hence methods which rely on such concepts are themselves inadequate. This alone has enormous consequences for such concepts as top-down functional decomposition.

3. The generation of the life-cycle and introduction of new definitions.

Other contributions have also been made in that a number of topics for future investigation have been uncovered.

The work on confirmatory functions has been reported at a Workshop on Specification, Design and Testing Techniques whose proceedings will be published shortly. A paper on the identification of functionality in natural language text is in preparation. Finally the generalised Life-cycle will be reported to the IEEE working party producing a standard for the Life-cycle, and will also be published separately.

"System Development Life Cycle"

Introduction

The term "System Development Life Cycle" is used to identify the main phases which occur in the development of a software system from initial conception to final realization. A glossary of terms used in this document is included at the end.

There are several System Development Life Cycles, sometimes called System Development Processes. These Life Cycles are used by various System Development Methodologies as an integral part, mostly without definition, e.g., SASD [5,18], LSDM_LBMS [16], NIAM [14], HOS [46] and SADT [11], etc. There are also some System Development Life Cycles defined by Enger N. L. [55], Biggs C.L. [56], Blum B. [57], Ramamoorthy C. V. & Ma Y.N. [58], O'Neill D. [59], Freeman P. [60], and many others. These Life Cycles have several problems in common, for example, not covering the whole span of System Development. They are suitable only for certain types of problems and have lengthy phases which may increase the probablity of committing errors. Their lack of by_pass options for the separate phases may increase the overall costs for particular projects, after all, unnecessary phases will cost money. The most critical deficiency is the lack of knowledge about the Requirements of the User/Management, since they do not in general have phases devoted to this problem.

In real world problems it is a common practice in developing any system to start from some vague idea, some proposal, some 'wants'. At the beginning of a development the actual requirements are usually incomplete, at least for a short period and possibly for ever in a complex system.

In computer software systems, it is a well recognized problem that users of software frequently cannot state clearly and

completely what they want until they see a finished system running. Due to this incomplete knowledge of 'wants' and their specification a large number of problems solved by computers are initially ill-defined.

Some explanations for this inconsistency as mentioned above are the present dominating, but inadequate System Development Life Cycles. Mostly thcurrently Cycles are based on the assumption that the user can express his/her 'requirements' in a complete and comprehensive way. It has been shown [1], that the most frequently occuring problems arise through (a) incorrectly specified requirements, (b) inconsistent and incompatible requirements and (c) unclear requirements. Past experience reveals that the user's requirements cannot be stated fully at the beginning because the User/Management are unable to articulate their wants and expectations completely. Further, they cannot forsee clearly the directions into which further requirements will develop. "Requirements specification languages are supposed to enable the developers to state their understanding of user's ideas in a form comprehensible to the user.....but the user himself has only very vague ideas of what he/she wants" [2]. The study of present System Analysis and Design Methodologies reveals that these methodologies do not cover the whole span of the System Development Life Cycle in the way we generally solve other problems. Furthermore they do not contain clear definitions of the starting and end points of their phases. Peter Freeman points out that "repeatedly I encounter life cycles in use that do not fit the 'natural' flow of work that should take place for a given application" [3]. In present methodologies there is no check to ensure that the user understands his/her problems and further that what he/she 'wants', really will solve his/her problems. There is no mechanism by which to enquire why there should be a new system, what this system will actually do and how it will do its tasks.

Different Systems Development Life Cycle models given by these methodologies have several common features and steps, and use

the same terms but for differing concepts, and frequently different terms for the same concepts. These terminologies and freqently used common words are used carelessly and without proper definitions. This causes communication problems especially when two different methodologies are used in the same organization. For example, the term **ACTION** is used in ACM/PCM [23] to update an **OPERATION** on an information base. The same term **ACTION** is used in JSD [21] to represent an **EVENT** in which one or more entities participate by performing or suffering the action. The term **EVENT** in LSDM-LBMS [16] is used for **PROCESS**. In SADT [11] the similar term **ACTIVITY**, identified by a verb in a natural language, is used to represent a happening, which may be performed by a computer or people, etc., in a system. In NIAM [14], and ISAC [25], the term **ACTIVITY** is used for an **ACTION** taking time and resources. In MASCOT [22], the term **ACTIVITY** represents a process. The term **ENTITY** in D2S2 [28], LSDM_LBMS [16] and SYSDOC [27] is used for an **OBJECT** which is actually occuring as a concrete thing of interest in a system. The same term in JSD is used for an **OBJECT** in the real world which participates in a time_ordered set of actions ie. entities perform and suffer actions. In SADT, the term **DATA** is used to represent **OBJECTS or THINGS** working together to perform functions in a system. Furthermore in ACM/PCM, ISAC and NIAM, the term **OBJECT** is used to represent an **ENTITY**. The term **FUNCTION** is used in SYSDOC in a general sense, i.e., to represent any mathematical or business function, where as in JSD the term **FUNCTION** represents an action or set of actions performed by the system and resulting in the production of output. **FUNCTION** in MASCOT represents an action. These methodologies start from a phase which identifies the problems to be solved and give a solution for the problem with the assumption that output will satisfy the user's requirement.

The above mentioned problems compelled us to develop a new "System Development Life cycle", the aim of which is to build a global framework which can help to develop a system in a way which is similar to that which is used to develop systems for

solving other problems. It is an amalgam of all cycles currently found in the literature. All such cycles are incorporated into it on the grounds that they are probably suitable to at least one class of problems. This Cycle consists of a set of phases, which are generalised. It will be appropriate for all classes of problems and different environments. It is not restricted to any Software Design Method or Methodology. This Cycle starts from the project proposal, i.e., the user's 'wants', not from 'Requirements'. There is a difference between 'wants' and 'Requirements' [4]. The 'Requirements' should ideally be targeted towards the user's 'needs'. The Cycle attempts to identify the true 'wants' and gives a better understanding of the problem. The major purpose of this System Development Cycle is to help the Analyst and the Designer to analyse, design, implement, and maintain a system in an evolutionary manner. This Cycle therefore covers all aspects of system development such as strategy, analysis, system specification, design, implementation, transition, maintenance and Verification & Validation. It is based on the active participation of the user as well as operational personnel (if there are any), who can help the Analyst in understanding clearly what they (the users) 'want'. This approach provides a conceptual framework for developing a system by using existing powerful and widely used Design Methodologies and allows the use of established programming methods such as JSP [61], Top_down, Bottom_up [62, 63], etc.

This Cycle gives clear but flexible phases which can be bypassed or modified according to the environment and type of problem. In a sense it is a global but at the same time standardized System Development Life Cycle. It offers guide lines and gives some ideas on how to proceed when developing and implementing a System. Any step can be by-passed if found unnecessary. Iterative processes and prototyping are regarded as integral parts of the Cycle.

