A comparative case study of Programming Language Expansion Ratios

A thesis presented in partial fulfilment of
the requirements for the
degree of Master of Technology in Computing Technology
at
Massey University

Lim, Ping Hwee
February 1989

Abstract

An effective size estimation tool must allow an estimate to be obtained early enough to be useful. Some difficulties have been observed in using the traditional lines of code (LOC) measure in software sizing, much of which is due to the need for more detailed design information to be available before an accurate estimate can be achieved. This does not allow the result to be obtained early in the software development process. Moreover, the inherent language-dependency of LOC tends to restrict its use. An alternative measure using Function Point Analysis, developed by Albrecht, has been found to be an effective tool for sizing purposes and allows early sizing. However, the function point measure does not have a sufficient historical base of information for it to be used successfully in all cases with existing models of the software development process. Because lines of code already have a sense of "universality" as the *de facto* basic measure of software size, it can serve as a useful extension to function points. Language Expansion Ratios are seen as the key in providing such an extension by bridging the gap between function point and lines of code. Several sizing models have made use of expansion ratios in an effort to provide an equivalent size in lines of code in anticipation of its use in productivity studies and related cost models. However, its use has been associated with ranges of variability. The purpose of this thesis is to study ***Language Expansion Ratios***, and the factors affecting them, for several languages based on a standard case study.

This thesis surveys the prevailing issues of software size measurement and describes the role and importance of language expansion ratios. It presents the standard case study used and the methodology for the empirical study. The experimental results of measurements of the actual system are analysed and these form the basis for appropriate conclusions on the validity and applicability of the expansion ratios studied.

This research shows that the use of Language Expansion Ratios is valid but it is considered inadequate when applied in its present form. This was found to be due to the weighting factors associated with the appropriate function value obtained for the different functional categories of the system.

Acknowledgments

I wish to extend my gratitude to Professor Graham Tate for consenting to be my supervisor for this thesis. His direction has been very important to this work. I am also grateful for his careful review and comments of the entire content of this thesis. His guidance and encouragement are very much appreciated.

A special thanks to Miss June Verner for her constant help and support during the course of this research. In particular, I wish to express my deep appreciation for her help in making the documentation facilities available to me.

Finally, I would like to express my gratitude to Miss Lili Ton for her help during the preparation of this thesis. Her efforts and patience in some of the typescripts and drawing of the many diagrams deserve my heart-felt thanks.

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

The impact of the Software Crisis emanates from the realization of the importance which computerization has on society and industries. An overwhelming demand for software escalated to such an extent that traditional software development techniques could not cope. The problem related to the difficulty of establishing techniques for handling the growing size and level of complexity of software systems whose development schedules could not be accurately predicted.

The cost of software grows disproportionately with the other associated costs of computer systems and this upward trend is an issue of concern [DACS87,JONE86]. In the search for a solution to the software crisis, the field of Software Engineering was created in a deliberate attempt to use a combination of techniques, methods, and tools for producing economical software that is reliable and works efficiently.

An important aspect of Software Engineering has been the focus on software size estimation as a prerequisite for resource planning and scheduling in the software development process. The unpredictable nature of the software development process has prompted intense research in finding a way of making programs measurable and, hence, more predictable. Source lines of code (LOC) have been widely used for research studies involving size estimation. The count of the number of lines of code is said to be related to the size of the effort required in the development process. Several studies have been pursued to verify this.

The study by Walston and Felix [WALS77] discusses research into a method of estimating programming productivity by measuring the rate of production of lines of code by project and relating them to factors which might influence its behaviour. Measurement data were collected from 60 projects in one organization and were maintained in a measurement database. Based on these data, productivity analysis of effort and product size shows a nearly first-power (or linear) relationship. In another study carried out in the Software Engineering Laboratory at the University of Maryland, Basili et al [BASI81] examined the relationships among the various basic software development variables, such as size, effort, project duration etc. The analysis reveals a high probability of a relationship between total effort and delivered lines of source code. The relationship is nearly linear with a coefficient of determination of 0.93 at the 0.001 level of significance. This high correlation indicates a possibility of using source lines of code to predict the total effort required in a development project. This result was found to be consistent with the study by Walston and Felix [WALS77].

The validity of using source lines of code as a predictor of programming effort is also based on the assumption that it includes a measure of functional complexity. Although studies by [WALS77] and [BASI81] indicate a linear relationship between source lines of code and effort, Basili et al [BASI81] also stated that the assumption which relates the functional complexity to program size is subjective. The argument is that size may increase at an even greater rate as complexity increases. This follows the notion suggested by Brooks [BROO75] that man and month are not necessarily interchangeable as the result of increasing one may not directly cause the other to decrease. However, it can be acknowledged that the link between lines of code and effort prediction is a valid one, though more factors relating to complexity will have to be considered. Already, several cost models have used lines of code as the input parameter for cost and schedule estimates. Of the better known costs models are the COCOMO and SLIM models [BOEH84].

The biggest difficulty in using today's Software Cost models has been the problem of providing sound sizing estimates. Several methods of size measurement have been developed to depart from the usual lines of code measure for size estimation in an attempt to search for a better way to estimate software size early enough for it to remain useful. One such method that remains popular is the measure for function value developed by Albrecht called Function Point Analysis. Function Point Analysis has been quite successful and is adopted in several commercial software sizing models. However, most of these models also provide an equivalent size in lines of code in anticipation of its use in productivity studies as well as with existing cost models. The use of an expansion ratio to convert from function points to lines of code has been recognized to contain ranges of variability due to the fact that it is obtained from different sources or based on different datasets. However, the expansion ratio remains useful in standardizing the basic unit of measure for quantifying software. The purpose of this thesis is to study ***Language Expansion Ratios***, and the factors affecting them, for several languages based on a standard case study.

Chapter 2 provides a summary of the background information relating to the issues of software sizing. A review of software metrics and the problems associated with them are presented followed by a discussion of the subjectivity of sizing in existing size estimation models. It emphasizes the need for an improved size estimation method which can be applied to the wider range of programming languages and examines the role of language expansion ratios in providing such an extension to existing sizing methods.

Chapter 3 presents the research objective for this thesis and a brief outline of the system analysis and design specification of the case study used. The languages used in this study are presented with specific emphasis on their main development features.

Chapter 4 provides an outline of the methodology adopted for this empirical study. It describes the system development tasks involved in the implementation phase as well as the software development strategy used. The implementation phase includes detailed design and code implementation. Other implementation-related activities such as size and cost/effort estimation, data collection, software measurement and statistical analysis methods are also included as an overall base appropriate for the empirical study.

Chapter 5 presents the experimental results of the case study. It describes the detailed analysis of experimental data and discusses related observations associated with the results of the analysis.

Chapter 6 provides a summary and appropriate conclusions based on the empirical study conducted in chapter 5.

CHAPTER 2

SURVEY OF BACKGROUND INFORMATION

To date, Software Sizing Methods still have a certain degree of subjectivity in their estimation of software size. Independent attempts in measuring and estimating software tend to concentrate more on certain aspects of the program characteristics and less on others. Efforts to incorporate some combination of the fundamental concepts of software metrics have been progressive although the subjectivity factor remains a persistent problem. The growing number of programming languages will no doubt influence the prevailing concerns for sizing. The link between sizing and effort estimation has been indicated as a continuum in the software development process [BOEH84,VERN87]. Notwithstanding the prevailing problems with estimating the size of software, efforts in cost/effort prediction models have shown significant progress although there is still an explicit dependence on size measures [BOEH84, PRES87, VERN87]. A more accurate size estimate is deemed a necessity if more accurate cost and effort prediction is to be possible. Hence, the continuing need for research in sizing becomes apparent.

This Chapter reviews the subjectivity of software size estimation by looking at the attempts at solving the software sizing and estimation problems and the estimation techniques adopted by various software sizing models. A review of software metrics is presented highlighting their use and their deficiencies . Metrics applied to existing software sizing models showed variability in the predictions. Their significance is discussed. The need for a sizing mechanism which extends across language boundaries becomes increasingly important amidst the growing number of available languages. The relevant potential for such a mechanism is investigated and the role and importance of *Language Expansion Ratios* will then be presented.

2.1 Review of Software Metrics

The effort expended in software development is invariably influenced by some dominant factors. Categorically, they involved the type of application, the language used and the development environment. Several attempts have been made to identify characteristics of programs deemed to be influential factors affecting program behaviour. Some relevant factors identified are the structure, function and size of programs. Investigative studies have explored these factors, and others, in certain application domains in an attempt to clarify ideas for improving programming productivity and to effectively measure relevant data so that the estimation process of software projects can be better controlled.

In a comprehensive review of research into measurement alternatives, Capers Jones identified five major categories which attempts to study program behaviour but depart from the usual unit of measure of Source Lines Of Code (SLOC) [JONE86]. The use of SLOC is considered to be subjective and language dependent. Among the different alternatives identified, three specific categories of research have captured much attention among researchers as their basic concepts are quite unique. These relate to the study of:

a. functional and data aspects of code
b. program structural complexity
c. technology-independent function count.

Each of these categories will be discussed in detail in the following sections. The explanatory power of each forms a unique theoretical base; however, there are some distinct deficiencies which still give some cause for concern.

In determining appropriate software metrics, Conte et al suggest that for a metric to be used effectively it must be objective and consistent under any prevailing circumstances [CONT86]. The objective measurement should be one which can be computed precisely with an algorithm and should be consistent irrespective of changes in time, place or observer. This is important for productivity studies as well as the convenience of providing an automated tool for it.

The continuing refinement of languages have reached a point where several problems are, consequently, introduced. These affects the process of software measurements; where subjectivity has seized-up on the objective goals. This will be discussed in greater detail in the following sections.

This section highlights some of the more common software metrics used by software researchers and its application have been extended for use in software size estimation in some commercial software sizing models [DACS87].

2.1.1 Size Metrics

Size metrics are direct measures of software and the process by which it is developed [PRES87]. It is considered a natural quantification of software and serves as an indicator to the effort required to produce the software. The size of programs is important because it provides the following advantages [CONT86]:

a. the computation of size from source code can be done quite easily
b. it serves as an important link to other models of the software process
c. it is useful for productivity studies.

Measuring software size dates back to the early days when computers were first introduced. At that time, the use of lines of code to determine the size of programs was

found to be reasonable where machine language, or the later assembler language, was concerned. Its use was propounded as being a useful size metric as it was easily determined from the source code listings. The changing form and style of programming languages begin to have a profound impact on size metrics as the number of variations made it very difficult to maintain objectivity. Size metrics, subsequently, have had to shift their fundamental emphasis. The use of token and function point counts are part of that fundamental change.

The difficulty in analysing program size is that "almost never does size change by itself" [JONE86 pp. 95]. Several other related factors tend to become more complicated as size increases. As indicated by Brooks in "The Mythical Man-Month" [BROO75], when the size of software is large, it is less likely that putting more men on the job would have any effect on shortening the schedule. In fact, the opposite result might prevail.

However, size metrics provide a quantifiable measure for software and have become a dominant factor in effort-related studies. Some concern about inconsistencies has led to the development of new techniques of measuring software. However, this difficult area has not been satisfactorily resolved. The following discusses the three main counting techniques used by software researchers that have been applied in various cost models and productivity studies [BOEH84] .

2.1.1.1 Line Count

Line count is considered the most natural way to quantify a piece of software. The counting of Lines Of Code (LOC) represents the count of the basic unit of measure associated with programming languages. In the early pioneering years of computer systems, the use of machine languages or assembler languages were more rigid and did not vary very much in terms of form or style. In using those languages, every action

had to be coded and every line of code relates to performing a particular task. Perhaps, it was a useful measure then as LOC had quite literally a 1:1 relationship with the effort required to write a line of code. However, the inefficiencies in coding with early programming languages soon caused a transformation of the basic code representation and, together with it, a change of form and style of programming.

Although LOC is a simple metric to use, there is "no general agreement as to what constitutes a line of code" [CONT86]. As higher level languages have free-form representation and a variety of other types of statements besides executable statements, it becomes unclear as to what should be included in the line count. Most researchers agree that lines of code which directly affect the function of the program need to be included [CONT86]. However, it is considered that it takes less effort to put in comments or blank lines into a program than it does to write an executable statement. Furthermore, free-form representation allows more than one executable statement to appear on a physical line. The problem now is whether to count physical lines or executable statements.

There are several definitions in counting LOC by various researchers [BOEH84, JONE86]. One such definition considered to be used by most researchers is as follows [CONT86] :

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

Capers Jones listed eleven major variations in line counting rules both at the program and project level [JONE86, pp. 15]. At the program level, the range of variation could be as much as 5 to 1 when comparing the most diffuse counting technique and the most

compact. At the project level, productivity studies can become confused when there is no distinction between the actual developed code and the code that already exists and is being used for convenience. Re-used code, temporary or scaffold code, tool and support programs are code which does not require the same effort as the actual developed code. Their inclusion in the line count becomes somewhat debatable. Such variation in counting rules does not allow effective comparison of results among researchers as such comparative studies would inevitably be clouded with uncertainties. LOC initially achieved objectivity but its use has slowly deteriorated in the direction of subjectivity. Unless standard counting rules are established, this situation is unlikely to change.

Besides the inconsistencies related to line counting, there are paradoxes associated with LOC in productivity studies [JONE86]. Capers Jones highlighted several of these paradoxes which tend to penalize high-level languages as well as higher quality programs. For example, in using LOC as a productivity indicator, high level languages are penalized for producing less code. As the rate of decrease in cost is not directly proportional to the rate of decrease in the LOC produced, the cost per source line can be shown to go up rather than down [JONE86, pp. 8]. This similarly applies when using LOC as a quality indicator. Other paradoxes relate to cost per defect and percentage representation. Hence, in order for lines of code to remain a successful size metric, Capers Jones indicated that two fundamental rules must be observed [JONE86, pp. 61]:

" *1. High-level languages will have characteristic rates of code production that are inversely proportional to the level of the language.*

2. The tasks and activities not directly affected by source language (ie. documentation) are increasingly significant as language levels go up."

As the level of the language goes up, so does the difficulty of counting the lines of code. Languages which incorporate graphics or form-filling features does not make explicit a

"line of code". Fourth generation languages fall into this category and require yet another counting convention.

LOC is considered to be language dependent, as one LOC in COBOL need not be equal to one LOC in FORTRAN for example. If comparative studies are required, a conversion factor must be calibrated to show the ratio of LOC between the two languages or to a reference language. This method provides the basis for comparison, but the method should be validated with empirical data [JONE86, pp. 59]. Most Cost Estimation Models continue to use LOC as an input parameter in spite of the problems of LOC [BOEH84]. The reason is to make use of the existing base of knowledge with LOC rather than developing a new unit of measure at the expense of time. Further, Capers Jones believed that the LOC concept is so "deeply embedded" in the programming industry that it cannot be quickly replaced.

The use of LOC in size estimation remains a controversial one as its use in estimation requires a level of detail that may be difficult to achieve in practice. It is still subjective when used in the estimation process as the result is based heavily on the experience of the person making those estimates.

2.1.1.2 Token Count

The use of Token Counts is a departure from LOC due to the prevailing problems associated with counting LOC. This counting technique is a more detailed examination of a line of code by breaking it down into the elementary parts. As will be indicated later, this concept does pose some restrictions to the wider range of languages.

Studies based on token count or data aspect of code are the ideas of the late Dr. Maurice Halstead who proposed a theory known as *Software Science* [HALS77]. Software science views the entire program construct as consisting of two basic atomic particles, ie. the

operator and operand [JONE86]. Operators are functional statements which direct an action, such as read or move data, add or subtract etc.. Operands, on the other hand, are variables and constants used to contain data which is being manipulated within the program. Hence, by counting the number of tokens in a program, ie. operators and operands, several related program characteristics can be studied.

Four basic software science measures are determined as the base for the counting rules [HALS77]:

n_1 = number of distinct operators appearing in a program

n_2 = number of distinct operands appearing in a program

N_1 = total number of occurrences of operators in a program

N_2 = total number of occurrences of operands in a program.

Using the four basic measures, the length of a program is defined as :

$$N = N_1 + N_2$$

and the vocabulary size is defined as:

$$n = n_1 + n_2$$

As noted in [SHEN83], N is closely related to the LOC measure of a program length. When counting operators and operands for machine language, if there is one operator and one operand in a line the relationship between program length and LOC is given by :

$$N = 2 \times LOC.$$

In counting the four basic measures of operators and operands, some variations in counting rules are being used among researchers [CONT86]. For example, Halstead did not include declaration statements and I/O statements. Statement labels were not counted either. However, current researchers maintain that it is reasonable to include declaration and I/O statements as part of the token counts. Most also include labels as

operand counts. As indicated in [SHEN83], there seems to be no apparent standard counting rules in use with software science and SHEN et al propose some consistency measures instead.

From the definitions of length and vocabulary, Halstead derived a set of measures for determining the volume of a program, program level, language level, effort and programming style for a program. As size metrics are discussed here, the length and volume metrics will be elaborated in this section. The other metrics will be discussed in a later section as they are more appropriately categorized under composite metric.

The volume of a program is defined as:

$$\text{Volume } (V) = N \log_2 n$$

Although token counts in software science provide interesting findings in relating data definition and functional statements to programming language measurement, it fails to include structural linguistic concepts of the language [JONE86]. The limitation of using operators and operands does not allow it to be effective when applied to higher level languages, such as the fourth generation languages, where functions and structures takes precedence. Several recent critiques of software science have been published to reveal these inadequacies as well as the anomalies in the theory [SHEN83, CARD87a, CARD87b, JONE86,LI87].

In the paper presented by Card et al [CARD87a], a study was conducted to re-examine the fundamental relationships of measures derived in software science. The study analyses the measures based on Halstead's own data and another larger dataset. In the investigation of the length equation, it was noted that there seems to be a mathematical dependence between the estimated and actual length. Card et al indicated that the estimated length and the actual length are dependent by definition; hence, the

correlation between them would naturally be good. The estimated program length equation is:

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

where

$\hat{N}$ is the estimated program length

n_1 is the number of unique operators

and n_2 is the number of unique operands.

The issue is that in order to maintain the validity of the equation, it will have to be established that the $\log_2$ terms actually have some additional effect on the length of the program, $\hat{N}$. Subsequent results of the study show that there is no significant additional information obtained with the presence of the $\log_2$ terms. Hence, this observation, and others, in the paper indicated that the basic relations of software science lacks theoretical support.

SHEN et al [SHEN83] give a comprehensive summary of criticisms of software science and point out the questionable derivation of some software science formulas. Their criticism of the length equation is that it may be valid in a statistical sense but in reality does not actually represent the length of a program. They suggest that poor programming practices may sometimes introduce impurities into software science metrics and cause high variability in the result.

LI et al [LI87] in an empirical study of software metrics, particularly concerned with the analysis and experimental validation of the relationships among static metrics, revealed some inadequacies with the length equation. In the study, the length equation was observed to be program-size dependent, the estimated length, $\hat{N}$, overestimates the actual length, N , for small programs and underestimates N for large programs.