The Life Cycle

In the following the first figure in the heading is the principal phase number, the second is the subphase number. The star indicates the phase name.

1 *Project Proposal Phase

Sub Phases

* Study of Proposal
* Initial Investigation
* Initial Feasibility Study Report

Before starting any System Development activity the User/Management will first have to consider how such a system might improve their work and make extensions in their organization. They then prepare a proposal in this regard. In this proposal they may 'want' to have some modifications in the existing system or to have a complete new system which may or may not be computerised. Usually such a proposal is prepared by the management in consultation with the user. It gives some description of the project and the objectives to be achieved, together with preliminary estimates of manpower and cost involved.

1.1 Study of Proposal

The main objective of this step is to study the proposal given by the management. There is no doubt that management and user always face difficulties, as in other fields, in expressing correctly and precisely what they 'want'. In fact in most cases they are not clear themselves what is their exact requirement. They either propose something ambitiously, thinking 'bigger is better' or some times, underestimate the problem and hence their needs. In this phase the Analyst will study the proposal, recording possible benefits that can be accomplished with it,

taking into consideration the size and working capacity of the organization, budgetary and manpower constraints, annual recurring systems, operating, maintenance and training costs which may be involved. The Analyst will try to understand at an initial level what the management/user actually 'needs'.

The possible outcome of this subphase will be in the shape of some comments or notes recorded by the Analyst for his/her own convenience, better understanding and memory.

1.2 Initial Investigation

Objectives of this subphase are:.

1. to study the most frequent problems likely to occur in the system, in the User's/Management's opinion,
2. to investigate the proposal and check for correctness, completness and redundancy of wants,
3. to find out possible benefits or disadvantages that can be obtained from the Proposal,
4. to review 'wants' and seek out actual needs and objectives because it is likely that the proposal will not cover the actual 'needs',
5. to select from the proposal any radical changes that may positively effect the working of the organization and decrease costs.

The Analyst assigned to study the proposal will discuss it with User/Management in a number of meetings to achieve these steps. The output of the Initial Investigation subphase should be a statement postulating actual 'needs' including preliminary identification of cost & benefits.

1.3 Initial Feasibility Study

The basic purpose of this step is to find out possible ways to implement the output of the Initial Investigation step, i.e., the user's 'needs'. The Analyst has to collect basic information that will be required to implement these 'needs'. The Analyst will then write his/her recommendations about the selection of a particular Analysis Methodology such as SADT [11], SSA [12], PSL/PSA [13], NIAM [14], STRADIS [24] and LSDM_LBMS [16], etc., suitable for this particular problem. The selection of a particular Methodology or Method is not an easy job and, "it is often difficult to determine whether a given methodology applies in a given situation. It is even more difficult to select one methodology from among all the ones that might be used" [17]. It is not possible to say that a particular Methodology or Method is superior to others in an overall sense even though it is better than others in a particular situation. Therefore, the criteria for the selection of a particular Methodology or Method or a combination of them, should depend on the type of problem, the environment and working area and finally the personal experience of the Analyst. Furthermore the Analyst will also write a brief plan expressing estimated cost, manpower required, tentative completion date, equipment required and finally the potential benefits from these changes. The deliverables of this step will be used as a basis for setting detailed objectives and developing strategies for further System Design. This report will be submitted to the User/Management for consideration.

2 * Strategy Phase

Strategy means a proposed set of actions clearly designed to achieve certain well defined goals over a period of time.

In the domain of Systems Analysis and Design, strategy means choosing between practices, guidelines, recommmended sets of actions and the ordering of development decisions. In general

cost of the system.

5 * Design Phase

Sub Phases

* Planning and Decision
* Conceptual Design
* Physical Design

"Design is the process of transforming what has to be done into a means of doing it" [18]. In this Phase emphasis will be given to the application of the System Specifications, rather than understanding the User's 'needs', and 'requirements'. "Design is concerned with what might be in the future if the Design be implemented, and hence is concerned with what is not yet or does not yet exist" [19]. The Design Phase starts after finalising the System Specifications which are in fact a plan for the detailed proposal expressed in technical terms, and must preceed and not be confused with implementation, i.e. its scope is up to implementation point. The Design activity, if performed carefully, converts this plan into an unambiguous model, (although there may be more than one model). It involves devising ways which lead to the attainment of the required goals, i.e., to solve the problem in a way which is according to the proposal and its Specifications and satisfies the User/Management. Usually experienced Designers have certain assumptions, beliefs and tricks and they develop a Design with the help of this known-how, i.e., their past experiences. They use these as a theory consciously or unconsciously. Such assumptions, beliefs and tricks are usually correct and worth using, if the resulting Design is passed through well prepared testbeds. The recent study of System Design Metholodogies revealed that in general there is no methodology or tool which is suitable for building a system for all different types of problems and environments and is capable of satisfying or likely to satisfy different user's needs, requirements and corresponding specifications. Any Design problem can be solved

boundaries, system interface, etc.,

3. to specify details about processes, functions and operations of a system,
4. to specify provisions for error handling and recovery thereof,
5. for use as a standard against which an implementation can be verified,
6. for use later to support system maintenance,
7. in some cases for use in building a prototype.

The quality of a system depends to a great extent on the way System Specifications are developed. In this Life Cycle, System Specifications are developed from a selected detailed proposal, i.e., output of Analysis phase-3. In the System Specifications phase the Analyst/Designer elaborates the proposal into a set of different clear steps which are used as a basis for system design. This elaboration can be based either on functions, processes or data, and some times on both. It may lead towards one or more concepts of system design such as Structured Design [18], Top-down method, Bottom-up method [62] [63], JSD [21], MASCOT [22], etc., as the case may be, according to the type of problem. It is, therefore, recommended that System Specifications should not be described in accordance with a particular design methodology. Again here at this point we are not ranking one design method above an other. These Specifications, we consider as the foundation of design, therefore, it is essential that user and management must agree and confirm that these Specifications represent their requirements and serve their purpose. This will help and encourage the Designer in making further steps.

Many implementation and maintenance problems are due to incomplete Specifications. Incompleteness in Specifications implies that the Analysis work and its output are incomplete and, therefore, may require a return to the Analysis phase (phase-3). Paying sufficient attention during the development of System Specifications can result in reducing maintenance

emphasize management and documentation aspects. These previous methods are not widely used in industry. Thirdly, and most commonly, there are specifications expressed in natural language.