The basic counting rules of software science are not consistent across programming languages. That is, the counting of tokens is subjective when applied to different languages. The usefulness of software science have been hindered by the restrictive counting of operators and operands only. Jones [JONE86] believes that by including richer structural linguistic concepts, perhaps a wider applicability can be pursued and its usefulness expanded. With their present definitions, software science metrics are not as effective for higher level languages. These include fourth generation languages, graphics languages and other powerful emerging symbologies for software.

However, the use of software science has some significance as a size metric. In a study of over one thousand commercial assembly language and PL/S modules, a linear relationship was found between LOC, the length equation (N), and the volume equation (V) [CONT86].

2.1.1.3 Function Count

Function counts were pioneered by Albrecht in 1979 when he proposed a methodology known as *Function Point Analysis*. The method was initially introduced for productivity studies but was later extended to be used for size estimation. A function may be defined as a collection of executable statements that performs a certain task, including declarations used for data manipulation [CONT86]. The idea of using function count is to allow one to think of functions to be performed rather than the number of lines of code required to perform that function.

a Function Point Analysis (FPA)

Function Point Analysis (FPA) was employed as a technology-independent method for sizing software. The method performs counts on the different types of functions contained within the software to be developed. Essentially, the types of functions

include inputs, outputs, master files and inquiries provided to, or generated by, the software. Points are then allocated to the counts of the functions where a weighting is applied based on the function value to the user. In Albrecht's method, the average weights are as follows [JONE86, pp. 75]:

a. Inputs : multiplied by 4
b. Outputs : multiplied by 5
c. Inquiries : multiplied by 4
d. Master files : multiplied by 10
e. Interfaces : multiplied by 7

By adding up the weighted function counts, the total is the "raw" function points. Further adjustments are then applied to account for the degree of complexity of the software. According to Albrecht, the raw function point total can be modified subjectively for other factors, such as high complexity, for a range of plus or minus 25% [JONE86]. The adjusted total function points arrived at is then representative of the overall functions contained in the software under consideration.

A function point is a dimensionless number which provides a useful comparative index, especially in productivity studies. The use of FPA has been extended and applied to predicting the size of software as it is considered to be a quick and easy method to use and can be applied at an early stage of the software development process [ALBR83].

Albrecht validated his Function Point approach using software science metrics and showed some fundamental agreement between function points and LOC [ALBR83]. In the study, he found a high degree of correlation between function points and the eventual LOC of the program. Subsequent validation also showed a correlation between function points and work-effort. These observations prompted Albrecht to suggest a two-step estimate validation process in which a link between function points, software science

and LOC can be established. This would allow function points to be used with a wider supporting base of knowledge, which are mainly in lines of code, of size data. However, some of Albrecht's studies into productivities revealed some ambiguous results which were later blamed on the paradox of lines of code [JONE86, pp.76]. These applies mainly to high level languages.

The use of Albrecht's Function Point Analysis seems to create some inconsistencies when it comes to counting the number of functions. This uncertainty in counting functions, such as inputs and outputs etc., proves to be quite difficult to resolve. Symons provides a modified version of the FPA approach which he called the "MARK II" approach in the hope of solving some of the difficulties of FPA [SYMO88]. The MARK II approach tends to make the initial counting of functions more objective.

An important contribution of FPA is its ability to be used as a size predictor with subsequent conversion to the equivalent LOC. Although the margin of error is quite high in such conversion [JONE86, pp. 76], its potential as an early size predictor tends to overshadow those flaws. However, FPA is not widely used for software where there is high algorithmic complexity and low data complexity. The main weakness of FPA seems to be in dealing with program complexity. Capers Jones mentioned that FPA is incomplete as a size predictor as "it lacks any effective quantification for the structural aspects of software, such as branching, loops, recursive calls and the like" [JONE86].

b MARK II Function Points

Symons, in his paper [SYMO88], evaluates the FPA method, reveals some areas of difficulty and suggests an alternative to the FPA method called the "MARK II" approach. His critical review of FPA mentions of the oversimplification of classifying the degree of difficulty of system component types. He believes that the weights used are rather

subjective and requires careful calibration for them to be more meaningful. Further, technical complexity factors are inadequate in revealing the internal complexities of the system and the function point counting rules tend to underestimate larger integrated systems as compared to the sum of the same system "dis-integrated" into several smaller discrete systems. He maintains that the larger integrated systems require greater effort to develop and should be weighted accordingly.

The "MARK II" approach looks, instead, at entities rather than files as Symons contends that it is more appropriate in today's database technology. His notion of entity-type relates to modern database practice and relates to data which the system provides as the main body of information. Input, output, inquiry etc. components are all regarded as consisting of logical input/process/output combinations and are referred to as "transaction-types". A collection of these transaction-types then constitutes the application system. Symons models his approach based on these entities and transaction-types to provide the basis for counting the unadjusted function points. "MARK II" formula for Information Processing Size expressed in Unadjusted Function Points (UFP) becomes [SYMO88]:

$$UFP = N_iW_i + N_eW_e + N_oW_o$$

where

N_i = number of input data element types,

W_i = weight of an input data element type,

N_e = number of entity-type references,

W_e = weight of an entity-type reference,

N_o = number of output data element types,

W_o = weight of an output data element type,

and N_i, N_e, and N_o are each summed over all transaction-types.

The above formula is based on the hypothesis that processing complexity is associated with the count of the number of entity-types referenced by the transaction-type. The

calibration of weights is based on the average of work-effort in man-hours on each of the input, output and processing activities.

This data was collected from 12 systems developed independently of each other. In order for the results from the MARK II approach to be compared with Albrecht's FPA, the weights are scaled so that the average system size in UFP, for 8 of the systems which have under 500 UFP, will come out to be identical on both scales. The MARK II formula, based on the data collected, becomes:

$$UFP = 0.44N_i + 1.67N_e + 0.38N_o$$

Based on this equation, Symons contends that it will be suitable for applications similar to those of the 12 systems considered in calibration process. Should there be a substantial difference in the type of application, or technology, then new weightings can be calibrated based on new data. Symons contends that calibration is an on-going process and its credibility will improve with more relevant data collected through time.

The comparison between Albrecht's FPA and the MARK II method indicates a general tendency for the information processing size for MARK II to be larger as system size increases. Also, the MARK II method shows greater sensitivity for systems with relatively high or low average numbers of data elements per input or output component.

The MARK II method is a more objective approach because it is based on a data model. Explicit counts can be carried out as opposed to Albrecht's method where function counts tend to be more subjective. However, the usefulness of the MARK II method tends to be restricted by its technology dependency. The calibration of the weights is dependent on the type of application and the technology used. Its use is necessarily

constrained to only those applications within the some domain. Therefore, it cannot be applied across languages as well as technological boundaries in productivity studies. The separation of technology dependency and language dependency is important if the measure is to be "universally" applied.

The use of FPA has proved to be fruitful though it has its own sources of ambiguity [JONE86]. It is claimed to be useful as a language-independent size predictor but lacks any effective quantification for the structural aspects of the software. However, this new concept provides a fresh approach and its potential as a sizing tool is worth exploring. Symons' MARK II model seems to be a positive step in that direction. A measure applicable across language is useful and, hence, the separation of language and technological dependency for the basic unit of measure is seen to be a positive step to providing a more "universal" measure across-the-board. However, this has not been successfully achieved in either of the FPA or MARK II methodologies.

2.1.2 Structure Metric

It has been noted that experienced programmers tend to look at the overall patterns of programs rather than the detailed coding, which the novice is likely to do [JONE86]. The association of familiar patterns to improved productivity provides an initial interest in studying the complexity of code.

Structure metrics assist the study of the various patterns associated with data or functions of a program. The metric provides measurements for the functions or data to be programmed and the programmed code itself. The former relates to data structure metrics and looks at how data is handled by the functions in a program. The latter is an attempt to measure logical structure; hence, logic structure metrics.

Jackson, Warnier and Orr assert that data is the essence of a system and changes to its structure will affect the overall development effort [JONE86,pp. 69]. However, this is more applicable to business systems than it is to scientific systems, such as real-time or process systems. In scientific systems there are apparently greater processing involved and its influence on the development effort can be considerable. Hence, data structure metrics seem to be more applicable to systems where the data complexity is high and processing complexity is low.

Although some insight has been obtained into measuring complexity using structure metrics, e.g. logic structure metrics; the real relevance of complexity to program size is not yet known [JONE86].

The following sections discuss the two structure metrics used by researchers.

2.1.2.1 Logic Structure

The approach used to study the logic structure of a program and determine its complexity is mainly attributed to the works of Thomas McCabe. He uses a directed graph of program control flow structures as a basis for the measurement. For example, a graph of nodes (or decision points) linked to each other is shown in Fig.1 below, which is taken from [JONE86], and illustrates the control flow of a program.

By counting the number of regions in the resultant graph, the cyclomatic complexity is determined . The regions counted on Fig.1 are marked by Rk where k ranges from 1 to 5.

McCabe noted a certain correlation between the complexity measure and the error-proneness of a program. A program with a higher measure of cyclomatic complexity or McCabe's complexity is likely to encounter more errors. Hence, the implication of

McCabe's study suggests a successful bug indicator. That is, it can be used to predict errors and, hence, at least one aspect of development effort.

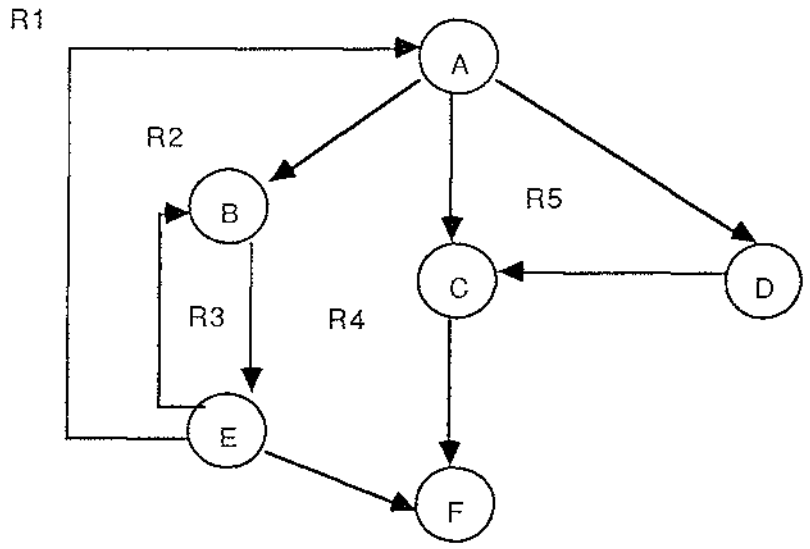


Fig.1: Control flow graph, G, with complexity, V(G) = 5.

McCabe's Cyclomatic Complexity measure is criticized for its lack of empirical support. A critique by Shepperd [SHEP88] provides a detailed evaluation and examination of the deficiencies of this complexity metric. One controversy suggested is the limitation of the code-branching situations considered in the method and, as a result, anomalies arises in the application of counting rules for the wider range of programming languages. The results of various empirical validation studies do not give a great deal of support to the cyclomatic complexity measure, whether it be the predictive value for software characteristics or the testing and maintenance of software. Further, McCabe's measure does not relate the effects of data complexity [JONE86]. This is important in many data-centered based applications as well as high level languages.

The use of the cyclomatic complexity measure has proved to be a successful predictor for errors but remains inadequate in other aspects, such as data complexity. However, this technique provides a good start to the study of this difficult area.

Other logic structure metrics are the Schneidewind and Hoffmann metrics for the minimum number of paths in a program and the reachability of any node. The minimum number of paths is determined by a unique sequence of arcs from the start node to the terminal node. The determination of reachability is to find the number of unique ways of reaching a node [CONT86]. Other logic structures involve nesting levels and transfer usage for direct transfers.

2.1.2.2 Data Structure

Data structure metrics involve measuring the behaviour of data within a module as well as across different modules. It counts the amount of data input to , processed in, and output from a program or software [CONT86]. Most metrics looks at activities related to process data, this metric measurement attempts to capture measurement related to the way data is structured and to determine how they would affect development. Hence, data structure metrics attempts to measure data complexity.

Halstead's count on unique operands, n_2 and total occurrences of operands, N_2 is an example of a data structure metric [HALS77]. However, this metric does not capture the amount of usage which relates the operands [CONT86]. Other data structure measures include counting the number of variables in the cross-reference listings of programs and their relative usage.

As noted in [JONE86], the study of data structure complexity remains difficult as the number of interactions between data elements could be enormous for large-scale file structures. Several design methods have incorporated the data aspect; indicating signs

of the need to have better understanding of data complexity. Although insufficient research has been carried out in this particular area, there is a growing interest in the light of the emergence of high-volume data storage devices.

2.1.3 Composite Metrics

A single metric is used to measure a particular feature of a program. A composite metric uses a combination of more than one of the basic metrics in order to provide a measurement which would better describe the overall behaviour of software.

Several existing metrics used by researchers are, in fact, composite metrics because they attempt to measure aspects of programs based on more than one metric. Software Science uses the basic counts of operators and operands to formulate the many different equations used to describe various aspects of program characteristics. Metrics such as the program length, program volume or potential volume and program difficulty are composite metrics which serve different purposes and extend its influence to some estimation models.

Function Point Analysis (FPA) uses the function point as a basic measure and applies an adjustment factor which deals with the complexity of the software to be developed. This attempt to incorporate complexity measures to function points is designed to provide a more accurate measure. Hence, it is also a composite metric. Capers Jones indicated the usefulness of such combinations and further suggest complementing Albrecht's FPA, which lacks structural complexity considerations, with McCabe's Complexity Metric, which is strong in dealing with structural complexity [JONE86, pp. 78]. There is a good indication that composite metrics will resolve many of the problems which single metrics alone are unable to achieve.

Baker and Zweben also proposed a composite metric of software science, E, and McCabe's metric V(G). Although composite metrics may resolve certain problems, it also has the possibility of introducing confusion to the results if additional metrics are added indiscriminately. Therefore, caution will have to be taken in metric addition for any composite metric to yield useful results.

Another candidate composite metric is the MARK II function point analysis method.

2.2 Review of Software Sizing Models

The attempts in measuring and estimating software characteristics paved the way for subsequent model developments. A combination of the fundamental concepts has been applied in the construction of some of the models.

In a comprehensive survey of software sizing models [DACS87], the Data and Analysis Centre for Software (DACS) presents a detailed review of fourteen different sizing models. The study categorized the sizing models on the basis of their underlying principles and techniques. Six automated models are then applied in an experiment to evaluate the accuracy of the size estimates. Lines of Code (LOC) is used as the output for all the six models for standardization purposes. The result of the experiment shows a wide range of estimates (in LOC) with the highest estimate being 37,600 LOC and the lowest estimate being 6,622 LOC. The actual size of the software developed is 9,177 LOC. This variation shows that the basis of the measurements for each approach does contribute to the size prediction of the actuals. Moreover, the departure of each estimate for each model from the actual suggests that there is still a great degree of uncertainty. A comparison of the basic differences for the four general categories of sizing models are briefly described below.

The first category is *Size by Analogy*. This method requires a database containing information from previous software projects. The approach then relates the proposed project with modules and systems of similar functions and environment requirements contained within the information repository. On this basis, a selection of appropriate sizes for the modules are made and a size estimate of the system is obtained. This category of sizing models are language-dependent as well as technology-dependent and the predictions are based on historical data. The use of LOC as size data in the information repository implies a language constraint. If it is to be applied across languages, more historical size data will have to be accumulated which makes it quite inefficient. Sizing models which come under this category are ESD Sizer, SSA (Software Sizing Analyser) and QSM Fuzzy logic size planner. ESD Sizer gives an estimated size of 37,600 LOC in the test case study. This has the highest departure from the actual size.

The second category is a statistical approach known as *Size-in-Size-out*. This approach uses approximate size estimates and transforms them into a more exact estimate using statistical methods. Some of the methods require the use of size ranges, relative ranking of data, estimates for a portion of the system and/or independent expert estimates. The application of this method requires a thorough understanding of the modularity and functionality of the system; including, perhaps, an expert consensus mechanism. Methods in this category use LOC as an input parameter and statistical techniques are applied to operate on the estimated LOC to produce a more accurate predicted size estimate. As the initial basis of the size estimate (in LOC) is intuitive, they are dependent on the experience of the estimator. This can be subjective. Models in this category are the Wideband Delphi Technique, the PERT (Program Estimating and Reporting Tool) model, SSM (Software Sizing Model) and the Curve-Fitting Model. SSM produces a size estimate of 11,700 LOC in the experiment. This is considered as one of the more accurate estimate in the experiment.

The third category is identified as the *Linguistic Approach*. Based on Halstead's theory on the fundamental relationship between operators and operands, this approach uses the length equation [HALS77] to determine the length of a program. The counting rule is based on counting lexical symbols used in the expressions of algorithms found in engineering equations or pseudo code of detailed design. This method may be useful in scientific applications but not business applications. As engineering equations may be obtained early in the development process, it does serve as an early predictor. However, business applications require pseudo code to be developed before any prediction on the size can be done. This does not allow early size estimates. Models which adopt this approach are the State-Machines Model and ASSET-R (Analytical Software Size Estimation Tool - Real-time) Model. ASSET-R also incorporates function point counts as part of its equation used in the estimation process. In the test case study, ASSET-R produces a size estimate which is one of the closest to the actual. This method is dependent on accurate engineering equations and, hence, is more suitable for scientific application.

The fourth category is *Function Point Analysis*. This method adopts the fundamental concept of Albrecht's Function Point Analysis method as described in an earlier section. Models for this approach include BYL (Before You Leap), SPQR Sizer/FP, QSM Size Planner and Feature Points. This method applied the composite metric concept. Although this category of sizing tools can be applied across languages, the resultant size estimates have to be converted to LOC for comparative as well as productivity studies. As noted in [DACS87], expansion factors used in converting function point to LOC in the implementation language(s) will have to be "corrected" to the current technology if the output in LOC is required for that particular organization.

In evaluating the above categories of sizing models, almost all require some form of historical data to be accumulated if a standardized unit of measure (LOC for example) is required as the output; either for across-model comparisons or between languages

within a model. As size information gathered from previous projects are dependent on the application domain, its use is somewhat limited. This will be further elaborated in a later section. The use of expansion factors for function point conversions are technology-dependent. They are not suitable for wider applications. Also, relevant application domain knowledge may be a pre-requisite to the appropriate choice of models to be used as not all models are developed under the same environmental conditions. An obvious distinction is between scientific and business applications. ASSET-R and SSM models have shown the most accurate estimates in the above test case study. The SSM model uses statistical techniques. ASSET-R uses both function points and token counts, ie. operators and operands. However, ASSET-R seems to be more appropriate for scientific and real-time systems. The case study conducted for the experiment is, apparently, a scientific application.

[DACS87] also identified the need for more experience amongst users of sizing models in order for more accurate estimates to be predicted. They attributed the cause of inaccuracy to the lack of definitive information early in the development life-cycle. This necessarily implies a need for greater experience amongst users if improved results are to be attained. Otherwise some form of intuition is necessary. When "guess-timates" are done, variability in the outcome can be expected.

Language expansion factors are found to vary, for the same language, from vendor to vendor. The main reasons mentioned are [DACS87]:

a. Values based on different dataset
b. Variations in programming skills and language
c. Model inputs tend to be subjective and different analysts may produce different function points for the system. Hence, the lines of code per function point ratio will differ.

Therefore, expansion factors used by different sizing models will, consequently, be inconsistent. This does not assist in productivity studies.

2.3 Need for Improved Sizing Mechanism

Software Sizing Models have some inherent deficiencies. Some models use Lines of Code (LOC) as an initial estimate. The accuracy of such estimates is dependent on experience of the programming language as well as the application domain. LOC have been widely used in research studies involving size estimation. The count of the number of lines of code is said to be related to the effort required in the development process [BASI81,WALS77]. However, LOC is recognized as being too language-dependent. Furthermore, estimates in LOC cannot be determined with any desired accuracy unless sufficient details of design are available. This often does not allow estimates to be determined early enough to be useful.

As an alternative to LOC, Albrecht developed a method of counting the functions that a software is to perform. This approach (FPA) has been described in an earlier section and its deficiencies are duly noted. Several software sizing models, as discussed earlier, have used the FPA method with modifications in the adjustment factors due to complexities etc. The main reasons for the popularity of this method lies in its ease of use and the fact that most of the information required is available at an early stage of the development process. Since the itemization of functions to be used or provided by the application forms the user's external view of the system, it can be determined at an early stage using the FPA approach [ALBR83]. Consequently, this method forms a new theoretical base for sizing models.

The continued use of LOC is required as the base of information available far exceeds that of the function point measure. Albrecht et al [ALBR83] demonstrated that there is a high degree of correlation between function points and the eventual developed LOC of

the program. He also noted that the development effort and the application size in LOC are strong functions of function points and input/output data count. The study is validated using Halstead's Software Science. On this account, Albrecht suggested a validation process to bridge the gap between function points, software science and LOC. This two step approach would allow LOC to be determined from FPA and subsequent work-effort can then be estimated from the available LOC. Such a method can be applied across languages which LOC alone can never do [JONE86]. Hence, the potential of FPA becomes quite attractive and is considered as a candidate for future sizing models [VERN87]. However, there are apparently some technology-dependent factors influencing the FPA estimate. This will be elaborated later in this section.

The use of LOC remains useful in effort estimation as some cost/ effort prediction models still use this measure as an input parameter to obtain the desired schedule [BOEH84]. Significant progress has been made in the area of cost models. Of these, Boehm and Putnam have developed comprehensive software development cost models which can be used for software cost prediction. These two models are quite similar as they are both based (in part) on the same theoretical foundation of the Rayleigh curve. They also build on the experience described by Brooks in "The Mythical Man-Month" [BROO75]. Brooks argued that the addition of staff to a project which was behind schedule would delay that project even further. In other words, when the staff size is increased, productivity will decline. In line with this theory, the Putnam and Boehm models contain a trade-off between elapsed time and development effort, such that if elapsed time for a project is reduced, the total development effort to produce the system will be increased. This protects the model from the many paradoxes of LOC as outlined by Jones [JONE86].

It has been acknowledged that by reducing the language-dependency of software sizing methods, the estimation process can be better controlled. Function point analysis has demonstrated this to a certain extent. Symons' MARK II model [SYMO88] attempts to

resolve the known difficulties of FPA and apply a more definitive approach in determining the required function counts as inputs for the modified version of the function point sizing model. This would provide some solution to the problems suggested in [DACS87] of the lack of definitive information during the concept and design phases of software system development. The potential of using the FPA method as a language-independent sizing tool looks promising, and has already been applied in some existing software sizing models [DACS87]. Its use as a front-end to cost estimation forms a useful mapping between size and cost estimates as described earlier. A model to sizing proposed by Verner et al [VERN87] looks at the usefulness of using FPA-type approach to link system specification to cost models. As identified in the paper, the sizing process is still relatively weak as compared to the cost models.

Notwithstanding the positive value of the FPA approach, some critical issues remain. FPA was initially proposed as being technology-independent but this is being disputed by several critical reviews of FPA [SYMO88, VERN88]. Verner et al [VERN88] indicated that "size estimation of FPA-type metrics are of necessity technology-dependent". Symons [SYMO88] states that it is reasonable to conclude that FPA is not a technology-independent measure of software size on the very fact that the weights used in FPA measurements imply "certain baseline or normative technology for an organization".

2.4 Role of Language Expansion Ratios

Although Function Point Analysis (FPA) does have some deficiencies, its continued use suggests the need for such a language-independent sizing tool. Function points (FP), however, cannot be easily applied across different languages as this unit of measure is not considered "universal" as yet. Its ease of use and estimates of functions has the advantage of being applied early in the development process. It has been quite successful when used within an organization. Because LOC has been a widely used size

measure which forms a comprehensive base of information for existing cost models, it remains a useful metric for cost estimates. The link between sizing and cost/effort estimation is seen as a continuum in the software development process. In order for FPA to be used with existing cost models, which are mainly based on an initial estimate in LOC such as those used in Boehm's COCOMO model, a conversion from FP to LOC becomes a useful extension for FPA. FPA can be used as a front-end sizing tool in the software development process. Language Expansion Ratios serve as the conversion function. Several existing LOC size measures using language expansion ratios as the conversion factor, however, have some inherent variability [DACS87]. Such variability raises the question of its validity. Language Expansion Ratios as applied to the sizing process is illustrated in Fig. 2.2. This is based on the sizing model proposed in [VERN88].

The role of Language Expansion Ratios is to convert function points to lines of code. The purpose of expansion ratios is to enable language-dependent size for a particular language to be calculated based on a language-independent measure with expansion ratio as the size multiplier. As such, Expansion Ratios can be defined as follows:

$$\text{Expansion Ratio (ER)} = \frac{\text{Size } (S_1)}{\text{Size } (S_2)}$$

Where S_1 = language-dependent size (eg. LOC)

S_2 = language-independent size (eg. function points, FP)

The size, S_1, is an objective measure of the actual size (eg. LOC) of the developed program or its component types for a particular language. This language-dependent size can also be the counts of tokens or other counts considered to be appropriate for the language. However, this measure must be objective and consistent for it to yield meaningful results. Size, S_2, is a language-independent measure (eg. FP) based on some design specification. An FPA-type approach can be adopted to provide an adequate measure of the functions required to implement the system (independent of language

considerations). Expansion Ratios, so determined, will act as a language-dependent size multiplier.

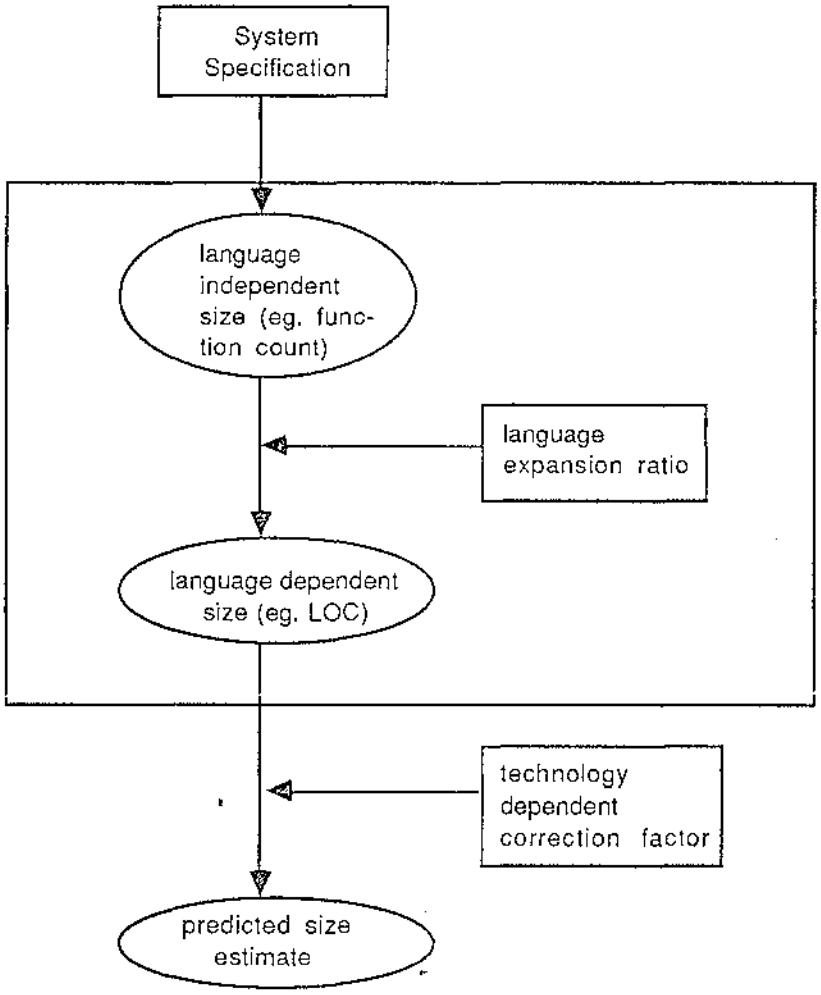


Fig. 2.2 Multi-language sizing mechanism.

The concept of language level can be considered to be closely related to expansion ratios from some irreducible function size. The use of language characteristics is deemed as an important influence in language expansion for the different levels of programming languages. Hence, the level of a language can be related to the expansion ratio(s) to provide a basis to the sizing models. The wide range of programming languages available requires a means of establishing a productivity gauge in order to determine

the most suitable language to be employed to a particular software project. The use of expansion ratios should serve to provide a means to those ends.

Expansion ratios also allow language-independent size estimates to be made without the influence of implementation language(s). This should enable more objectivity in determining the initial language-independent size estimate. Further, with a common reference measure (such as the FPA-type method as the common denominator), comparative productivity studies of applications and languages can be carried out without the influence of implementation factors. Also, a standardized unit of measure should assists in enhancing such studies. However, this will have to be established when calibrating expansion ratios from empirical data.

The use of expansion ratios is important if a consistent measure is to be used across languages. In spite of the known variability of expansion ratios from different sources, they are becoming increasingly important. The purpose of this work is to provide some empirical evidence of the validity, or otherwise, of language expansion ratios, in the light of its advantages.

CHAPTER 3

OVERVIEW OF CASE STUDY

3.1 Research Objective

The purpose of this thesis is to study ***Language Expansion Ratios***, and the factors affecting them, for several languages based on a specific case study. The case study is provided by the IFIP working group 8.1 conference which contained the business analysis and system design specification for an Inventory Control and Purchasing System. A detailed specification is attached in Appendix A. This case study is particularly suitable in that it has unnecessary complexity removed in order that a more tractable study can be conducted. However, the implementation of the system is still a non-trivial task.

The selected languages to be employed in the implementation of the above case study are:

a. Micro Focus Level II Cobol

b. Advanced Revelation

c. Informix 4GL

These programming languages are suitable for the development of data-centred business applications and are appropriate for the purpose of comparative studies.

This thesis attempts to investigate the following areas:

(i) The relationships between the languages, the application and the expansion ratios.

(ii) The inter-relationship of expansion ratios across languages.

(iii) The usefulness/relevance of expansion ratios in relation to software sizing.

The computation of the expansion ratios, based on the definition in section 2.4, uses lines of code as the language-dependent size and function points as the language-independent size. An overview of the research is shown in Fig. 3.1.1. In the diagram, ER_A, ER_I, ER_M are expansion ratios from a single largely implementation-independent size in function points (FP) to the specific implemented sizes in lines of code (LOC).

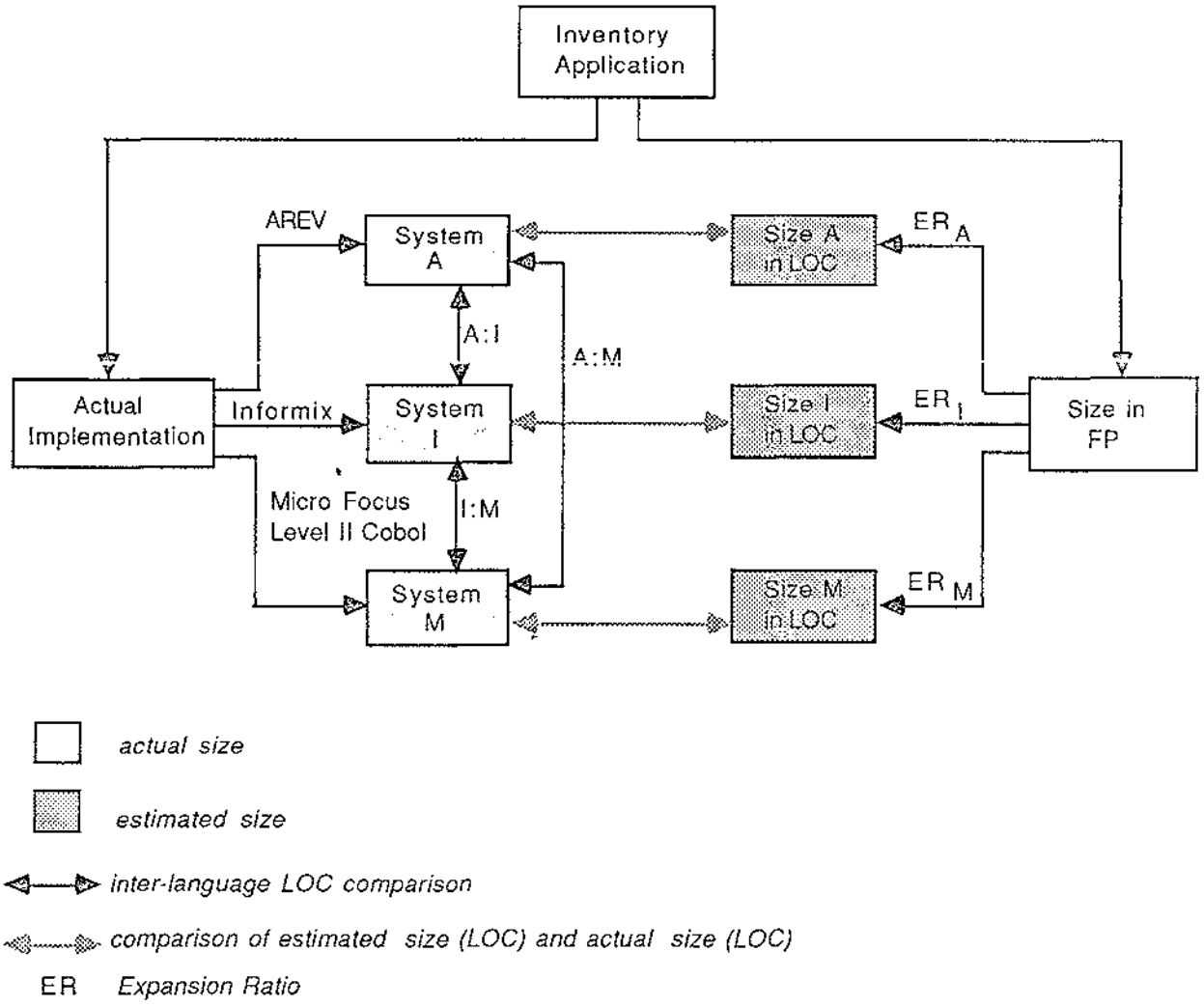


Fig. 3.1.1 Overview of Research

The actual implementation in each of the selected languages provides the basis for comparative study of sizes in lines of code. Expansion ratios are calculated and used in

a detailed study of the differences between the languages. A comparison of estimated and actual size for each language is also included for the purpose of examining the effects of expansion ratios in their present form.

3.2 Case Study Specification - Overview

The specification for the system to be implemented is based on a typical Inventory Control and Purchasing Application. The application is for a small company which maintains stock of various items for its own use. It is assumed that all stock is kept at one location. When re-ordering for replenishment is necessary, the company would order new supplies from one of its suppliers. The application should, therefore, enable records to be kept of the type of stock items available from each supplier and maintain the current status of stock-on-hand for all its inventories.

The system requirements are carefully analysed in two separate phases adopted:

a. business analysis phase
b. system design phase.

Each phase is scrutinized to provide informative as well as definitive specifications.

The analysis carried out in the business analysis phase involves eight related steps. They are:

1. Business activity analysis
2. Entity type identification
3. Entity type relationship analysis
4. Attribute analysis
5. Population analysis

6. Relationship cardinality analysis
7. Business activity to entity type cross reference
8. Feasibility of design alternatives.

The business activity analysis defines the area of interest and categorizes activities into main and subordinate activities. This provides a hierarchical breakdown of the different activities within the system and enables an overview of the application to be shown.

Entity type identification specifies the entity types related to the activities identified in the business analysis step. Entity types are classified by entity type category as well as dynamic class. The dynamic class gives an indication of the frequency of use of the different entities of the entity type categories. Entity types are classed as :

a. Categorizing
b. Fundamental
c. Cross reference
d. Event.

This step may be iterative.

The structure of the interaction between entity types is analysed in the entity type relationship analysis step. A resultant data structure diagram is then drawn to show the relationships between entity types; either in a one-to-many or one-to-one relationship.

Attribute analysis then follows to present a description of the attributes of each entity type. The description provides detailed information with tables showing attributes and

unique identifiers. An example of attribute analysis for stock-on-hand is shown in Appendix A.

The cardinality of a relationship is the number of entities of one type which correspond to an entity of the other type. Relationship cardinality analysis assesses the cardinalities of relations between two entity types to identify potential problem areas. This step is important to both the understanding and the performance of the system to be designed.

The business activity to entity type cross reference consolidates the information gathered and provides a cross-reference table to associate entity types identified with the business activity categories. This allows inconsistencies to be detected. The business activities may be revised to remove inconsistencies.

Based on the business analysis phase, the system design steps follows. This involves nine related steps. They are:

1. Entity type to table assignment
2. Attribute to column assignment
3. Table specification
4. Definition of referential constraints
5. Task definition
6. Specification of stepping stones between tables
7. Task to menu assignment
8. Task to table cross reference
9. Assignment of tasks to access control classes.

The steps involved in the system design phase use the information established in the business analysis phase and, where necessary, include some modifications. Tasks are

assigned to each table associated with an entity type. These tasks include add, modify, delete, and retrieve. Having identified all the tasks necessary for the system, they are then assigned to related activities categorized in the business activities phase. This is done by drawing up appropriate menu structures for each activity and the associated tasks. A cross-reference table is also set up for tasks and tables (entity types) where the information flow between tasks and tables is illustrated. These can then be used in data flow diagrams.

The design philosophy follows a fully normalized data design concept as a cornerstone of the approach where uniqueness and referential constraints are considered. It attempts to express the semantics of data in the data design instead of leaving such expression to the specification of individual program. Tasks are analysed to minimize "cascading" of any kind, eg. in the case of delete or modify. That is, the updating of a table does not require a similar update to an associated table. Quantitative estimates of data sizes and volumes are provided to enable potential problems to be predicted. The design methods used attempt to strike a balance among four important factors, namely flexibility, comprehensibility, completeness, and compactness. Above all, the system is designed to provide a tractable case study. A detailed specification can be referred to in Appendix A.