The System Specifications phase represents an interface between Analysis and Design phases. This phase separates completely the Design work from the Analysis and refers to that information which must be delivered from the Analysis to the Design phase. It converts the terminology and pragmatics of the application area into the terminology and pragmatics of computing. For example a User's requirement that a system should be reliable to a given extent must be transferred into constraints which are relevant to computing, such as appropriate values for reliability metrics. Another example would be that a requirement for completion of a document for a particular task would lead to a precise definition of the contents of that document in terms of the inputs to the computer system. This phase separates the User's/Analyst's view of the problem, and its solution, from the Designers. Without such an interface it will be difficult to know how to represent the output of the Analysis results technically. Further such an interface helps to determine, up to a reasonable extent, that proposed changes will not have any adverse effect on the present system (if there is one). This interface will also help in providing flexibility in system design, enabling the Designer to execute easily the changes which may occur from time to time. System Specification can be considered as a basic source from which alternate design decisions can be made.

The main purposes of System Specifications are:

1. to express in precise and consistent form what the system will do, according to the User's/Analyst's proposal,
2. to define the objectives and constraints which the system must satisfy, e.g., system

system, and possible effects if the old system is continued.

Systems required purely to achieve certain technical objectives may have unpredictable human consequences because technical decisions may cause changes in policies, strategy, environment, status of personnel, and organization. These changes may result in serious industrial relations problems. It is, therefore, necessary that the Analyst in the feasibility study and detailed proposals should avoid chances of developing human aggravation and provide a way to gain the user's confidence.

Sometimes changes may occur during the development process which cannot be ignored. These changes may be due to the time factor and a better understanding of the requirements, or earlier communication failures and their correction. It is recommended to submit more than one complete proposal with the Analyst's recommendations, and conclusions, giving summarized statistics about, cost, time, etc. and comparisons between the present and the new system and the possible benefits thereof. These proposals should be discussed, before any move, by User/Management or a committee on their behalf. They will decide accordingly which proposal should be accepted and approved for further progress. However, if User/Management or their committee is unable to decide and does not agree with any proposal then it will be essential to repeat the Cycle, do all the excercises again, and submit new proposals or otherwise terminate the project.

4 * System Specifications Phase 4

System specifications should state clearly and unambiguously, what a system must do. It is a statement of the requirements in technical terms which are meaningful in a computing environment. There are three main description methods for system specifications. Firstly the use of formal notation methods, such as HOS [46], AFFIRM [54] and VDM [45], etc.. Secondly methods like PSL/PSA [13] and SREM [15], etc., which

3.6 Feasibility Analysis and Detailed Proposals

Feasibility Analysis means studying how the Requirements Specification can be implemented within the given constraints such as technical, operational and economic factors. It is useless and painful to design a system which cannot be implemented under given constraints. In this subphase the Analyst will prepare a detailed report, the purpose of which is to give Management:

1. a detailed review of the requirements and Requirements Specification and surety that there are no redundant factors and that its implementation is feasible,
2. the expected implementation, running and maintainence costs of the new system,
3. a detailed plan with which to develop, implement, and test the new system,
4. estimates about completion time, staff needed and hardware required,
5. cost benefit analysis for the new system,
6. information about essential changes in policies environment, status of personnel, organizational changes and effects thereof.

The Analyst should select first those requirements which may cause major changes in the existing system (if there is one), and are relatively expensive to implement. On the basis of this selection the Analyst will recommend whether it is advisable to implement those requirements or not. This report should also justify the implementation of the rest of the requirements for the new system, keeping under consideration the implementation, running, and maintenance costs against possible benefits which management may get from the new system.

The Analyst should inform the management in the report about the expected results which may come after implementing the new

3.5 Search for Existing Available System

Shortage of skilled and experienced Analyst/Designers and their cost of employment is becoming a serious administrative problem. For example, it is estimated that up to 90 percent of the data processing intellectual effort in a large corporation is devoted to developing and maintenance of Software [64]. This problem can be eased by making use of available proven systems from the software market. This involves detailed searching of software banks/catalogues to pick up a system suited to the Requirements Specification, or otherwise modify one of the software systems which is similar to the required one and involves a minimal degree of changes. In such an approach the Analyst has to make sure that the system which is being selected is sufficiently free from bugs, has sufficient good documentation and is acceptably user friendly etc. This information can be obtained in a number of ways, e.g. by submitting a number of questions to various establishments using the chosen system. After this selection the Analyst can move directly to the implementation step by-passing all intermediate steps of the cycle which are discussed below. Before this the Analyst has to satisfy the User/Management that the implementation is consistent with the Requirements Specification. Purchasing systems in the above manner will cost less money and time than creating it by oneself because the development cost of a system available commercially is spread over a number of purchasers.

It may be that a system can be found which is close to the required system. In this case commercial consideration may force the omission of these missing requirements. Clearly some considerable thought must be given to finding these missing requirements and then studying the implications of these omissions.

Maintainability

Inevitably errors will be introduced in the final product which have not been trapped during the previous stages. If it is important to the user that the system be corrected then the ability to maintain the system will be a requirement. Transporting the system to other machines may also be a part of this requirement. The criteria by means of which this requirement can be enforced must be considered and implementation anticipated.

3.4 Check for consistency

By consistency we mean to examine critically in order to locate any contradictions in the Requirements Specification which may be due to:

1. a difference between the Requirements Specification and what the user 'intended',
2. modification of the Requirements Specification, which may happen from time to time,
3. contradiction and incompatibilities between requirements,
4. missing components, undefined entries, etc..

After every modification care must be taken to ensure that consistency is maintained. For these reasons it is essential to have frequent consistency checks. These checks should be made before moving to further steps of the System Development Life Cycle, especially before searching for existing available systems and writing feasibility analysis and detailed proposals. Any error at this level may cause many problems in future steps. To eliminate contradiction at this stage is not an easy job and becomes more difficult when the system being modelled gets bigger and more detailed. This problem can be greatly simplified if a detailed cross_reference is available.

omissions in the Requirements Specification and to develop them in such a manner that they are understandable. The understandability of Requirements Specification can be obtained by imposing constraints on the style of writing in natural language and using diagramatic help.

Reliability

A Requirements Specification having a high probability of being complete, correct and consistent and which can lead towards successful development and implementation of a software system over a period and under predefined conditions, is called a Reliable Requirements Specification. Unreliable Requirements Specification leads towards uncertainty as to whether or not the system can be successfully implemented. To get a reliable specification it is essential to verify consistency between Requirements and Requirements Specification. This can be done, as mentioned earlier, by using the participative method of systems design in which the Requirements Specification is shown to the user to avoid complications, chances of misunderstanding and to reduce discrepancies, before implementation.