3.3 Language Implementation

For a comparative study of this type, it is necessary that the application domain is identified for two reasons. Firstly, it "has the advantage of restricting the variability within the domain of interest" [VERN87]. Secondly, the emergence of application development languages for the business data processing environment, such as the fourth generation languages, makes it advisable for the purpose of this thesis.

The following describes the three different application development languages used, with each having its own distinct features. However, all of them are suitable for developing data-centred business applications. A data-centred application is, basically, one which is built with a database as the central repository of information and process components organized in terms of the flow of data into and out of the database. All of the three languages specified below have features suited to this type of application domain but applied in different ways.

3.3.1 Advanced Revelation

Advanced Revelation (AREV) is a fourth generation language mainly used for business application development. It has several standard built-in features which assist in tasks such as data validation, query construction, screen painting, process calls, etc. A comprehensive help facility is also provided as a guide for users. A set of nine basic application tools are provided as the base for application development. The following briefly described each of these tools in relation to the development function. The basic idea of AREV is to eliminate much of the procedural code normally required in an application development.

a. Tools That Build Tools

The basic philosophy of AREV is to provide a basic set of tools for application development with the added versatility of improving the language environment's capability, when necessary. Extension to an existing set of tools can be carried out using the basic tool-set available. This enables new tools to be built when needed. Enhancement of the environment can, therefore, be achieved to satisfy various development functions.

b. Paint

The paint tool is used to design data entry windows. It also allows the behaviour of the window to be controlled by various pre-, post-, or mid-window processes. It can be used to create prompts on windows with behind-the-scene processing options for each prompt. A prompt is a data-entry field defined on the screen.

c. Popup

A popup is synonymous with an inquiry where details of a file can be displayed, by pressing a function key, with options to sort different columns of the displayed information as well as for selective displays. No recursion is allowed with the use of popups, so terminating a popup returns to the previously displayed window.

d. Menus

Menus allow control to be passed to different areas of an application. Control can be passed to other data entry windows, R/LIST reports, the editor, the operating system or other menus. Menus can also be called from R/BASIC routines.

e. R/Basic

This is an interpreted procedural language, used for coding program segments, to be used either with windows or on its own. All variables and constants used in R/BASIC are variable-length dynamic strings which allow great structural flexibility. Variables can contain file handles, interpretive object code, dictionary information, dimensioned arrays, dynamic arrays or floating point numbers. It also provides statements for the use of non-procedural query language within its code. This is extended for use with external routines.

f. Editor

The editor provided is a full screen editor. It is used for writing R/BASIC source code, to edit records of any filing system, as well as in writing batch files and operating system commands files. Object code can also be edited using this editor. Various screen prompt entries used throughout the system also have the underlying screen handling features provided by the editor. Hence, the editor can be used right across the system environment.

g. Macros

Macros allow users to define a process and then attach that process to a particular key. Processes such as a data entry window, menu or report can be attached using macros. The basic idea is to enable access to various parts of the environment with a keystroke. This suspends the current process and executes the process invoked.

h. The Command Level (TCL)

The command level executes operating system commands. However, all other commands, tools and application window can be accessed from TCL. This facility applies the basic macro concept of process invocation. A stack of levels of TCL is maintained by the system.

i. Operating System

AREV is an application that builds and maintains applications within itself. It has its own native Pick-based operating system. However, it still preserves interfaces to

other applications as well as the external operating system (eg. MS-DOS). All operating system services can be accessed through the AREV environment.

3.3.2 Informix 4GL

Informix 4GL is a language which contains more procedural language concepts in its language structure than most other 4GLs. The rationale for the greater proportion of procedural statements is to allow versatility in software application design. The power of the non-procedural parts forms the basis for productivity enhancements.

Informix 4GL is based on the relational database management system concept. It provides tools to create databases and database languages to interact with the database. It, therefore, provides a data-centred approach to application development. The following discusses, briefly, the features available on the Informix 4GL environment.

a. A Database Language

As the basic design concept is for relational databases, Structured Query Language (SQL) is implemented for high-level accesses to databases. It meets the standards established by ANSI for SQL implementation. SQL statements are embedded within the programming language but are executed at run-time by the underlying "SQL engine".

b. A Programming Language

Informix 4GL still retains features of the third-generation programming languages and, hence, it serves to provide those general-purpose capabilities. Applications can be constructed by modules which are then compiled and linked using the program compilation option. The language also enables calls to external routines written in the C language. Language statements are designed to be at a higher level and line delimiters

are not required. However, the language statements have a specific syntax whose format has to be followed.

c. Screen-Building Utility

Screen forms can be created using the utility provided by FORM4GL. It allows Forms design to be painted in a straight-forward manner and can be done with a minimum effort. Specific forms can relate to specified files in the database and each form can access several files. Screen forms can also be used as a "FORMS ONLY" display where it serves as a screen input facility to be used in processing. Some standard data validation facilities and automatic displays, such as date and time, are incorporated as part of the features of this screen-building facility.

d. Menu-Building Utility

Menus can be created using the MENU statement that simplifies the process of menu creation. This facility is a standard menu display which occupies two lines at the top of the screen. Options on the screen are differentiated by their first character and are, therefore, limited in their range of expressiveness. However, this should not be a problem as menus are generally small in size. Furthermore, different levels of menus can be constructed to allow for the different categories of activities. This extends its usefulness for larger applications.

e. Report Writer

Similar comments to those on the MENU statement apply to reports. It provides features such as ordering, grouping or formatting of data to a required specification. It also has a built-in feature which allows calculations within the report. The user can set up the required report format using the report writer. Selected records from files

using SQL can then be passed to the module where the formatting is done so that the required output can be generated.

3.3.3 Micro Focus Level II Cobol

Micro Focus Level II Cobol or Level II Cobol comprises four basic software tools. These are the Level II Cobol compiler, Animator, Forms-2 and Fileshare. Although Level II Cobol is considered a third generation language, the added software tools have enhanced its development functions. Standard library routines for screen control and function-key settings are available to provide greater language capability in screen and key-board control. Language extensions also include a screen-painting feature for formatting CRT screens for input and output. Additional features adopted in Level II Cobol form coherent set of extensions in line with existing fourth generation languages, eg. screen-painting. The following briefly discusses the four software tools provided by Level II Cobol.

a. Level II Cobol Compiler

Level II Cobol compiler functions as a translator from a source program into its equivalent object program where syntax errors are indicated for incorrect use of the language statements. Programs are created using a conventional text editor. The compiler compiles the programs and the run-time system links them with the compiled output to form a runnable program. A listing of the program is provided by the compiler during compilation with error messages inserted in the listing where necessary. The compiler supports sequential, relative and indexed-sequential files as well as interactive communications via the Accept and Display verbs.

b. Animator

The Animator enables the internal workings of the source program to be shown by displaying the source program on the screen while it is running with the cursor moving from statement to statement. It is used for running and debugging programs and allows the user to follow through the logic of the program code. The Animator is a full screen symbolic debugging tool which enables the examination and alteration of data, setting breakpoints and changing the flow of control where necessary.

c. Forms-2

Forms-2 is a useful screen painting facility which can be invoked during program coding. Once invoked, it enables the user to define desired screen formats for data inputs or outputs by a straight forward screen painting process. Two modes are provided to differentiate between text description and field definition. A screen layout defined using Forms-2 will automatically produce the relevant data division statements for direct inclusion in the Level II Cobol program. The data division statements are stored under a copy file name in the library. A copy statement is included in the source program for that data description segment generated.

d. Fileshare

The Fileshare provides for a multi-user environment where programs can be run simultaneously to access the same files. Concurrency is supported through the file locking facilities provided. This should provide control for selective access by users.

CHAPTER 4

METHODOLOGY FOR EMPIRICAL STUDY

Based on the Case Study specification outlined in section 3.2, the following system development tasks are included in the subsequent implementation phase :

(i) Detailed design and implementation

(ii) Size and cost/effort estimates

(iii) Data collection.

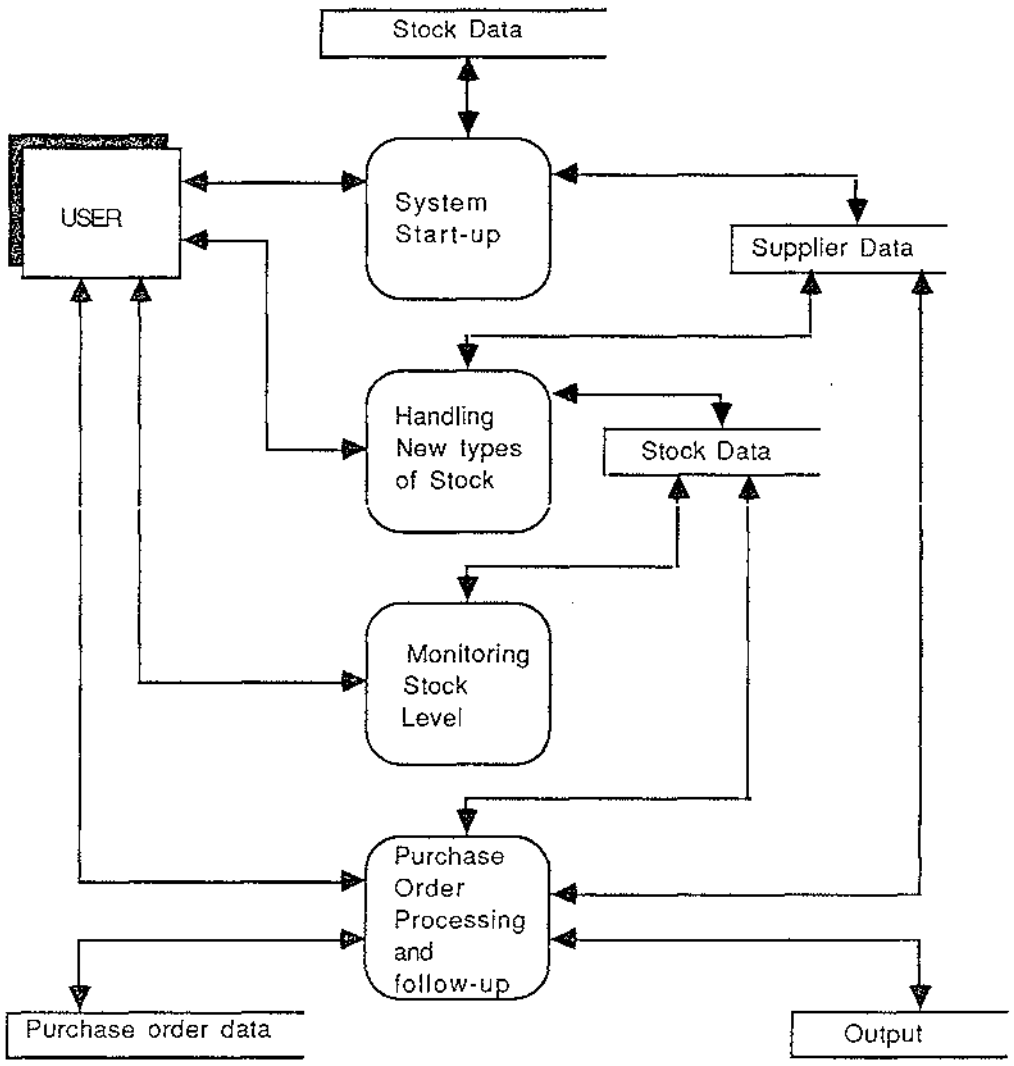
The detailed design forms the basis for the implementation in the three selected languages; namely Advanced Revelation, Informix 4GL and Micro Focus Level II Cobol. The following sections described the method applied in the implementation of the case study.

4.1 Detailed Design and Implementation

The task to entity-type cross-reference table of the design specification provides the necessary details required for the detailed design. Initial data-flow diagrams (DFD) for the four main processes are designed and each process is, subsequently, exploded to the level of detail required for the design of the structure charts. All information flows for the tasks within each process are provided in the cross-reference table (see Appendix A, pp. A-32). Fig. 4.1.1 shows the top-level data-flow diagram (DFD0) which is from a direct mapping of the four main activities in the main menu (see appendix A). The first level data-flow diagram expands each main activity to the group of tasks required for that activity. A second level explosion is deemed necessary for the

essential details for each task to be shown. Both the first and second level data-flow diagrams are presented in appendix B.

FIG. 4.1.1 Inventory Control and Purchasing System (DFD0)



The second level data-flow diagrams are then used to derive the structure charts where details of the flow of information into and out of each transaction is shown. The detailed design provides a visual representation of the system to be developed. Subsequent implementation is then carried out using the diagrams from the detailed design phase.

SYSTEM START UP MENU

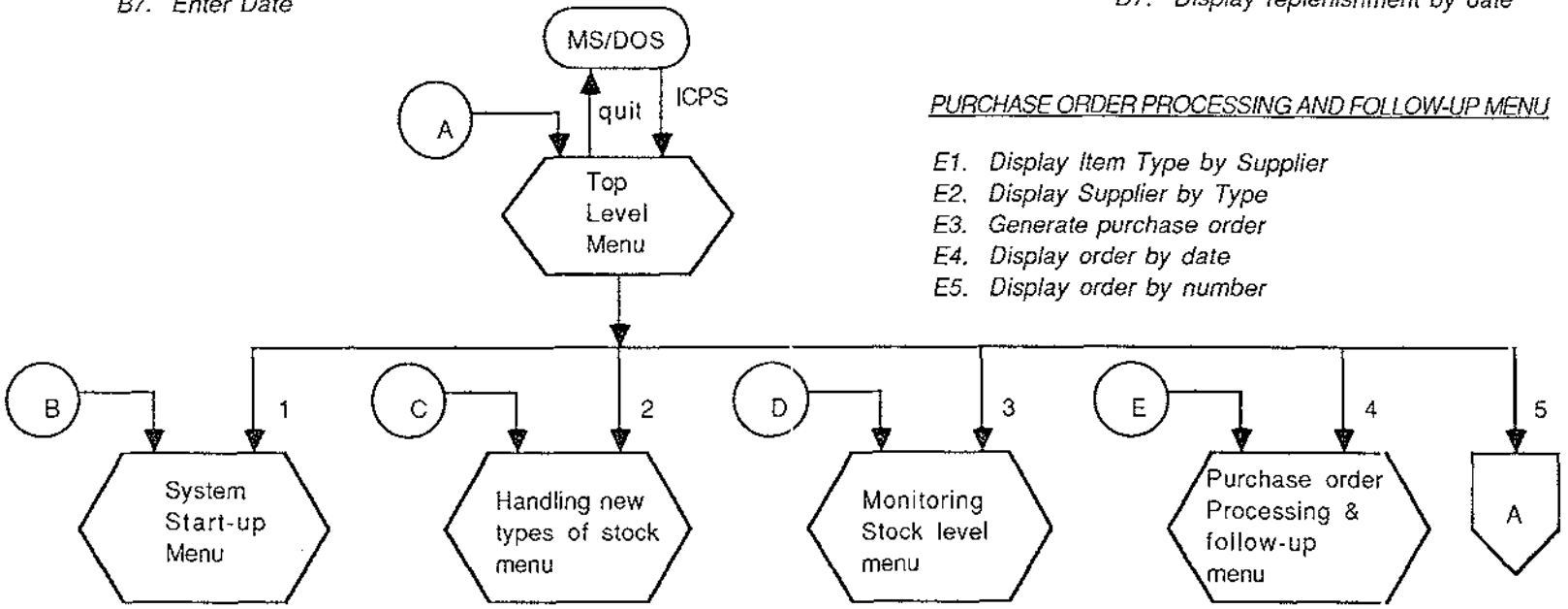
B1. Add Stock Item Type
B2. Modify Item Type
B3. Display ilem Types
B4. Add initial List of Items
B5. Add initial Suppliers
B6. Add Item Type for Supplier
B7. Enter Date

HANDLING NEW TYPES OF STOCK MENU

C1. Modify Item
C2. Display List of Items
C3. Modify Supplier
C4. Display List of Suppliers

MONITORING STOCK LEVELS MENU

D1. Manual Recording of SOH
D2. Display SOH by dates
D3. Display SOH by type
D4. Record stock withdrawal
D5. Display withdrawal by date
D6. Record stock replenishment
D7. Display replenishment by date



PURCHASE ORDER PROCESSING AND FOLLOW-UP MENU

E1. Display Item Type by Supplier
E2. Display Supplier by Type
E3. Generate purchase order
E4. Display order by date
E5. Display order by number

FIG. 4.1.2 Modified State Transition Diagram - User-interface for Inventory Control and Purchasing System