Evaluation

The Requirements Specification should reflect the user's requirements which may not be static and may change whenever the user wants. These changes may be due to improvement, elaboration, specialisation and generalization. It is, therefore, necessary to develop a Requirements Specification which is evolvable. In our System Development Life Cycle any changes or modifications should compel the Analyst/Designer to go to the start of Phase 3 and repeat the same excercise. It will be easier to do this as the previous skeleton or model for the system can be re-used.

4. to specify initially the data, processes or functions which might be required in the system,
5. to act as a basis for system assessment.

The preparation of a Requirements Specification is a complex and time consuming process. Its success depends to a great extent on the quality of the Requirements Analysis. This job needs high quality, intellectual, creative and experienced Analysts.

The output of Requirements Specification must be expressed in a clear and comprehensive report. This report will be used further as a basis to prepare detailed proposals for the new system and can either be prepared by using a specification language or a natural language supported by diagrams.

A Requirements Specification having the following properties can be considerd to be of high quality.

Feasibility

Requirements Specification which are developed in such a way that they are capable of being implemented successfuly and are according to the agreed needs and constraints mentioned in Phase 1, are said to be feasible. It is difficult but important to develop such Requirements Specification. Lack of feasibility may cause serious implementation problems and hence increases cost and frustration.

Understandability

Requirements Specification which are developed in such a manner that they are clear, unambigious and difficult to misinterpret are said to be understandable. During the process of software development some Requirements Specifications are ignored intentionally or unintentionally by the Designer because of these problems. It is, therefore, necessary to avoid errors and

overwhelming in industry regarding the dramatic consequences of incorrect or inefficient problem definition work. The causes are misunderstanding of the problem and insufficient analysis to decompose the problem" [10]. Some essential properties of the Requirements must be:

Completeness,
Consistency,
Unambiguity and
Modifiability.

3.3 Prepare Requirements Specification

It is difficult but desirable to have a clear definition of the Requirements Specification. The problem in formulating a clear definition is that different authors have different concepts about 'Requirements' and 'Specification' and hence of 'Requirements Specification'. We are proposing a definition which is a general one. To specify is to state particularly and precisely. Hence a specification is a particular and precise statement of something. A requirement is something which is deemed to be needed. A Requirements Specification is therefore a particular and precise statement of those things which are deemed to be needed. Note that this perception of needs is from the viewpoint of the user, their management and environment.

The main objectives in preparing a Requirements Specification are:.

1. to define clearly what the system or part of it must do in accordance with agreed requirements, i.e., the output of the subphase for Requirements Analysis,
2. to help in transforming the requirements into a system specification by providing a well defined starting point,
3. to act as an initial model which can exhibit, as much as possible, relevant information for forming a design,

understanding the problem and of communicating that understanding among the concerned individuals and organizations" [7]. In short, Requirement Analysis encompasses all aspects of the system and yields its specification. Its main objectives are:

1. to decide clearly whether or not those identified needs will actually be required,
2. to collect further information about 'needs' and formulate its structure,
3. to consider the social, legal, security, privacy, managerial, and, financial impacts which may come as a result of fulfilling the above 'needs'. These aspects may impose additional requirements on the system.
4. to consider the environment where the system will work,
5. to communicate all the above information to User/Management.

A part of this work can be done best by using the "Participative approach to System Design method" [8] [9], which gives emphasis to the need for meeting the human needs of staff when designing computer systems. This method is supported here because the user is likely to be the most appropriate person inside the organization, who may know something about the 'needs' and predict any further changes in these user 'needs' which may possibly occur during the process of system development. This is because the 'needs', may not be stable throughout the entire system development process. The process of Requirements Analysis developed in the above way can help both user and Analyst to learn from each other and thus to write a better analysis report.

The product of Requirements Analysis must be a clear statement of the user's requirements, some sound results from which the Requirements Specification can be built. "The evidence is

mentiones that "A high quality of system design will be very difficult to produce unless systems Designers have an adequate understanding of the industry in which the organization does business .... (and the) technical environment in which the system must be implemented" [6]. Unfortunately in present existing cycles too little emphasis is given to the importance of collecting information about the environment in which the system will be working.

Objectives of this subphase are:

1. to collect and understand information about the environment in which the system will work,
2. to know about the application area for which the system will be designed,
3. to develop a dictionary of all different terms, concepts, notations and definitions which are used in the establishment, in order to avoid misinterpretation of words and communication problems,
4. to investigate factors which may effect the system environment, i.e., factors which are responsible for the occurrence of incorrect data, creating inefficient processes or functions, and
5. to investigate factors which cause the system to be less portable, create security and integrity problems, etc..

The output of this subphase is a small report showing adequate information about the environment and working area for which the system is being designed. This report will be used by the Analyst in subphase 3.2,i.e, Requirements Analysis.

3.2 Requirements Analysis

"Requirements Analysis deals with the difficulties of

(if it exists) or an idea for a new one into its components in an attempt to reveal and examine how those components work and interact with each other in order to accomplish the purpose. It is a process to show 'what is in' the system and 'how it works' within the user's environment. "In the specific domain of computer system development, Analysis refers to the study of some business or application area, usually leading to the specification of a new system" [5]. The term 'Analysis' is used in our System Development Life Cycle for the purpose of analysis of 'needs', (obtained from phase 1), in order to identify and find the causes of the problems and user's requirements. The result of this analysis leads eventually towards detailed proposals and their feasibility for developing system specifications. In these proposals the Analyst develops a definition of the Requirements and Requirements Specification, important parameters, and the environment in which the system will work. Emphasis should be given to the Analysis phase in order to get a better understanding of user's requirements and expectations.

3.1 Analysis of Environment

In general, no Designer designs a system badly by choice, but at present many systems which are working in different environments are either working badly or are total failures. One of the possible reasons for that is that both Analyst and Designer are unaware of the problem area and its environment.

In the process of System Development the Analyst often does not have much information about the user and the environment for which the system will be designed. Hence an outcome of the Development process may be something over which the user/ management disagree with the Analyst, resulting in dissatisfaction and frustration. An Analyst who does not have a thorough understanding of the environment, working area and its terminology cannot communicate effectively with the User/Management and convince them if required. Brown G.L.

the strategy adopted to solve a particular problem, depends on the size and complexity of the problem. In a similar way the strategy to develop a software system will depend on its size, complextiy and environment. This is because a software system for a complex problem cannot be developed in a similar way to one for simple problems.

In this phase the output from the Initial Feasibility Study subphase will be used to develop a strategic plan discussing how the Company will proceed with the project. Whether some aspects will be pursued further than others, who will undertake the task and accept the consequences and responsibility of:

1. the changes in personnel, hardware, etc.,
2. the effects of implementation, transition and possible delay,
3. the arrangements for formal training and development of new manuals, etc..

This strategic plan should be developed in the light of managements long term policies.

3 * Analysis Phase

Subphases

* Analysis of Environment
* Requirement Analysis
* Prepare Requirement Specifications
* Check for Consistency
* Search for Existing Available System
* Feasibility Analysis & Detailed Proposal

Analysis means to decompose something (which is under study) into its component parts for identification, examination and interpretation. Systems Analysis means to break an old system

in an infinite number of ways. This number eventually is reduced by deciding to adopt a single or a set of particular methods or Methodologies. There are several Design Methods or Methodologies suitable for different fields and environments, such as Structured Design [20], SADT [11], JSD [21], MASCOT [22], ACM/PCM [23], ISAC [25], LSDM-LBMS [16] and STRADIS [24], etc., each one having many drawbacks. These Design Methods or Methodologies, can be classified according to four distinct main approaches, i.e.

1. Building a system by giving emphasis to functions and processes This is called functionally or Process based Design such as ISAC, NIAM, SASD, etc..
2. Designing a System by giving emphasis to the flow of data to be used, i.e., describing the System in terms of data. This is called a data driven System, such as SYSDOC, D2S2, LSDM-LBMS, etc..
3. Designing a System by developing a model of the real world. This is done without initially mentioning the functions to be performed by the System, but introducing these functions explicitly at a later stage, by using verbs in the Specification. The JSD method is of this type.
4. Designing a system based on the States of entities and processes or their behaviour and changes in any combination within the environment. Designs based on SDL [47] [48] are of this type.

Criteria for the choice of a particular method, or Methodology, as in the Analysis Phase, should be based in the Feasibility Analysis and the Detailed proposal. To obtain a good Design the following are some general guidelines and recommendations; but of course as mentioned above, the final selection must be based

on considerations of the type of problem under study.

1. Similar to the Analysis Phase develop a dictionary of all the different terms, concepts, notations, and definitions, which are to be the in the System in order to avoid misinterpration of words and communication problems.
2. Decompose the overall problem and its Specifications successively into smaller and smaller subproblems, possibly called modules or subsystems. This decomposition continues till a point is reached from which the Design looks relatively easy. The criteria for this decomposition depends on the type of problem, environment and experience of the Designer.
3. The modules or subsystems must be able to be conveniently and coherently connected.
4. *Each major module or subsystem* must be further investigated with a view to finding out its main functions, processes, entities, actions, and their relationships, etc..
5. Each module or subsystem must be tested to make sure it is performing its functions and processes up to a satisfactory level. This needs the development of suitable test data having acceptance criteria and possibly mentioning the time dependence between modules or subsystems.
6. Each module or subsystem should be protected against the possibility of incorrect input data. This needs the development of certain constraints to prevent incorrect input.
7. Modules or subsystems must be easy to integrate, manage and perceive, i.e., which can easily be implemented, maintained and amended.

As the design is decomposed it may be necessary to use the Life

Cycle recursively. This is because further requirements relevent to a subsystem may need to be determined or clarified. In this case many of the subphases need not be performed.

The detailed subphases for Design are as follows.

5.1 Planning and Decision

"Design involves both making decisions about what precisely a system will do and then planning an overall structure for the software which enables it to perform its task" [26]. The Planning and Decision activity involves using the System Specifications to develop the System. In this subphase crucial decisions about the basic structure of the Software Design and how to convert them into a series of hierarchical steps are made. At this stage, great care is needed as these decisions have great impact on the final Software Design and can have consequential effects for the later Maintenance Phase. This activity needs consideration and it is probably of an iterative nature.

The Designer must establish, in consultation with the Analyst and the User/Management, clear definitions, main functions and general processes of the system, operating parameters, nature of data to be stored, and patterns of output. The Designer must prepare a list of instructions and formulae, if there are any, depicting how to solve the problem. Clear definitions of functions and the general processes of the system will help the project team in understanding and deciding which functions and process are to be taken and which are to be by-passed. It will further help in making decisions about which tool or methodology should be implemented.

5.2 Conceptual Design

The Conceptual Design or visualized model means an appropriate image of the actual Design, its information flows, processes

network, entities, functions, modules, subsystems, their relationships, and interactions in the Designer's mind.

In this subphase the Designer will make certain basic assumptions about the Design to be built based on the decisions and planning made in the previous subphase, the environment, nature of the selected proposal and System Specifications thereof, developed in phase_4, and of course his/her experience. The Designer will produce ideas for creating modules, subsystems, databases, etc., for example:

1. The Designer can assume that it will be better to develop a conceptual model of the real world, based on the results of Analysis and System Specifications phases; and later introduce functions. Thus the Designer uses the JSD method with the essential required knowledge about the Systems Environment, Analysis results, and System Specifications, i.e., some extra items which are missing in the JSD method.
2. The Designer can build a conceptual Design or model about entities types, relationship types and data element types using the Analysis results and System Specifications e.g., using the SYSDOC Methodology.
3. For a Real time System the Designer can make a conceptual Design based on the Analysis results and the System Specifications showing that changes in the computer system will occur in the same sequence as changes in the environment, e.g., SDL method.
4. A Conceptual Design of a System consisting of object and activity classes and rules can be developed by using the information obtained from the Analysis results and the System Specifications. The description of this Conceptual System is called an abstraction system by NIAM, and it can be expressed by using

a Conceptual grammer.

5. If the Structured Design approach is followed, then a Conceptual Design can be built by developing concepts about data, data structure, data flow diagrams, a data dictionary, hierarchies of modules, etc. based on the Analysis results and the System Specification.

The output of Conceptual Design should be documented properly after consulting with the Analyst and the User/Management for further use in developing Physical Design.

5.3 Physical Design

By Physical Design we mean designing details of data, data format, data structure, data base, data dictionary, functions, processes, modules, subsystems, etc., according to the selected Design Method or Methodology which can be represented and implemented physically, (like actual things) at a particular time in a system.

In Physical Design each module or subsystem will be physically or actually identified along with its interfaces with other modules or subsystems.

To develop a Physical Design in this Life Cycle the output of Conceptual Design will be used. In this subphase modules or subsystems, data dictionaries, data bases, etc. conceived in earlier subphases will be developed physically in detail.