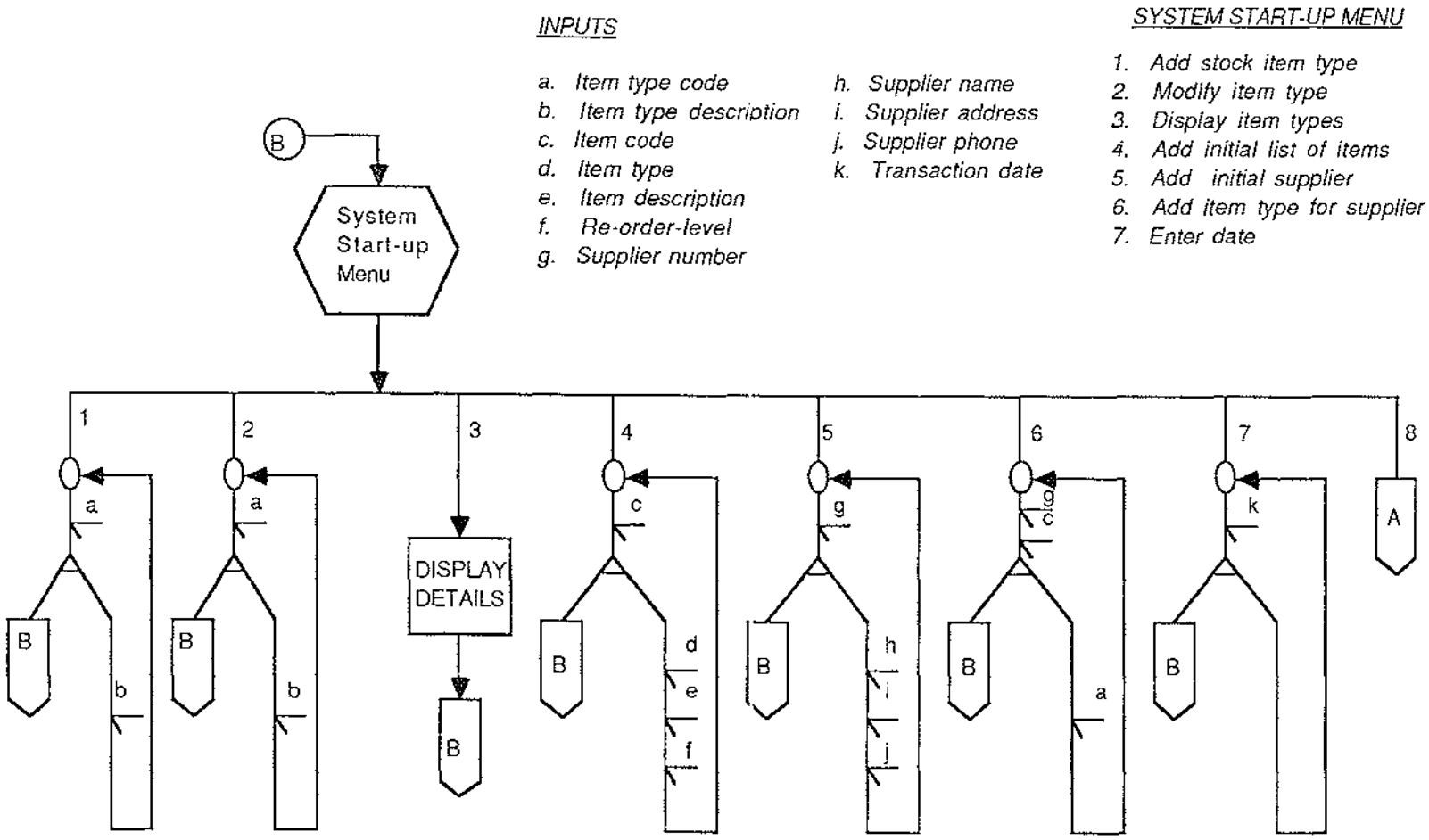


FIG. 4.1.3 User Interface for System Start-up Menu

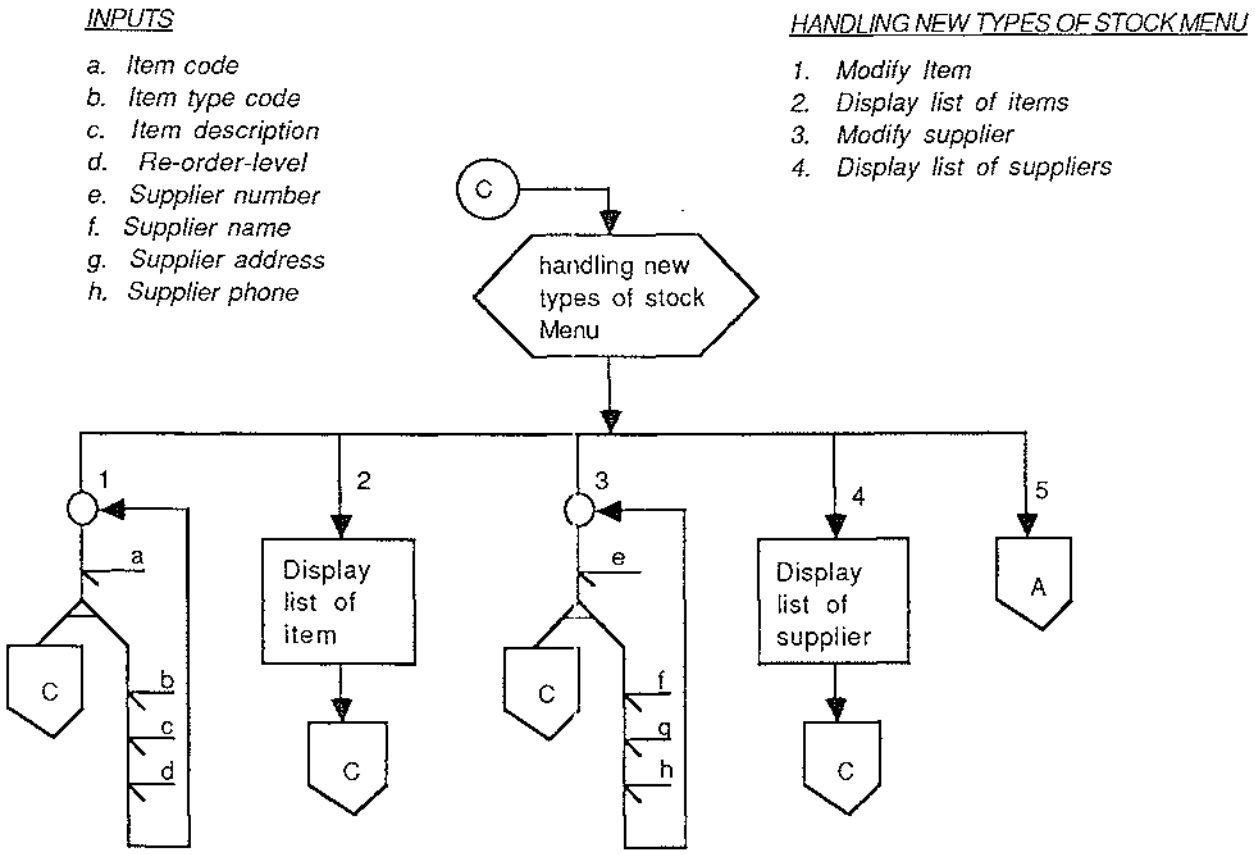


FIG. 4.1.4. User-interface for Handling New Types of Stock Menu

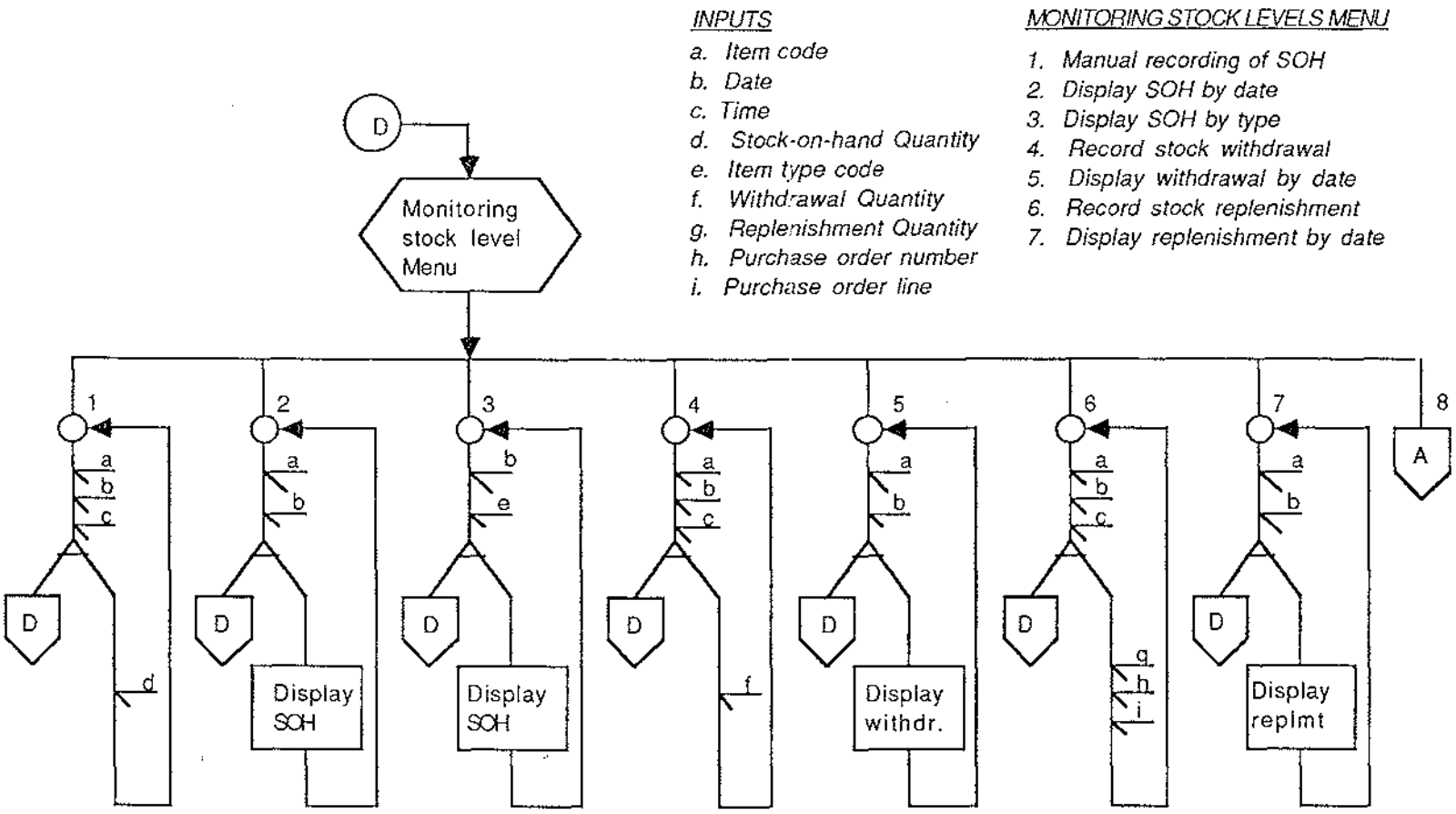


FIG. 4.1.5 User-interface for Monitoring Stock Level Menu

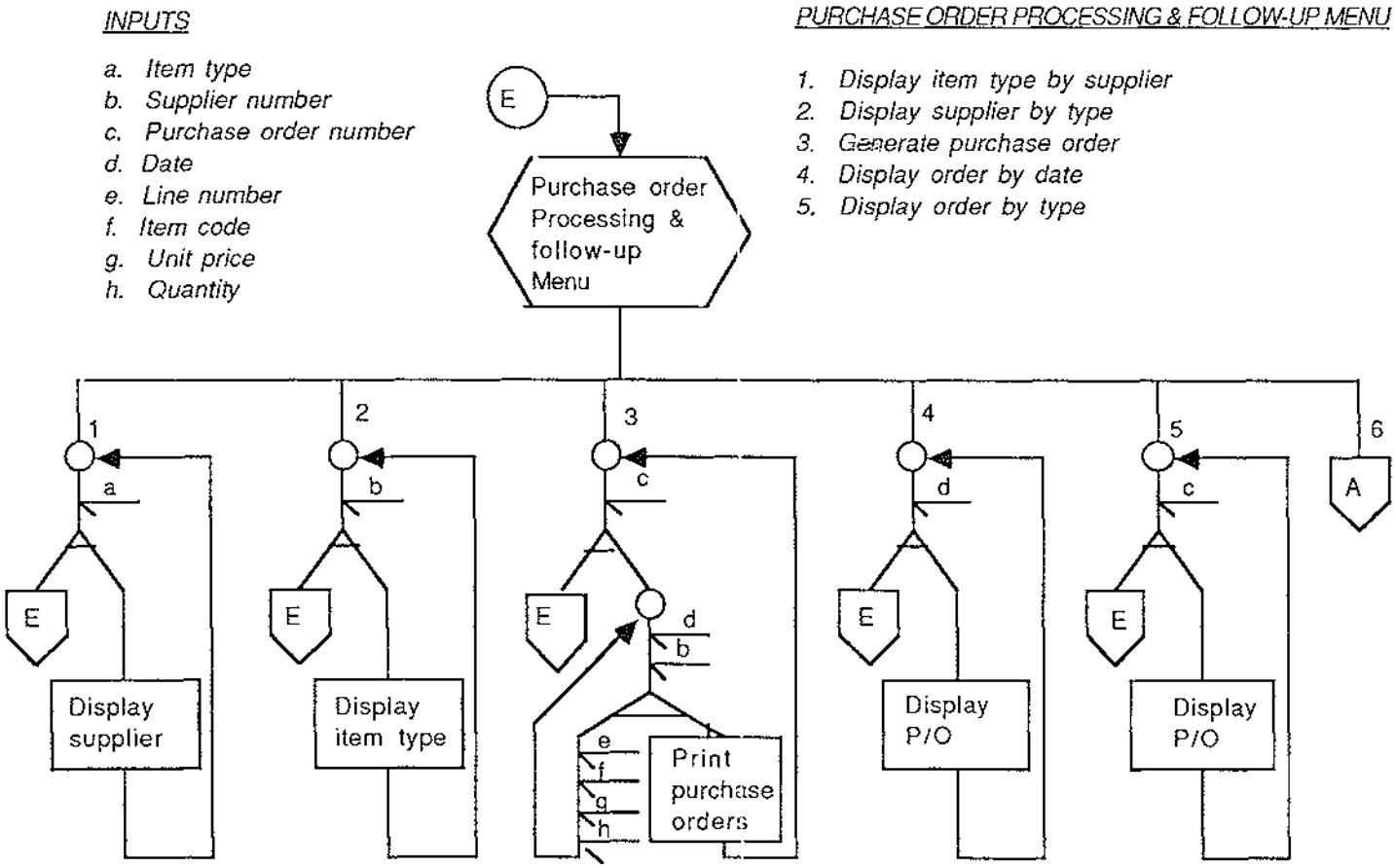


FIG. 4.1.6 User-interface for Purchase order Processing & Follow-up Menu

The application to be developed is an interactive system and, hence, a suitable user-interface will have to be included for ease of use of the system. Depending on the language used, the user interface may differ between the different languages. As it is essential that the user interface be implemented in a consistent manner across the three languages, a user-interface is designed using the modified state transition diagram. This is shown in Fig. 4.1.2 to Fig. 4.1.6.

As the object of the design specification is to eliminate cascading of any kind, each task within a process can be referred to as a transaction type. This implies that a task or program basically operates on the underlying database and perform such functions as add, delete or modify. Hence, each task performs a particular function and is independent of the other tasks. Each task can, therefore, form a module or program and the application comprises a collection of these tasks. Such an organization for an application can be termed as data-centred. This is useful for incremental development.

There are three different languages in which systems have to be implemented using the same design produced in the detailed design phase. As the familiarity of the author with two of the three languages (ie. Informix and AREV) was relatively low, it was decided that the system was to be developed incrementally. This development strategy is appropriate for two reasons. Firstly, the modular nature of the application allows modules to be developed independently. Secondly, familiarity with the new languages should improve as development progresses through the different increments. The familiarity with the fourth generation languages was not considered as being adequate to allow for a smooth implementation of the required system, even with incremental development. Therefore, prototypes in the two fourth generation languages were implemented first before the actual system implementation. This should eliminate superfluous exploratory details. The lines of code count is a critical measure in the final analysis of the results and, as such, the above consideration is an important one.

As the programs consists of four distinct types; namely add, modify, delete and display, each category of one or more of these four basic types which operates on the same files are to be implemented at the same increment. This should allow records added to be checked using the display program for example instead of the need for additional programs to be developed to perform such testing. Fig. 4.1.7 shows the four increments adopted for the implementation. Each subsequent increment will be able to make use of existing files created in the previous increment. The design specification which emphasized the data design, where data dependence between modules is shown, has assisted in determining the sequence of events for incremental development. Test data for the first increment can be used for the next increment and so on. The method should enhance the testing for each increment.

The three languages are implemented one after another. That is, one language implementation is fully completed before the next begins. Of the 28 programs in the design specification, 24 of them are to be implemented using each of the specified languages. An initial familiarisation period for each language is deemed necessary before actual implementation. This serves to provide an organized approach to using the language.

4.2 Size and Cost/Effort Estimation

Size estimations are carried out using Albrecht's Function Point Analysis (FPA) as it is considered an appropriate language-independent size estimation tool for the purpose of this case study. An initial size estimate in function points (FP) is obtained based on the design specification. This is done independently of implementation language considerations. As the author's familiarity with the fourth generation languages is relatively low, estimates in lines of code will not be used. Instead, published Language Expansion Ratios will be used to give an approximate size expansion from FP to LOC for

each of the three languages. As a general guide-line, the language expansion ratios for each of the implemented language are as follows :

a. Micro Focus level II Cobol 105 LOC/FP [JONE86]

b. Advanced Revelation 25 LOC/FP [PRES87]

c. Informix 4GL 50 LOC/FP

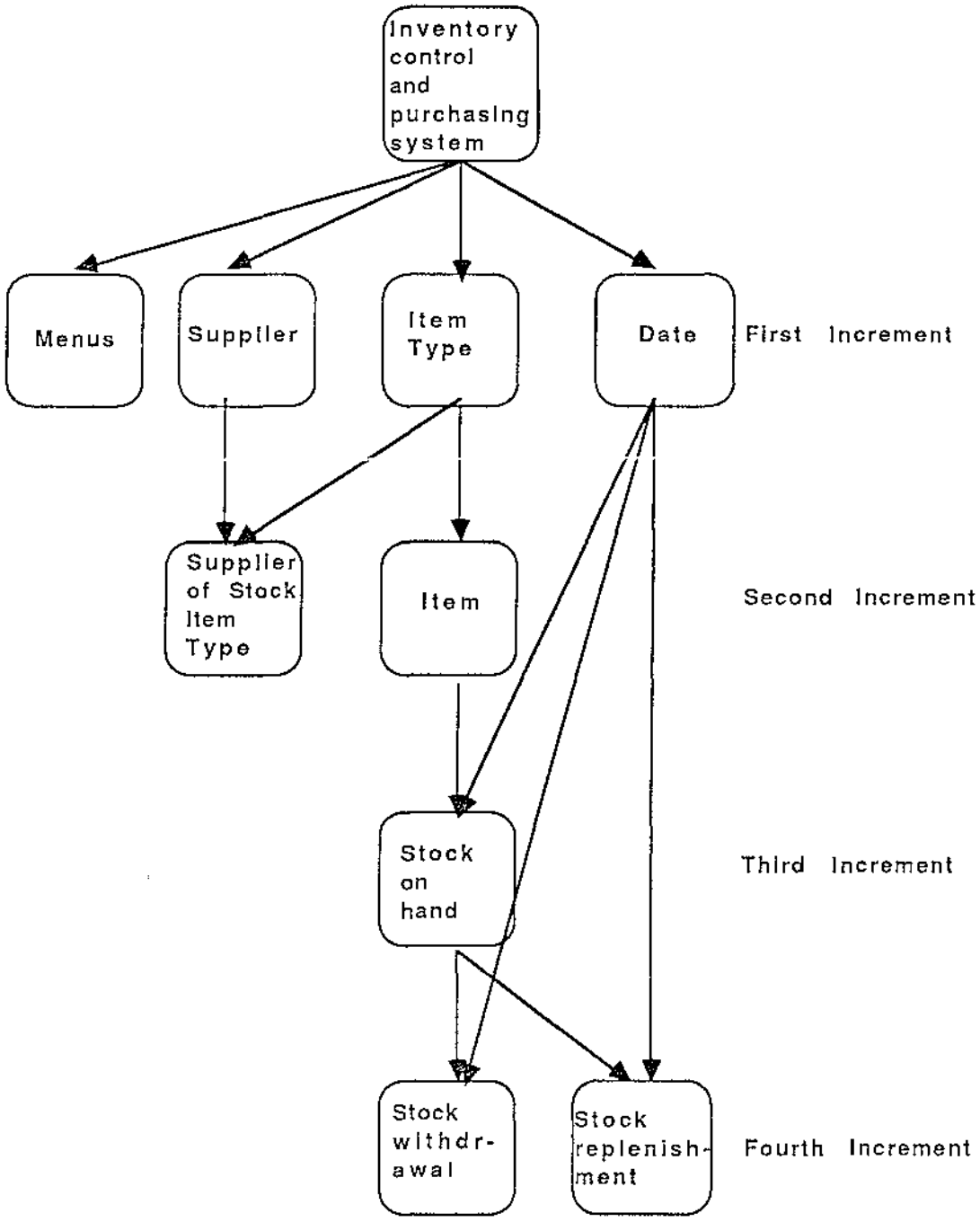


Fig. 4.1.7 Incremental Development

The expansion ratio for Level II Cobol is a commonly-used ratio and therefore is being applied accordingly. For fourth generation language (4GL), Pressman [PRESS87] provides a summary of works which attempts to relate FP to LOC and gave an average relationship for a 4GL (a simple data base query language) to be 25 LOC/FP. As noted in [PRESS87] , one line of 4GL code is between two and four times the functionality of a conventional programming language. In considering the features of Advanced Revelation 4GL, such as form-filling features and query language etc, the 25 LOC/FP seems appropriate and is used for the size estimate for the language. Verner et al [VERN88] provides a contraction factor of 0.16 from Cobol code to give an equivalent size for a 4GL. Since it was used in size estimation for a fourth generation language, its use is appropriate for this study. The size estimate using the contraction factor of 0.16 can be applied as a cross-check with the language expansion for Advanced Revelation 4GL and will be used in the initial size estimation process. Informix 4GL, however, contains substantial procedural code and is quite different from many other 4GLs. Although Informix 4GL provides a query language as well, the procedural code does not provide the same functionality as the form-filling features of AREV. Therefore, it is considered appropriate that the functionality provided by Informix 4GL to be about two times that of Level II Cobol or doubled the language expansion for Advanced Revelation 4GL. Hence, an approximate language expansion ratio arrived at is 50 LOC/FP.

The size estimate obtained is then used with the WICOMO (Wang Institute's implementation of COCOMO) model to produce the cost/effort schedule estimate. The cost/effort estimate will be used for the comparative productivity studies. The initial language-independent size estimates in function points are reviewed as development progresses. This is necessary should there be any changes to the design. All three languages are implemented using the microcomputer system operated under MS/DOS.

4.3 Data Collection

The use of a time log is required to record all time spent in the development process. This provides the basis for the comparative studies of the estimated and the actuals. The essential details for the time log should account for all tasks carried out during development. As it is considered important to ensure all time periods spent in development are in at least three hours block sessions, this is not always possible. Hence, all effort was made to ensure a minimum of interruptions so that there will not be a break in the train of thought. Problems encountered during the implementation are recorded in the time log so as to account for the delays in the estimated schedule in person-months.

4.4 Software Measurement

The measurement of the developed code posed some initial problems. As there are no appropriate baseline measurement definitions for fourth generation languages, a measurement definition in conformance with some of the appropriate counting rules established in [JONE86,VERN88] will be used instead. The measurement of all codes produced will be counted in terms of lines of code (LOC). The counting rules applied for the counting of the programs are as follows :

a. Executable statements, data definition, and labels are considered to be the elements of a line of code. Executable statements include commands, verbs, and functions.
b. Comments and blank lines are not counted.
c. Each physical line is counted as a line of code. A line of code is generally maintained within the 80 character width limit (ie. the width of the CRT screen) for the code produced.
d. Deleted code is not counted.
e. Temporary code is not counted.

f. New functional extensions to a module, such as inquiry etc, will be counted only once irrespective of the number of times it is used throughout the system.

g. Re-used code will not be counted if the basic function(s) is/are not altered.

h. Tools and support programs to support development and testing will not be included in the line count.

The above counting rules are applied for counting Level II Cobol lines of code. Some modifications, however, are necessary to accommodate the fourth generation languages. Informix 4GL has a different approach to forms design. To be consistent, each line of code on the forms module will be counted as a line of code. Therefore, all other line types will be counted as a line of code except for comments or blank lines. Other rules as defined above are applied accordingly for Informix 4GL. Advanced Revelation 4GL provides several form-filling features as well as window painting facilities. Again, for reasons of consistency, lines of code will be counted for all text or prompt lines painted on the screen as well as executable statements created using R/BASIC or RLIST. For prompt windows where form-filling features are used, a group of prompt options which describes or validate a particular aspect of an object will be counted as a line of code. Codes generated in process calls will be included in the module where it appears. Re-used code will be applied according to rule g above.

The basic counting rules established above tend to eliminate duplication, especially that of re-used code. As re-used code does not normally contribute to extra effort, its inclusion is not regarded as essential. The consistency observed in the rules serves to remove any bias when studying the language expansion for the three implemented languages.

4.5 Analysis of Data

The analysis of data collected is carried out in a series of related steps. Fig. 4.4.1 illustrates the steps by which the analysis of the data is conducted. The essential elements will involve an iterative process in view of the likely changes in the problems associated with such experimental study.

The statistical method of regression analysis will be used to determine relationships among the different sets of data. Evaluation of those results should provide relevant information as a basis for making inferences.

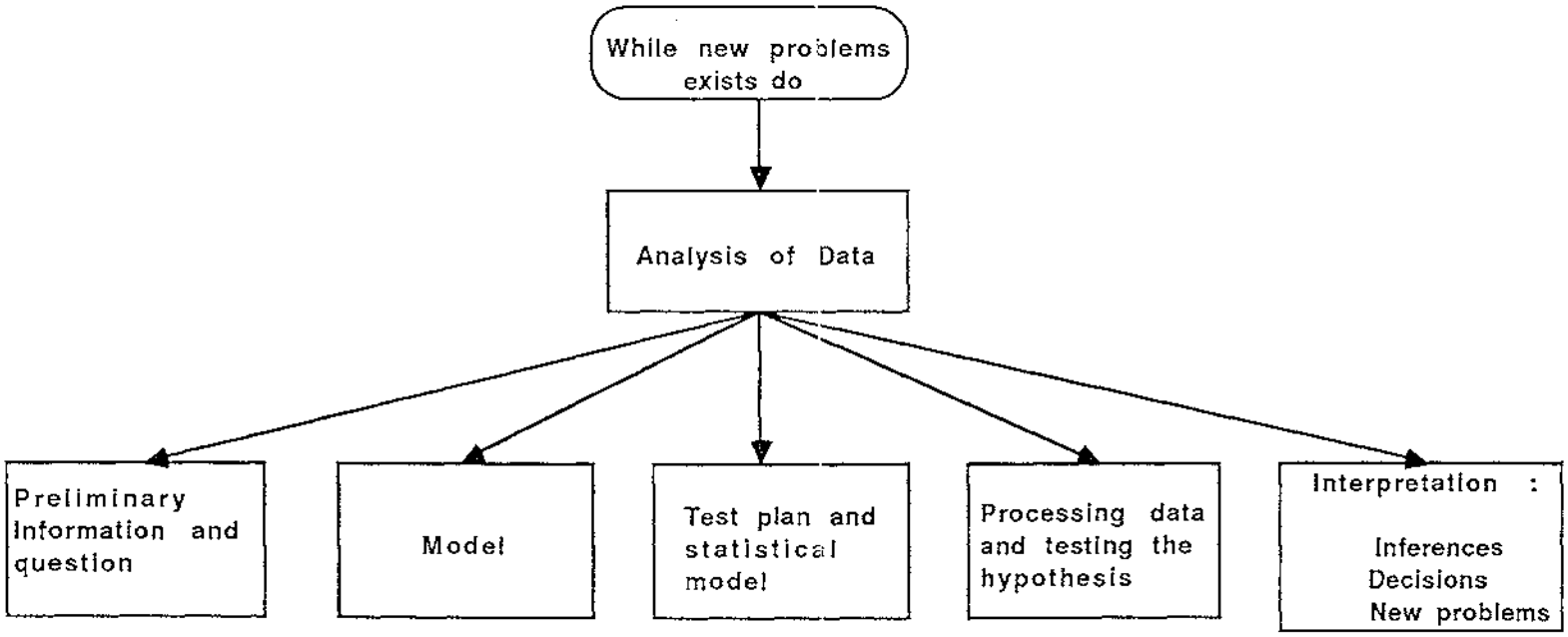


Fig. 4.4.1 Steps in the Statistical Analysis of Data

CHAPTER 5

EXPERIMENTAL RESULTS AND OBSERVATION

To Study the relationship of language expansion between the three languages, linear regression was used. Regression analysis establishes the relationship between two variables X and Y; where X is the independent variable and Y the dependent variable. In other words, X is the controlled variable while Y depended on X so that Y is considered a random variable.

The quantities for X and Y are obtained for n sets of data points (X_k,Y_k) where k ranges from 1 to n and n is the sample size. A scatterplot of those points is then drawn and a straight line is fitted through the data points. To establish the linearity of the relationships in the plot, the coefficient of determination (R^2) is used to test the strength of association between the two variables. The value of R^2 ranges from 0 to 1. This range forms the basis of the explanatory power of the bivariate regression model. The higher value of R^2 implies that the independent variable accounts for most of the variation in the dependent variable. That is, when R^2 = 1 or 100%, the independent variable (X) completely accounts for the variation in the dependent variable (Y); whereas for R^2 = 0 or 0%, the independent variable (X) does not account for any variation in the dependent variable (Y). Hence, in the latter case, X does not help in predicting Y. To study the analysis of variance for the bivariate model, the F-distribution test statistic (F-ratio) is used.

This chapter presents a detailed analysis of the experimental results obtained for this case study. It focuses on the relationship of size between languages and its relevance to the language expansion ratios. The results of system size and effort estimates, together

with a general evaluation of productivity measures for the relative size and effort for the three languages, are also presented for comparative studies of each language characteristic.

5.1 Regression Analysis of Empirical Data

The data used for the bivariate regression models are based on lines of code (LOC). These are based on the line counting rules specified in section 4.3 and are applied, as required, to all three languages; namely Advanced Revelation 4GL (AREV), Informix 4GL (Informix) and Micro-Focus Level II Cobol (Cobol). The abbreviation for each language name will be used herein for convenience and relate only to the above respective languages. Three cases are studied in this section for the three languages and they are listed as follows :

1. Case (a) - Informix versus AREV.
2. Case (b) - Cobol versus AREV.
3. Case (c) - Cobol versus Informix.

Each of the above will be referred to as case (a), case (b), and case (c) respectively in the following discussion for convenience.

Six sets of data were measured for each of the implemented languages and they are analysed for all three cases mentioned above. These are categorized as follows :

a. System - The system data comprises the collection of all implemented programs as listed in Fig. C.1 (see Appendix C).

b. Module Types - This comprises data for those definable functional categories contained within the application. These functional categories includes input/update

programs, reports programs, menu programs, file inquiry programs, and forms. The forms module type is considered separately although it is also part of input/update and reports programs. This is because this user-interface function is implemented differently between the three languages and their relative sizes may provide useful information in the analysis. Each module type in this category consists of the total line counts of all programs classified under that module type.

c. Input/update transactions - This category consists of either input or input and update programs.

d. Reports - This comprises programs which produce reports as the output.

e. Forms - This consists of data-entry screens used in the input/update programs.

f. External Inquiries - This follows the definition of the same category in Function Point Analysis and includes menu and file inquiry programs implemented in this case study.

Lines of code (LOC) are used as a basis for size measure of a program where the relationship of sizes between related functions in different languages can be studied. That is, the association of the size of one function in a language may be related to the size of another language by this common unit of measure. This section looks at the relationships of languages for the overall system as well as the individual component types. The sizes of 24 programs implemented for the case study are measured and tabulated according to the above six categories. These tables are presented in appendix C. Regression analyses are conducted for each set of data and the detailed results are presented in appendix D. Each case involves two steps. Regression analysis is carried out for the overall system and subsequent related studies are conducted for the module

types. The relationship for the overall system for each case can then be more clearly examined by relating the contribution of each module type to the overall system.

5.1.1 System

Two separate analyses are conducted for the system data. They are referred to as system 1 and system 2 (see appendix C). System 2 counts file inquiries as separate functions whereas system 1 does not. These two sets of data are used because of the observed outliers in the original dataset (ie. system 1). This will become apparent in the following discussion.

The general appearance of the plot for the overall system for all three cases for system 1 is shown in table 5.1.2. The R^2 value for cases (a) and (b) indicates that about two-thirds of the relationships can be explained by the size association between the two languages in each case. This is significant at a level of 0.001. Case (c), however, shows an even greater R^2 value of 87.7%. This implies that more than three-quarters of the variation between the two variables can be accounted for by the relationship in case (c). Case (c) has Cobol as the dependent variable and Informix as the independent variable.

When the actual plots are analysed, three distinct points appear to be outliers in the plots. The outliers are most apparent for case (a) and case (b) as shown in Fig. 5.1.1 and Fig. 5.1.2, where the three points are highlighted. The plot in Fig. 5.1.3 shows a more consistent pattern and does not relate to the above two cases. Analysis of the residual against the predicted value for all three regression models showed similar results. That is, a distinct pattern is reflected in the same two cases as shown in Fig. 5.1.4 and Fig. 5.1.5 where both cases do not show a random scatter. This implies that the regression model does not fully explain the relationship for case (a) and case (b).

TYPE	Y (LOC)	X (LOC)	R^2	F-ratio	Significance level
System	Informix	AREV	69.30%	56.40	0.001
	Cobol	AREV	71.70%	63.20	0.001
	Cobol	Informix	80.40%	103.00	0.001
Module	Informix	AREV	99.10%	439.00	0.001
	Cobol	AREV	86.30%	26.20	0.025
	Cobol	Informix	91.40%	31.90	0.025
Input/update	Informix	AREV	49.60%	8.84	0.025
	Cobol	AREV	55.60%	11.30	0.01
	Cobol	Informix	82.70%	43.10	0.001
Reports	Informix	AREV	82.10%	27.60	0.005
	Cobol	AREV	97.90%	285.00	0.001
	Cobol	Informix	82.40%	28.10	0.005
Forms	Informix	AREV	80.00%	24.00	0.005
	Cobol	AREV	61.00%	9.38	0.025
	Cobol	Informix	70.80%	14.50	0.01
Menus	Informix	AREV	91.80%	33.40	0.025
	Cobol	AREV	90.90%	29.90	0.025
	Cobol	Informix	72.40%	7.88	0.1

Table 5.1.1 Results of Statistical Analysis for System 2.

When examining the source of those three outlier points, it was discovered that all three are input/update programs where external inquiries are included in the line count. As the counting rules specify the inclusion of all code called by a module, these have been computed correctly. Examination of the codes shows that these inquiries do not contribute to the actual processing for the calling program and only serve to assist them. In view of this, it is a function of its own and should therefore not be included in the calling program. The separation of external inquiry functions are also considered when using Function Point Analysis. Therefore, it should also be consistently applied to the counting rules.

TYPE	Y (LOC)	X (LOC)	R^2	F-ratio	Significance level
System	Informix	AREV	58.00%	30.40	0.001
	Cobol	AREV	65.50%	41.70	0.001
	Cobol	Informix	87.70%	156.00	0.001
Input/update	Informix	AREV	13.80%	1.45	> 0.1
	Cobol	AREV	25.00%	3.00	> 0.1
	Cobol	Informix	85.80%	54.40	0.001

Table 5.1.2 Results of Statistical Analysis for System 1.