The Physical structure of data, data dictionaries, modules or subsystems etc., for example, can be described by charts, diagrams, tables, and coding formats. This involves the development of

1. dataflow diagrams,
2. structure charts, decision tables, etc.,

3. modules or subsystems as a part of a hierarchy and showing their points of entry and exit,
4. input details, and the layout of, for example, the data format, the data structures, the input variables, and their points of entrance into the module or subsystem etc.,
5. output details and their layout e.g. output, output structure of files or databases, their point of exit from modules or subsystems and conditions thereof, format of error messages, and physical source of output etc.,
6. Details about file structures, and the procedures to retrieve and update, indicating which files are to be permanently kept.

The Physical structure of processes, procedures, activities, and functions can be described by using structured English, pseudocode, machine processable language, diagrams, charts etc.

The Physical Design of user interface can be developed from user options, clerical routines, available or proposed Hardware. This interface will assist the User/Management in the selection of the report option.

The output of the Physical Design subphase will be a Design in which each software module or subsystem will be physically available along with all its interfaces to other modules or subsystems, and will be matched.

It is important to mention that the whole process of developing the physical Design like other subphases is of an iterative nature, and needs frequent consultation with the Analyst and the User/Management. The output of this subphase will be available for the Program Development subphase.

The details about modules or subsystems, their control, dataflow, output layout, different constraints etc.,

elaborated in the different Design subphases, will be integrated to form a frame or skeleton of a Software System to be used in the Implementation Phase.

On the basis of this frame or skeleton the Analyst/Designer will be able to estimate the program development, installation, and implementation cost of the System. This cost estimate will be submitted, along with the Design frame, to the User/Management for their approval prior to any further move.

0 * Verification & Validation Phase.

Verification is defined variously as:.

1. "The process of determining whether or not the product of a given phase of the software development cycle fulfills the requirements established during the previous phase."
2. "Formal proof of program correctness."
3. "The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services or documents conform to the specified requirements" [52].

Verification implies to the establishment of the correctness of some thing by comparing it against some standard. For example to compare the developed system with its Requirements, System Specifications, etc.

Validation means:

> "the process of evaluating Software at the end of the software development process to insure compliance with Software requirements" [52].

Thus validation ensures that the output of a Phase complies with a standard such as the Requirements, the Detailed

Proposals, the System Specification, etc..

The purpose of a V & V Phase is to:

1. guide the Analyst/Designer and the User/Management in making decisions about their next move, which may be to go ahead or abandon the project,
2. satisfy that the System Specifications are developed according to the result of the Analysis Phase and serve its purpose,
3. satisfy oneself and the User/Management that the System Design is developed according to the Requirements and the System Specifications,
4. check out the output of the Design Phase against the basic rules of the selected Design Method or Methodology, such as rules about input/output data structure, criteria to build modules or subsystems, their connections and calling or exit methods etc, and
5. ensure that its result satisfies the test data.

The V & V Phase may be activated at any time and in any Phase of the Life Cycle if needed. Specifically it is activated after:

1. Requirements,
2. Detailed Proposals,
3. System Specifications,
4. Design output,
5. Program Structure,
6. Coding, etc.

For example.

1. The V & V Phase should be called during the Analysis Phase to ensure that the Requirements and the Detailed proposals have been developed properly, because any error at this Phase can create major effects in terms of complexity,

time, and cost.

2. The V & V Phase should be used during the Design Phase to ensure that the modules or subsystems are complete, cohesive and minimaly coupled. That the combined effect of the modules or subsystem, data flow, control flow etc. is according to the Requirements & System Specifications. Furthermore it must be documented that the Software is feasible to implement and can serve the purpose and the environment for which it was designed.
3. The V & V Phase must also be applied during implementation to ensure that the Program is an appropriate representation of the Design output and no unapproved changes were introduced by the Programmer during the Phase. It will be better to use another V & V Phase after integration of modules or subsystems to check their combined effect.
4. The V & V Phase can be used for detecting unperformed Requirements and mistakes, which later can be traced back to find the source of error.

The V & V Phase can be achieved by,

1. Testing or Auditing,
2. Inspecting & Reviewing,
3. Implementing & Running the Phase.

Activities such as testing the output of a phase, must be based on testing against certain pre-selected factors and constraints to ensure that the Requirements and the Specifications have been implemented properly.

The goals of the V & V Phase can be achieved by inspecting and reviewing the selected factors which according to the User/Management may effect the performance of the System.

Many of the V & V activities for any Phase can also be achieved by running and implementing the System or by using a Prototype variant of the Life Cycle.

Great care must be taken in preparing the list of the important factors and constraints which might be used for the above purposes. This list must depict the sequence and level of extent up to which the V & V Phase will be done. The output of the above activities must be documented for possible future use.

If the User/Management or anyone concerned is not happy with the output of the Validation Phase , it means that perhaps:

1. User/Management have changed their mind about what they 'want'.
2. The Analysis was not done properly.
3. The System Specifications were not developed carefully.
4. The selected detailed proposal, the basis of System Specifications does not reflect the Requirements of User/Management or is not suitable for that particular problem and the environment.
5. The Design activity was not done properly.
6. The selection of the Design Method or Methodology is not appropriate and cannot solve the problem properly.

Hence due to any of the above reasons it will become necessary to go back to the appropriate phase and do the whole excercise again using the recursive property of the cycle.

This re-useability of a V & V Phase in the Life Cycle can certainly increase the quality of the output for the particular Phase in which it is being used and as a direct result the overall quality of the System.

Comments

This life Cycle has been constructed by piecing together the various Life Cycles and Life Cycle fragments occuring in the literature. In order to do this, words and names have been changed or redefined for the sake of consistency. It is hoped that in doing this no useful information has been lost. The rationale and motivation for some of the steps are not always obvious to the other but these steps have been included provided only that the original sources include some explanations. It is assumed that these explanations are meaningful to these who actually work in those particular application areas.

Missing from the Life Cycle is a Prototyping Phase. The reason for this is that prototyping appears to be a name for a particular (simplified) form of the Life Cycle itself. All prototypes must be built or evaluated etc.. These are particular phases of a more general Life Cycle. It seems unlikely that this is the only special case of a Life Cycle but the authors have not detected any other well established instances as yet. One possibility is the concept of establishing a general 'framework' by means of a rapid and specific Life Cycle and then adding details or additional facilities by another pass over a more general Life Cycle.

It is clear from the literature that iterative or recursive application of Life Cycles are the norm and indeed there is little discernable evidence that Software should be implemented by a single pass. It is understood that experiments are proceeding and more information from controlled experiments should be available in the future.

Glossary

System

1. A System represents a set of interconnected items or elements arranged in a certain order, for example, management of an organisation, a computer system, a railway network, etc..
2. In the domain of computer science a system may be defined as any combination of hardware and software, which fulfils a specific purpose.

System User

A system user is:

1. a person who will be using the system such as an operator, software programmer, quality controller, stock control officer, accountant, etc.,
2. a person who will be using the system in a managerial capacity such as a bank manager, quality control manager, etc.,
3. a person or organisation who will be using the system as the owner, for example, a government ministry, a banking organisation, etc.,
4. a person who will be using the system without knowing anything about hardware or software, but are supposed to know clearly about the results they are expecting and be able to check the input or output, e.g., bank customer, user of flight information services, etc.,
5. the clerical and other people who will work directly with the system by using terminals, filing output forms or interpreting output for their jobs etc.,
6. a person or organisation whose requirements are to be evaluated and understood and the result of

the study is then submitted to them for further actions, e.g., bank manager, government officials, etc.,

7. anyone who uses the system by issuing commands.
8. but it excludes Analysts and Designers and anyone who supplies the system.

Management

1. The management of an organisation for which the system is to be developed.
2. The management of the system development team.
3. The management whose requirements are to be evaluated, understood and the results of the study to be submitted to them.
4. The management of a data processing section or computer section within an organisation.

Systems Analyst

A person who talks and works with the user and their management in order to discover their needs and wants, and identifies the working environment.

Designer

A Designer is a person who:

1. uses proposals developed by the Analyst to develop system specifications,
2. specifies how to convert system specifications into a design to meet the user's requirements,
3. is concerned with designing details of a system, subsystems, modules, etc..

Programmer

A programmer is a person who implements the design decisions in a programming language.

N.B. words like Analyst, Designer, User, Programmer and Management may be used as either singular or plural and male or

female. Combinations such as user/management should be interpreted as a member of either or both groups.

Methodology

Methodology involves philosophies, management techniques, methods, phases, rules, techniques, tools, aids and documentation required to develop a software system. A methodology may contain more than one method to do different software development activities together with the rules to apply them.

Method

A method is a way or steps which should be followed during software developing activities, such as project Management, analysis, design and testing, etc.. It does not usually cover all the activities of software development. It may in fact be considered as a subset of a methodology. It is available for small projects only.

Phase

A phase is a state or stage of development of a software activity over a discrete period of time. Conceptually they do not overlap and keep time ordering, although in practice this is not usually essential. Each phase has both input and output.

References

1. Bell T. and Thayer T., "Software Requirements: are they really a problem", Proceedings of 2nd. International Conference On Software Engineering, San Francisco, 1976.

2. Fischer G. and Scheider M., "Knowledge Based Communication Processes In Software Engineering", Proceedings of 7th. International Conference On Software Engineering, Orlando, Florida, USA., March 1984, pp. 359.

3. Freeman P., "Why Johnny Can't Analyze", Systems Analysis and Design, A Foundation For The 1980's, pp. 323, North Holland Inc.,1981.

4. Longmans New Universal Dictionary,

5. De Marco T., Structured Analysis and System Specification, pp.4, Prentice_Hall Inc., Englewood Cliffs, N.J. USA.

6. Brown G. L., "Logical Design Of Computer Based Information Systems", Systems Analysis and Design, A Foundation For The 1980's, pp. 182_83, North Holland Inc., Englewood Cliffs, N.J. USA, 1981.

7. Wasserman A.I.,"The User of Software Engineering Methodology": An Overview, Information Systems Design Methodologies : A Computer Review, Proceedings Of IFIP WG.8.1, pp. 591_628. North Holland Publishing Company.

8. Mumford E. & Henshall D., A Participative Approach To Computer System Design, Associated Business Press London, ISBN 085227 2219, 1979.

9. Mumford E. & Weir M., Computer Systems in Work Design_The ETHICS Method, Associated Business Press London, ISBN 085227 2308, 1979.

10. Sarvari I. L., "Modules & Languages For Software Specification and Design", pp. 119, Workshop Notes: International Workshop on Models & Languages For Software Specification & Design, March 1984, Orlando, Florida, USA, ISSN 0225-0667.

11. An Introduction To SADT, SofTech, Inc., Waltham, Ma., Document 9022_78, Feb. 1976.

12. Gane C. & Sarson T., Structured Systems Analysis, Englewood Cliffs, N.J. Prentice_Hall 1979.

13. Teichroew D. & Hershey E. A., "PSL/PSA: A Computer Aided Technique For Structured Documentation And Analysis Of Information Processing System", pp 41_48, IEEE Transaction Of Software Engineering Jan. 1977.

14. Rheijen V. E. & Bekkum V. J., "NIAM, Nijssen Information Analysis Method", Information System Design Methodologies: A Comparative Review, Proceedings Of IFIP WG. 8.1, pp 537_89, 1982, North Holland Publishing Company.

15. Alford M. W., "Software Requirement Engineering Methodology", Final Report_Volume I; Ballistic Missile Defence Advanced Technology Center, Huntsville, AL, August 1979.

16. Introduction To LSDM, Learmonth & Burchett Management Systems, 22 Newman Street , London W1P 3HB, Document No. 9/83.

17. Teichroew D., Macasovie P., Hershey E. A. & Yamamoto Y., "Application Of The Entity_Relationship Approach To Information Processing System Modelling", Entity_Relationship Approach To Systems Analysis And Design , P.P. Chen (ed), pp. 15_38, North Holland Publishing Company, 1980.

18. Yourdon E., Structured Design Workshop, pp. 1-2, Edition 2.1, 1980, Yourdon Inc., 1133 Avenue Of Americas, New York, N.Y.10036.

19. Chapin N., "Graphic Tools In The Design of Information Systems", Systems Analysis And Design, A Foundation For The 1980's, pp 121_162., North Holland Inc., 1981.

20. Yourdon E. & Constantine L.L., Structured Design, Yourdon Press, 1133 Avenue Of Americas, New York, N.Y. 10036.

21. Jackson M.A., Michael Jackson System Development, Prentice_Hall, International.

22. Jackson K. & Simpson H.R., "MASCOT", RRE., Technical Note No. 778, RRE. Procurement Executive, Ministry Of Defence, Malvern Worcs.

23. Brodie M.L. & Silva E., "Active And Passive Component Modelling ACM/PCM", Information Systems Design Methodologies: A Comparative Review, Proceedings of IFIP WG. 8.1, pp 41_91, North Holland Publishing Company, 1982

24. McAuto, Stradis: System Development Methodology Product Description (undated)

25. Lundeberg M., "The ISAC Approach To Specification Of Information Systems And Its Application To The Organization Of An IFIP Conference", Information Systems Design Methodologies: A Comparative Review, Proceedings Of IFIP WG. 8.1, pp. 173_234, North Holland Publishing Company, 1982.