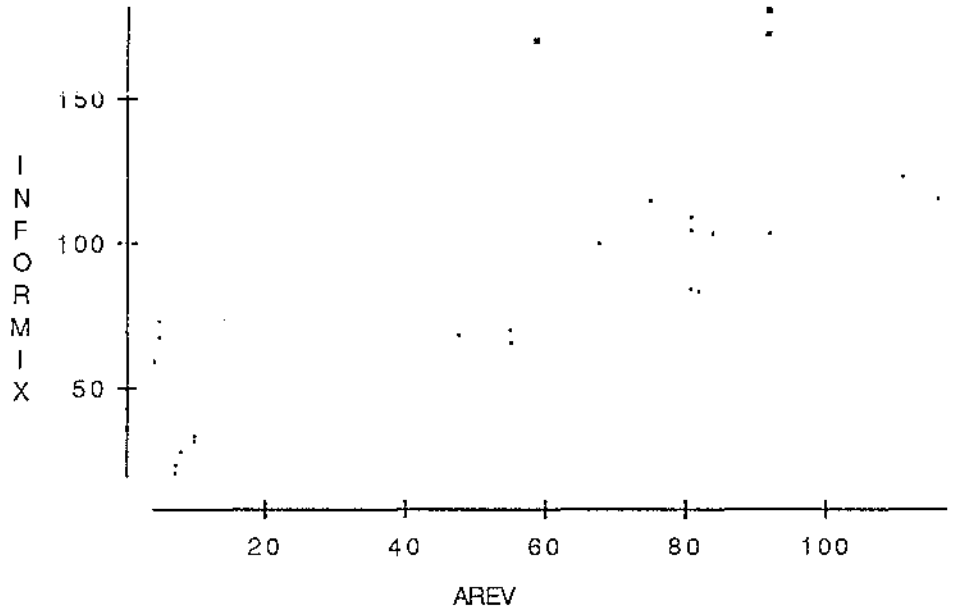


Fig. 5.1.1 Plot of Informix (LOC) vs AREV (LOC) - System 1

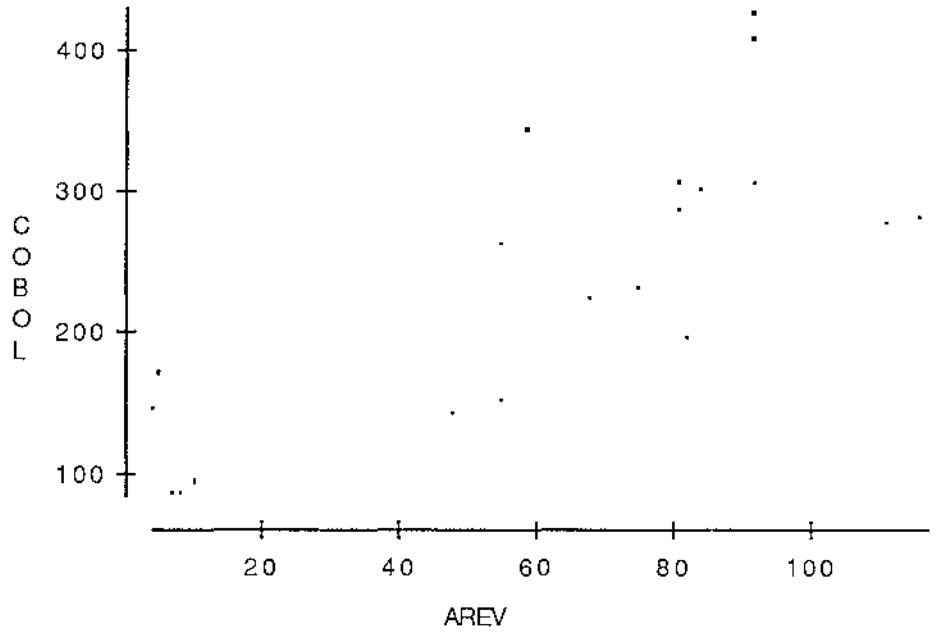


Fig. 5.1.2 Plot of Cobol (LOC) vs AREV (LOC) - System 1

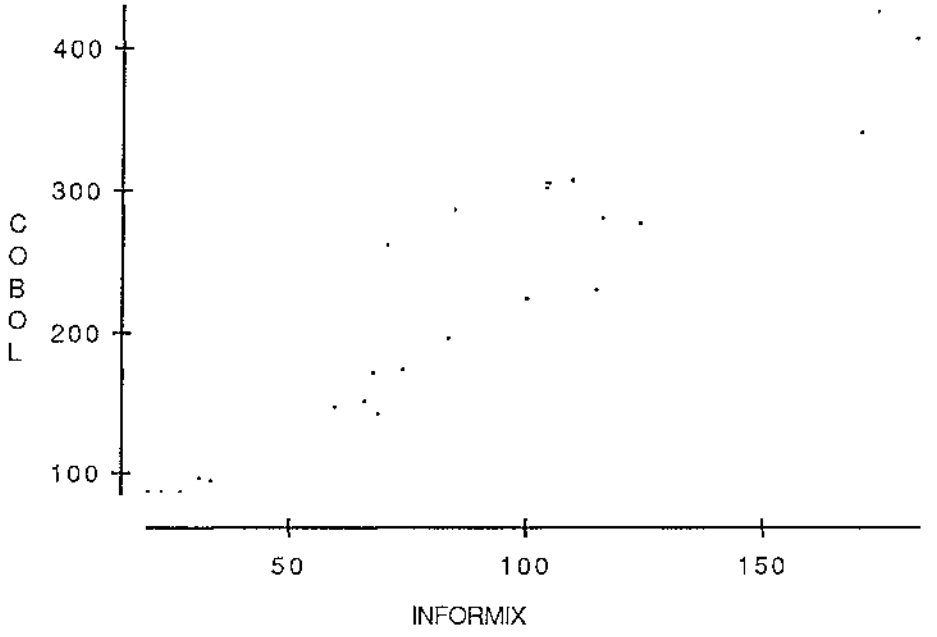


Fig. 5.1.3 Plot of Cobol (LOC) vs Informix (LOC) -System 1

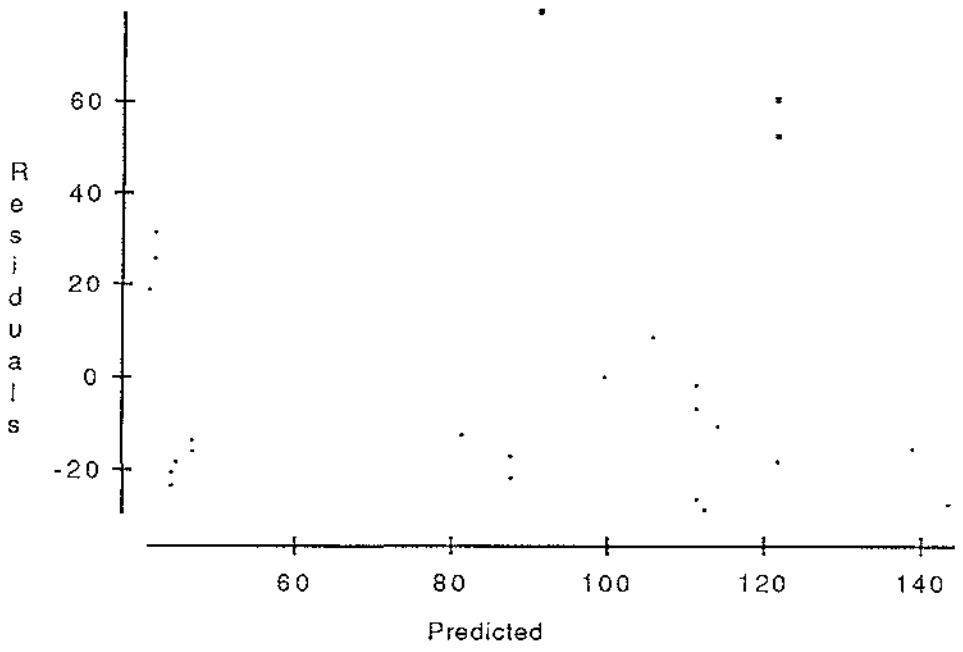


Fig. 5.1.4 Residuals plot for Informix vs AREV - System 1

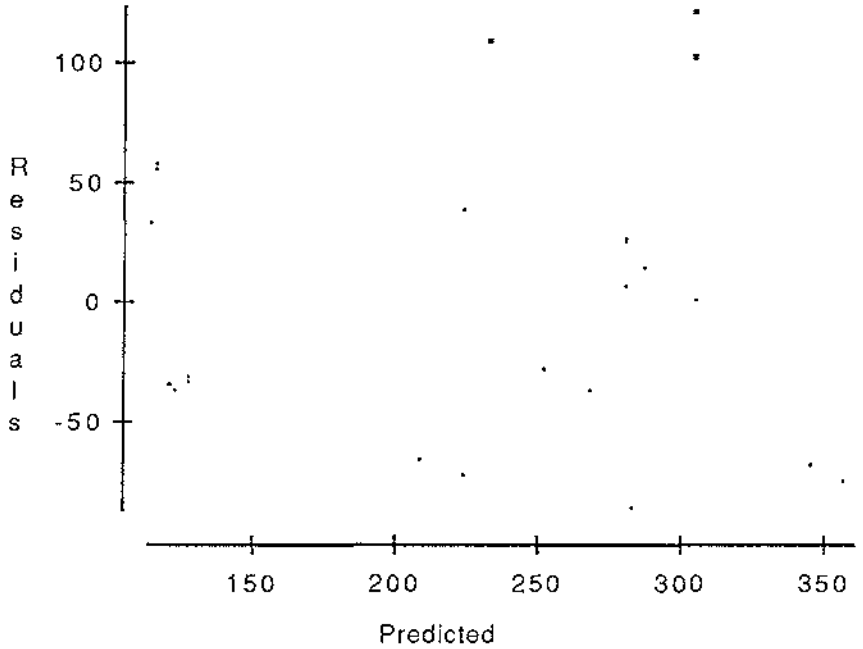


Fig. 5.1.5 Residuals plot for Cobol vs AREV - System 1

By excluding the external inquiry calls for the input/update programs three new plots are obtained. These are shown in Fig. 5.1.6, Fig. 5.1.7 and Fig. 5.1.8. The results

showed some improvement in the relationships for case (a) and case (b). However, the same does not appear in the third plot for case (c). The value for R^2 in the revised plot for case (c) decreases by 7%. However, this is not significantly different from the analysis of the initial dataset. The estimated regression equations for the three cases are shown below :

a. Case (a) - Informix versus AREV

$$\hat{Y} = \underset{(11.11)}{37.46} + \underset{(0.17)}{0.92}\,X \qquad \text{(original data set)}$$

$R^2 = 58.0\%$ $S_e = 30.51$

$$\hat{Y} = \underset{(5.53)}{45.33} + \underset{(0.09)}{0.67}\,X \qquad \text{(revised data set)}$$

$R^2 = 69.3\%$ $S_e = 17.40$

b. Case (b) - Cobol versus AREV

$$\hat{Y} = \underset{(22.46)}{105.16} + \underset{(0.34)}{2.17}\,X \qquad \text{(original data set)}$$

$R^2 = 65.5\%$ $S_e = 61.67$

$$\hat{Y} = \underset{(13.38)}{111.38} + \underset{(0.22)}{1.72}\,X \qquad \text{(revised data set)}$$

$R^2 = 71.7\%$ $S_e = 42.12$

c. Case (c) - Cobol versus Informix

$$\hat{Y} = \underset{(16.53)}{41.26} + \underset{(0.17)}{2.09\,X} \qquad \text{(original data set)}$$

$R^2 = 87.7\%$ $S_e = 36.87$

$$\hat{Y} = \underset{(18.74)}{18.99} + \underset{(0.22)}{2.26\,X} \qquad \text{(revised data set)}$$

$R^2 = 80.4\%$ $S_e = 35.03$

The reduced R^2 value for (a) and (b) above suggest the original model generally fits the data less well. The difference in the prediction caused by the outliers can be seen from the comparison of the standard errors (S_e) as well as the standard errors for the parameter estimates (in brackets). By changing the counting rule for the outliers, the results have improved. However, case (c) reflects the opposite result where the revised data set reduced the R^2 value by approximately 7%. One possible explanation for this is the similarity between Cobol and Informix in terms of their procedural code. However, the revised data set does not appear to deviate a great deal from the initial plot and the result is still significant at a level of 0.001. AREV programs, on the other hand, have much of their procedural code eliminated where less complex programs are concerned. This is most apparent in the inquiry function where it only requires a total of 27 LOC in AREV for all three inquiry programs whereas it requires 216 LOC for Informix and 419 LOC for Cobol. This marked difference in AREV's inquiries as compared to the other two seems to explain the departure of its relationships from norm.

The relationship for case (a), ie. Informix against AREV, has a constant value of 45.33 and a slope coefficient of 0.67 for the regression model on the revised data set. It shows that a system implemented in Informix has a certain overhead as compared to

AREV irrespective of its size. The slope coefficient of 0.67 further suggests that although Informix does have a relative overhead, the effect of this will be slowly

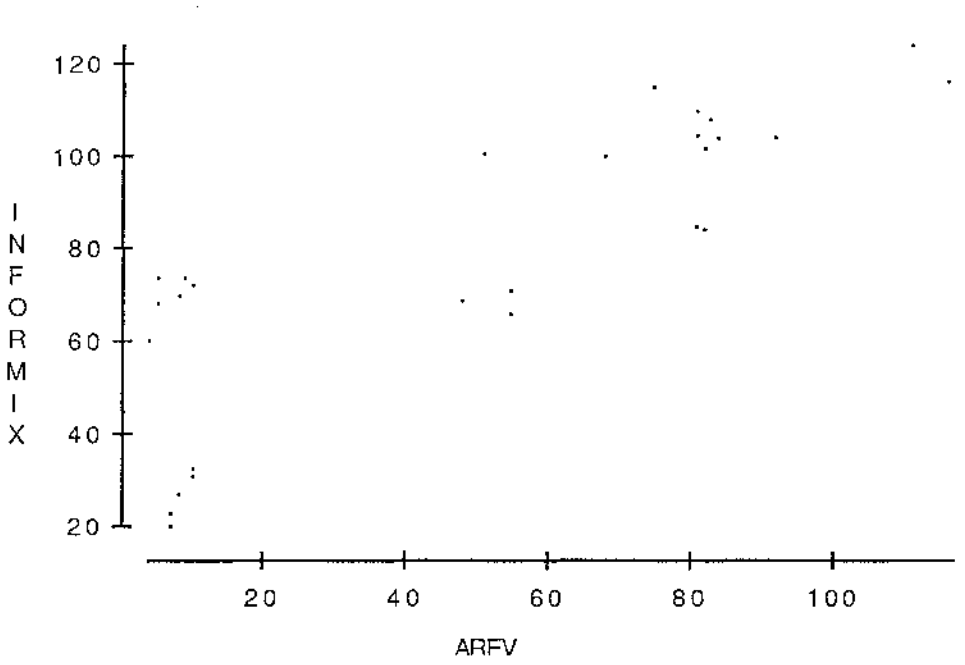


Fig. 5.1.6 Plot of Informix (LOC) vs AREV (LOC) - System 2

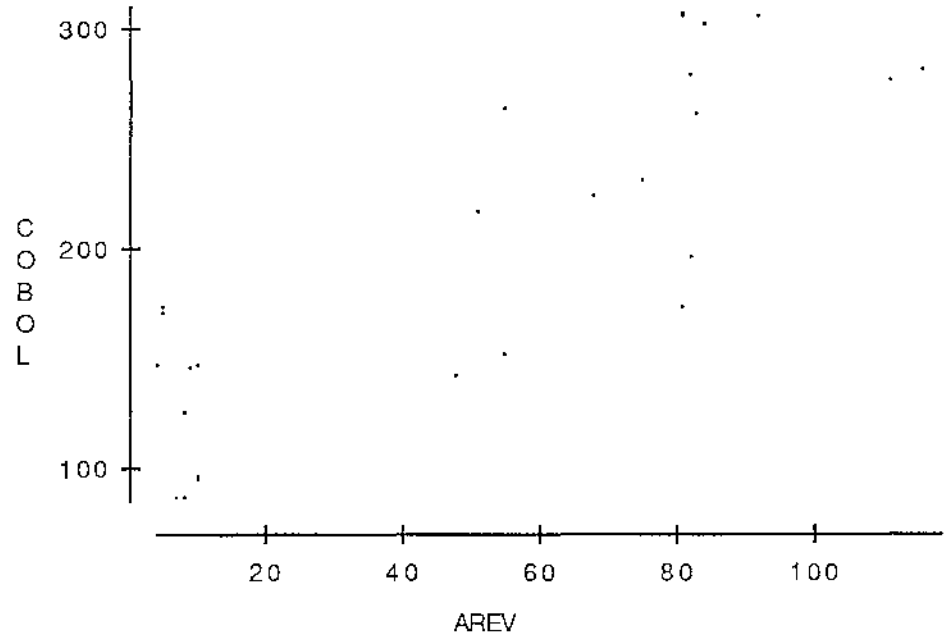


Fig. 5.1.7 Plot of Cobol (LOC) vs AREV (LOC) - System 2

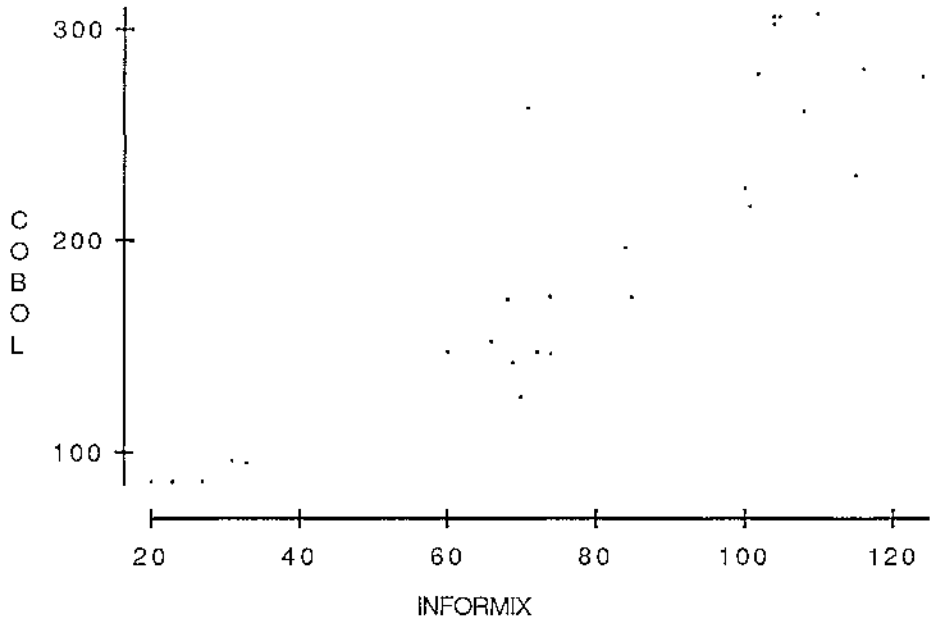


Fig. 5.1.8 Plot of Cobol (LOC) vs Informix (LOC) - System 2

reduced as the size of an AREV program increases. Therefore, in terms of comparative sizes, smaller systems with relatively less complex processing will see Informix requiring a larger size as it requires a relative overhead over AREV. However, for larger systems with moderate to more complex processing, this gap will be narrowed as can be observed from the regression equation. One reason for this is the need for procedural code to be included into the AREV implementation for more complex systems as the existing features do not cater for this type of complexity. The slope coefficient of 0.67 implies that for sufficiently large systems, Informix would in fact require less lines of code than AREV.

The regression equation for case (b), ie. Cobol against AREV, shows that Cobol also requires a relative overhead of approximately 111 line of extra code over AREV based on the measurements for this case study. The slope coefficient also suggests that the size of Cobol is about 1.7 times that of an equivalent size for an AREV implementation as system size gets larger, in addition to the overheads incurred. This suggest that AREV is more efficient than Cobol where size in lines of code is concerned.

In analysing case (c) of the relationship between Cobol and Informix, a more significant size difference can be observed as systems get larger and more complex. The size of Cobol is shown to be 2.26 times that of Informix in incurring a relative overhead of approximately 19 lines of code. At the outset, this indicates that the initial difference in overhead for the implementation in the two languages is rather small but, as the size of the system increases, the size of the system implemented in Cobol will grow proportionately at 2.26 times that of an equivalent Informix implementation. This makes Informix the more favourable language to use as compared to Cobol.

This case study indicates that the size relationships of the languages in the regression equations account for approximately 70% of variations between the languages. The level of a language is seen to relate to the relative size in lines of code. That is, a higher level language tends to require fewer lines of code than a lower level language. In this case, Informix and AREV are the higher level languages which require about half the lines of code needed for an equivalent Cobol implementation.

The above three cases provide a basis for the comparison of each of the languages with the others. AREV is seen to be more efficient, where lines of codes are concerned, as compared to Informix for small and less complex system components but not so for the larger and more complex ones. For larger system components, Informix seems to be a better choice. In the case of Cobol and AREV, AREV requires a smaller size as compared to Cobol for all systems, whether they be small or large. In this case, AREV is superior to Cobol in terms of the required lines of code. Similarly, Cobol requires more lines of code than Informix whatever the size or complexity of the implemented system. Hence, Informix is more appropriate if a smaller size implementation is deemed to be more efficient. Table 5.1.1 and 5.1.2 summarizes the results of the regression analysis for the overall system and the module types.

Based on the above analysis, it is relevant to consider the importance of using language size expansion in predicting software sizes. The above study shows that all three cases are statistically significant at the 0.001 level. This implies a strong relation between the languages based on their function size in lines of code (LOC). The following sections study in greater detail the relationships of the module types between the three languages in relation to the trends observed with the overall system as described above. The detailed results of these analysis are illustrated in Appendix D. The parameter estimates for all cases were found to be consistent with the above analysis. A one-tailed t-test shows that the slope parameter estimates were found to be significant at the 0.05 level.

5.1.2 Module Types

The importance of the function categories in language expansion can be investigated by a regression analysis of the plot of relative function sizes between two languages. Table 5.1.1 shows the statistical significance of the results for the three cases. All three cases show a high coefficient of determination R^2 of greater than 91% with the highest for case (a) recording a value of 99.1%. The significance level for the relationship for case (a) is 0.001 and the others shows a level of 0.025. This shows that the function categories are highly significant in accounting for the variation of size in one language with the other. It also supports the earlier observation.

5.1.3 Input/Update Transactions

The effects of input and update modules posed some problems in the line count due to the effects of including external inquiry code in the calling program. An examination of this module type confirmed the earlier observation that external inquiry code should be counted as a separate function. A regression analysis of the input and update modules for the three cases showed similar results observed in section 5.1.1. This is tabulated

in table 5.1.1 and 5.1.2. Table 5.1.2 shows that case (a) and (b) are both not significant at a 0.10 level with the R^2 for case (a) being 13.8% and that of case (b) being 25.0%. This suggests that AREV does not associate very well with either Informix or Cobol. An interesting observation is that of case (c), ie. Cobol and Informix. Again, this is consistent with the observation in section 5.1.1.

The analysis for case (c) shows a 3% difference for the R^2 value for the initial dataset and the revised dataset. This is not a significant difference. As observed from the analysis of case (a) and (b) for the revised dataset for input/update modules, the changes have improved the results for these two cases. Therefore, the revised counting rule is an appropriate one.