26. Liskov B. H., " A Design Methodology For Reliable Systems", Tutorial On Software Design Techniques, 3rd. Edition, Edited By Peter Freeman, A.I. Washerman IEEE,

1980.

27. Frode A., "IFIP WG. 8.1 Case Solved Using Sysdoc and Systemator", Information System Design Methodologies: A Comparative Review, Proceeding of IFIP WG. 8.1, pp. 15_40, North Holland Publishing Company, 1982.

28. Macdonald I. G. & Palmer I. R., "System Development In a Shared Data Environment", The D2S2 Methodology, Information System Design Methodology: A Comparative Review, Proceeding of IFIP WG. 8.1, pp. 235_283, North Holland Publishing Company, 1982.

29. Meyer B., "A System Description Method", p. 42, Workshop Notes: International Workshop on Models & Language for Software Specification & Design, March, 1984, Orlando, USA., ISBN 0225_0667.

30. Mascot Supplies Association, "Official Handbook of MASCOT", RSRE, Malvern, Worcs. U.K.

31. Howden W. E. and Eichorst P., (1978), "Proving Properties of Programs from Program Traces", in Tutorial Software Testing and Validation Techniques. Eds. E. Miller and W. E. Howden. IEEE Computer Society, pp. 46-56.

32. Howden W. E., "Functional Testing and Design Abstractions". J. of System and Software. Vol.1, No.4, 1980, pp. 307-314

33. Parnas D. L., "On the Criteria to be used in Decomposing Systems with Modules". CACM, Vol. 15, pp. 1053-1058, Dec.1972

34. Goodenough J. and Gerhadt S.L., "Towards a Theory of Test Data Selection",IEEE Trans. on Software Engineering, SE-1, pp. 156-173, 1975.

35. Woodward M. R., Hedley D. and Hennell M. A., "Experience with Path Analysis and the Testing of Programs". IEEE Trans. Software Eng., Vol.6, No.3, pp. 278-286, May 1980.

36. Stucki L. G., "Automatic Generation of Self-Metric Software". Proc. IEEE Symp. on Computer Software Reliabilty, N.Y., APRIL 1973.

37. Hennell M. A., Woodward M. R. and Hedley D., "The Testing of a Software Tool". Proc. Int. Symp. on Applications and Software Engineering, Montreal, September 1979, ACTA Press, ed. M. H. Hamza, pp. 16-20, 1980.

38. Holthouse M. A. and Hatch M. J., IEEE Computer, Vol.12, No.8, pp. 33-36, August, 1979.

39. Al-Jarrah M.M.F., "The Study and Application of Program Analysis in a Cobol Environment". Ph. D. Thesis, Brunel University, Middlx, England, 1982.

40. Kishida K., "Techniques of C_1 Coverage Analysis". Testing Techniques Newsletter, Vol.3, No.3, 1980 p. 4.

41. Boehm B., "An Experiment in Small-Scale Application of Software Engineering",IEEE Trans. on Software Engineering, SE-7,482-493,1981.

42. Hennell M. A. and Yates D., "An Examination of Standards and Practices for Software Production". Computers and Standards. Vol. 1, pp. 119-132, 1982.

43. Hennell M. A. and Delves L. M., "Conference on the Performance and Assessment of Numerical Software". Academic Press. 1980.

44. Foster J. M. and Foster P. D., "Abstract Data and Functors". RSRE Malvern, Worcestershire, U.K.

45. James C. "Software Development: A Rigorous Approach". Prentice Hall. 1980.

46. Hamilton M. and Zeldon S., "Higher Order Software - A Methodology for Defining Software". IEEE Transactions on Software Engineering Vol. SE-2, No. 1. pp 9-32. March 1976.

47. Series Z, Recommendations (Z.101 to Z.104) Specification and Description Language (SDL) Vol. VI-1. CCITT.

48. Cerchio L., "A System Design Methodology Based on SDL", SDL News Letter No. 4 November 1982, pp. 19-20.

49. Allen E. and Engholm C. R., "The Need for Clear Structure in "Plain Language" Legal Drafting". Journal of Law Reform Vol. 13, No. 3, 1980

50. Hennell M. A., Woodward M. R. and Hedley D., "The Testing of a Software Tool". Proceedings of International Symposium on Applications and Software Engineering, Montreal, September 1979. ACTA Press Ed. M. H. Hamza. pp. 16-20.

51. Hennell M. A., Woodward M. R. and Hedley D., "Experience with an Algol 68 Numerical Algorithms Testbed". Proceedings of the Symposium on Computer Software Engineering, New York. April 1976. Polytechnic Institute of New York Microwave Research Institute Symposia Series, Vol. XXIV, ed. J. Fox. pp 457-463, 1976.

52. IEEE. Standard Glossary of Software Engineering Terminology, IEEE Std. 729-1982

53. Johnson S. C., "YACC: Yet Another Compiler-Compiler". Bell Laboratories. 1978.

54. Gerhart S., Musser D., Thompson D. et al, "Overview of the AFFIRM Specification and Verification System", IFIP 80.

55. Enger N. L., "Classical and Structured Systems Life Cycle Phases and Documentation", Systems Analysis and Design, A Foundation For The 1980's, pp. 1_24.

56. Biggs C. L., Birks E. G., Atkins W., "Managing The System Development Process", Prentice-Hall Inc., Englewood Cliffs N.J. 07632, ISBN No.0-13-550830-4.

57. Blum B., "Models and Language For a Specific Application Class", Workshop Notes: pp.68-9, International Workshop on Modles and Languages For Software Specification and Design, 1984, Florida, USA.

58. Ramamoorthy C. V. & Ma Y. N., "Design and Analysis of Computer Communication Systems", Systems Analysis and Design, A Foundation For The 1980's, pp. 452-460.

59. O'Neill D., "Software Engineering Techniques Applied To The Systems Development Process", Systems Analysis and Design, A Foundation For The 1980's, pp. 330-340.

60. Freeman P., "The Context of Design", pp. 2-4, Tutorial on Software Design Techniques, 3rd. Edition, 1980, IEEE Computer Society.

61. Tutorial, JSP & JSD: The Jackson Approach to Software Development, IEEE Computer Society, ISBN No. 0-8186-8516-6.

62. Mills H., "Top-down Programming In Large Systems", (Courant Institute).

63. Hamilton M, and Zeldon, "Top-down, Bottom-up Structured Programming and Program Structuring", Charles Clark Darper Laboratory, M.I.T., December 1972.

64. Corner M. F.,"Structured Analysis and Design Technique", p.213-234, Systems Analysis and Design, A Foundation for the 1980's.


DATE
FILMED
0-8