As both Informix and Cobol require an underlying framework for the development of an application, the inclusion of the inquiry code for both these cases was observed to be consistent with each other. AREV, on the other hand, provides an underlying language utility which handles the presentation aspect in terms of layout and selective data display. This grossly reduced the code required and is one important difference between the languages. As there is no distinction in the initial counting rule with regard to specific functions involved in the call statement, the computation of the line count has not been misrepresented. The results which ensue indicate the need to distinguish between functions involved with the processing in the calling program which assist the program and functions such as inquiries or extra utilities in screen formatting for example. As section 5.1.2 shows, the distinction of function categories improves the relationships between two languages. This in fact does not disadvantage the use of Function Point Analysis as it distinguished functions more precisely which could, perhaps, form a better base for the initial computation of function points.

The prediction of Y for the revised data set shows the R^2 value to be 49.6% and 55.6% for case (a) and case (b) respectively. These two cases do not show very significant

explained correlation between the two variables. As discussed earlier, the inherent features provided by AREV significantly deviate from the conventional programming concept with the application of non-procedurality. Hence, size is sensitive to the type of processing contained within this module type. That is, where an input/update program implemented in AREV does not require any procedural code for its processing, the relative LOC can be reduced quite dramatically. But, when procedural codes are required for more complex processing, the LOC difference between Informix and AREV seems to be reduced. Examples of the code difference for this case is illustrated in table 5.1.3. Notice the almost identical count in terms of lines of code. Although this is only two example programs, it serves to explain the observation.

Case (c) shows a higher correlation with an R^2 value of 82.7%. This is consistent with the basic procedural code required for both Cobol and Informix. The savings in Informix is in the non-procedural codes, such as forms generated using Form4GL and the SQL implementation. The association between Cobol and Informix is significant at a level of 0.001 (see table 5.1.1).

Program-ID	AREV	Informix
MSL4	116	116
MSL6	111	124

Table 5.1.3 Input/Update programs with file-to-file operations.

The three cases examined for input/update module type shows that about 50% can be explained in terms of the variation in size between the three languages. When less complex processing is involved with little procedural code, AREV shows some relative savings. This effect evens out when more complicated processing is required. The available features in AREV, such as form-filling for example, do contribute to reduced LOC count. This is the fundamental difference as compared to Informix. AREV's

advantage over Cobol includes the query languages as well, such as the LIST and SELECT statements for example.

Cobol and Informix show a strong relationship as the high R^2 value shows. More than 80% of code expansion from Informix to Cobol (ie. case (c)) can be accounted for by the relationship. Examination of the codes for each language shows the similarity of procedural code in both of them. The higher level functions in Informix are the more powerful statements, the forms utility, and the query language implementation. This has considerably reduced the lines of code required as compared to Cobol.

5.1.4 Reports

Regression analysis on the reports module type shows very high correlation for all three cases. The R^2 value is greater than 80% for all three cases with the significance level for case (a) and (b) being 0.005 and that for case (c) being 0.001. This is consistent with the observation in section 5.1.1.

For case (a), the regression equation is given by :

$$\hat{Y} = 63.17 + 0.47\ X$$

This equation indicates a relative overhead for Informix over AREV. The slope coefficient of 0.47 suggests that the size difference would diminish as system size increases. The size relationship between AREV and Informix has an R^2 value of 82.1%. This means that the equation accounts for over three-quarter of the variation in the dependent variable. For AREV, reports use query language statements like LIST or SELECT to generate the required data and format for the report. A default report format is available but a user-specified format can also be set up. This similarly applies for Informix, although the code is structured in a different manner. The procedural code

for Informix forms the relative overhead required as compared to AREV. Hence, the reporting concept provided by these two languages forms the basis for the strong size association between them. Cobol, on the other hand, is predominantly procedural in its code implementation. However, the high R^2 value for case (b) and case (c) indicates that the size of a Cobol report is also associated with an equivalent function size in either Informix or AREV. This is consistent with the earlier observation.

The high R^2 value for all three cases for the reports module type shows that the reports implementation between the languages have a similar pattern. The different concepts in record selection and presentation provide the basis for the reduced size. The regression analysis for all three cases shows size relationships for reports between languages to be statistically significant at a level of 0.005. This shows that the sizes between languages for the reports module type are associated and that one language size can be used to predict the other with 99.5% confidence as far as reports are concerned.

5.1.5 Forms

Forms are front-end data entry screens provided as user-interfaces for an application. Table 5.1.1 shows a high coefficient of determination R^2 for all three cases for the Forms module type. The size of Forms in one language is significantly influenced by the size of an equivalent function in another language. Case (a) accounts for 80% of the variation in Informix by the changing size of AREV. Case (b) accounts for 61% of the variation in Cobol by the independent variable AREV. Case (c) accounts for 70% of the variation in Cobol by Informix. The results shows a distinct correlation of the size for Forms module between languages in this case study where more than two-thirds of the variation can be explained by the relationship.

The windows provided by AREV have a straightforward screen painting process and the window itself can be sized down to requirement and located at a specific position using

the standard environment utilities provided. This does not require additional code. Prompts, or data entry fields, are created where the prompt names can be hidden from view. This reduces the lines to the number as seen by the user. An underlying data dictionary is used for data definition which can be used in subsequent screen design.

Informix also has an improved screen design utility which is quite similar to AREV but applied in a different way. The design is created by a direct screen painting process in a forms module with the data field elements defined. Data field elements are defined to relate to database files.

Cobol's Forms-2 design also provides a screen painting utility but the code is not hidden as is the case for Informix and AREV. The forms designed in turn generated code for the data division in the Cobol program. The size of this code is then dependent on the complexity of the forms designed as well as the number of input data elements. Because of this generated code, the size of forms for Cobol tends to be much larger than the two 4GLs, though the effort to produce them is comparable as will be shown later.

In examining the code for the three implemented versions of forms design, AREV shows the least required lines which includes only text description and prompt entries. Informix needs in addition to relate the input data elements to database files. Therefore, it requires those extra lines over AREV. Cobol requires substantially more lines in the data division and therefore more lines of code as compared to either Informix or AREV. The regression analysis results in table 5.1.1 show that although the forms are implemented in slightly different ways, their sizes are closely related. The level of the languages is observed to relate to the relative size of forms in terms of lines of code.

5.1.6 External Inquiries

External inquiries for this case study includes menus and file inquiries. As observed in section 5.1.1, AREV's file inquiries (popups) showed substantial reduction in terms of lines of code as compared to Informix and Cobol. This is because there is a fixed display format for the popups in AREV. As such there is no need to construct a framework for such a display structure. Informix and Cobol requires procedural code for the construction of this display for file inquiries. This explains the extra code needed for a similar implementation in both Informix and Cobol. The reduced size for Informix as compared to Cobol is because of the higher level statements provided by Informix together with its windowing features.

A similar display to popups applies for menus in AREV where the size is dependent on the number of menu options. Informix has the menu statement to deal with the display of a standard menu on the first two lines of the screen. The menu facilities are quite comparable for both Informix and AREV. Cobol, on the other hand, requires the data division for screen description of menus and this forms the extra code needed. Also, extra procedural code is required for selective display of menu options for an equivalent Cobol implementation.

The underlying structure for Informix and AREV used to handle screen display and selection of options for external inquiries provides relative savings in terms of lines of code. Cobol, being a conventional third generation language is lacking in this area and results in extra procedural code. Table 5.1.1 shows a high coefficient of determination R^2 for all three cases studied for the menu module type. The relationship between the sizes for the three different languages is significant at a level of 0.10. These results support the observation on this aspect of code implementation.

5.1.7 Comparative Expansion Ratios

The regression analysis conducted indicates that more than two-thirds of the variation in the function size of one language can be related to a similar function size of another language. Section 5.1.2 also shows that the function categories for the case under consideration is statistically significant between languages. The study of language expansion ratios is to focus on its relevance in expressing the characteristic size of the identified functions based on the ability of obtaining such functional categories for an application. The above statistical analysis shows that it is possible to relate similar functions based on their relative sizes in lines of code between different languages and within a specified application domain.

Instead of using functions points or lines of code as the basic unit of measure, neither of which proves to be very effective when applied on its own, it is appropriate and useful to apply language expansion ratios to bridge the two measures. This section analyses the expansion ratios calculated for the three languages in this case study as well as the usefulness of using function points to capture the relative size of functions.

Language Expansion ratios are calculated based on the lines of code (LOC) measurements as well as the function points (FP). The LOC count follows the rules as specified in section 4.4 on software measurement. One change applied during the course of the experiment is the separation of external inquiry subroutines from the calling program. FPs are determined by using a language-independent size estimation method. Language expansion ratios are then computed using LOC as the dividend and FP as the divisor. This value follows the definition outlined in section 2.4. An Expansion ratio (ER) is calculated for the overall system as well as the module types for each language implemented. The module type category follows the grouping of different functions in FPA.

Using the Function Points as the language independent size measure, the ***System Expansion Ratio (SER)*** can be defined as :

$$SER = \frac{\frac{S_O}{A_O W_O F_O} + \sum_{i=1}^{n} MER_i}{n + 1}$$

where
- S_O = Size of relative overhead in lines of code
- A_O = Adjustment factor for overhead
- W_O = Relative weight for complexity associated with F_O
- F_O = Unadjusted function points
- n = Total number of module types
- MER = Module type expansion ratio.

The ***Module Type Expansion Ratio (MER)*** relates to the language expansion from the language-independent function points for individual module types for the system. This ratio can be defined as :

$$MER_i = \frac{S_i}{A_i W_i F_i}$$

where
- i = Module type
- S_i = Size measure in lines of code
- A_i = Adjustment factor
- W_i = Relative weights for F_i
- F_i = Unadjusted function points.

The above formulas are used in the following computation of the expansion ratios using both Albrecht's ***Function Point Analysis*** and Symons' ***Mark II method***.

The results for the language expansion ratios obtained using Albrecht's Function Points as the common denominator are tabulated in table 5.1.4 below. The system expansion ratio is based on the total lines of code of the overall application including any utility programs which do not fit into any of the specified module types. The module types specified are peculiar to a typical data-centered business application class of which

this case study is an example. It is appropriate for this study as it allows a closer examination of language expansion for the module types and the identification of any differences between them.

The data in table 5.1.4 shows a prevailing trend for expansion ratios within each of the specified language. All three languages show the highest ratio for the input/update module type followed by reports and then external inquiry. Of all the expansion ratios for the module types, only the reports module type for Informix and AREV are consistent with the system expansion ratio. The Input/update module type for all three languages has an expansion ratio which is twice the value of the system expansion ratio. Significant differences are also shown for the external inquiry module type for AREV, as compared to the ratio for the system, but lesser for Informix and Cobol.

The difference observed between expansion ratios for each module type as compared to the system expansion ratio suggests the inadequacies in the weightings associated with each module type when computing their relative function points. In order for each module type to reflect a representative expansion ratio which should be consistent with the system expansion ratio, the function points allocated to each module type must be indicative of the function value of each module type. For example, given the function points for the input/update module type to be 28.05, then the equivalent lines of code for this module type in AREV is calculated to be 393 when the system expansion ratio is used. However, if the expansion ratio for that module type is used instead, then the equivalent lines of code would be 898. This is more than double the value calculated using the system expansion ratio. The value of 898 lines of code is supposed to be the actual size but is not adequately captured in the function points allocated to this module type. Moreover, when the system comprises of a different combination of module types, the use of the system expansion ratio will not be accurate. That is, for a system with more input/update modules, the existing system expansion ratio will tend to under-estimate the required size.

Language	Expansion Ratios (LOC/FP)			
	System	Input/update	Reports	External Inquiry
AREV	14	32	15	4
Informix	24	46	26	20
Level II Cobol	56	106	73	49

Table 5.1.4 Summary of Language Expansion Ratios based on Albrecht's Function Points.

As an alternative to Albrecht's Function Point Analysis, Symons proposed the Mark II approach. In order to compare the expansion ratios determined above, function points derived using the Mark II approach is used. The Mark II function points uses the number of input and output data elements and the entities referenced in the application system to derive the required unadjusted function points. A weighting factor is then applied to each category. A technical complexity factor is then used as the adjustment factor. As the technical complexity factor for the Mark II method is not available, the adjustment factor of 0.85 determined in Albrecht's FPA is used instead. The five additional factors introduced by Symons are not relevant to this case study and as such the use of Albrecht's adjustment is deemed to be sufficient for the required adjustments. Table 5.1.5 shows the resultant expansion ratios derived using Mark II function points. The weighting for each of the specified categories, namely input and output data elements and entities referenced, are based on those calibrated by Symons [SYMO88].

As can be seen from table 5.1.5, the expansion ratios using Mark II method have shown some improvements. Expansion ratios by module types does not show very wide differences as compared to the system expansion ratio. In relative terms, AREV's external inquiry expansion ratio have a larger difference than the others. This

difference can be due to the fact that AREV provided an underlying utility for constructing this module type and , as noted earlier, this has greatly reduced its size in terms of lines of code. The reports module type for all three languages also showed lower expansion ratios as compared to their respective system expansion ratio. This may be due to the types of reports generated for the implemented system which may not be representative to a typical report as seen in larger systems where substantially more line types may be used. The overall results showed the expansion ratios to be more representative of the implemented system as well as their different module types.

Language	Expansion Ratios (LOC/FP)			
	System	Input/update	Reports	External Inquiry
AREV	15	18	10	5
Informix	26	27	18	24
Level II Cobol	61	62	49	60

Table 5.1.5 Summary of Language Expansion Ratio based on Symond's Mark II Approach

A comparison of the expansion ratios obtained using Albrecht's and Symond's function points suggest the importance of the weights applied to each module type so as to reflect a representative function value. If an expansion ratio is to be useful as a conversion factor between function points and lines of code, then it is necessary for function points itself to reflect a more consistent size equivalence. Albrecht's function points seems to be less effective in capturing this function size equivalence than the Mark II method. The fact that both methods show similar system expansion ratios is an indication of the basic consistency and relevance of the function points approach. The tabulated results above implies a need for a further fine-tuning of the weighting factor associated with the degree of difficulty of implementing the different module types so that the calculated function points will be more representative of their equivalent size.

In summary, the function points obtained using Albrecht's function point method does not adequately represent the size of different module types. As such, the system expansion ratio tends to depend on the proportion of the different module types. Mark II method seems to provide a better expansion ratio for the system as well as the module types. Irrespective of the function point methods used, the system expansion ratios for all three languages are comparable between the two function point methods. This suggests the relevance of function points as a measure of program size. Because module type function points determined using Albrecht's method does not provide consistent expansion ratios, the weightings applied become questionable. By using a different counting method and weighting factors, Mark II method showed some signs of improvements. This means that a more accurate measure of function points can be calibrated so that the expansion ratios can then be applied more confidently for the wider application, including software size prediction.

5.2 System size and Effort - estimates vs actuals

The size and effort estimates were carried out according to the procedures outlined in section 4.1. These estimates are tabulated in table 5.2.1 below. The detailed design is used as a basis for coding for all three languages. The actual implementation effort consolidates the two phases of code implementation as determined by COCOMO, ie. the code and unit test, and the integration and test phases. Because incremental development was used, there is no clear distinction between these two phases. The figures in table 5.2.1 reflect the effort required to develop a software program as opposed to a software product. A software product normally includes such activities as thorough documentation, testing, error-proofing, user-tailoring etc, which were not thoroughly applied for this case study. As indicated in [BOEH81], a software product requires about three times as much effort to complete as an equivalent-sized software

program. Therefore, the tabulated results should be interpreted with this in mind when comparing the estimates using COCOMO and the actual results.

Because of resource constraints, machine time for development tends to be restricted. Efforts to record the time spent for development have been made difficult because of this constraint. The results tabulated in table 5.2.1 reflects the actual implementation effort without a real consideration for the miscellaneous tasks such as those mentioned above. However, the actual efforts have been recorded with sufficient care to reflect the actual code implementation effort in person-month. This should be appropriate for comparative studies.

Phase	AREV			Informix		Cobol	
	estimate (i)	estimate (ii)	actual	estimate	actual	estimate	actual
Lines of Code	2423	1628	1328	4845	2250	10175	5406
Detailed design	2.2	1.5	2	4.5	2	8.7	2
Code and unit test	2.8	1.9	1.4	5.8	1.5	10.2	0.8
Integration and test	1.4	0.9		3.2		6.3	
Total effort	6.4	4.3	3.9	13.5	3.5	25.2	2.8

Table 5.2.1 Estimated and Actual Effort in person-months.

As indicated in table 5.2.1, the COCOMO estimate for all three language implementation substantially over-estimated the actuals. The initial size estimates do not provide an accurate representation of the actual size and this results in the over-estimation in the effort prediction. The language expansion ratios used do not correspond to the expansion ratios determined in the last section. As no definite guide-lines on the use of expansion ratios for 4GLs have yet been established, the above differences were anticipated.

In the case of AREV, the expansion ratio used was 25 LOC/FP as compared to 14 LOC/FP in table 5.1.4. This has led to the over-estimation of size and, hence, effort. The same situation applied to both Informix and Cobol. One reason for this is the subjectivity of the expansion ratios itself. Because expansion ratios have not been specifically calibrated for each of these languages, the expansion ratios used are a conservative approximation of the actuals. Taking the second estimate for AREV (ie. using a contraction factor of 0.16 obtained from [VERN88a]), which shows a close approximation to the actual result, the estimated effort for detail design is 1.5 person-month as compared to the actual effort of 2.0 person-month. Looking at the effort for code implementation, the COCOMO estimate gives 2.8 person-months (ie. adding the figures for code and unit test, and integration and test) whereas the actual result is 1.4 person-month. Considering that COCOMO tend to estimate a software product as opposed to a software program and following the definition above of software programs, a multiplying factor of 2 is appropriate to elevate the actual result to be comparable with the COCOMO estimate. Using this logic, the actual effort in person-month for code implementation in AREV is found to be 2.8 person-month. This correspond well with the estimate, which incidentally is also 2.8 person-month. The other estimates for Informix and Cobol are quite different because of the initial size estimates. Therefore, it is not appropriate to make any sort of comparison. However, based on the estimates obtained for AREV, COCOMO's estimates seems to correspond quite well with the actual result after applying the adjustments.

The actual code implementation for the 4GLs, namely AREV and Informix, was carried out with an initial implementation of a prototype. This is because it was felt inappropriate to begin development of the actual system when familiarity with those languages was relatively low. The figures reflected in table 5.2.1 are based on the overall effort, including the effort in building the prototype. The effort expended in implementing the actual system is shown in table 5.3.2.

5.3 Comparison of Productivity Measures

Table 5.3.1 summarizes the productivity measures for the three languages. The tabulated results shows the relative size and effort for each language. These results forms the basis for the comparison between the three languages.

Comparing the relative system size for the three languages, AREV have considerably less lines of code than either Informix or Cobol. This has been observed as being due to the non-procedural concept of AREV. Informix, on the other hand, requires about less than half the size for an equivalent Cobol system. The reason for this is because Informix has a higher level language characteristic and some non-procedural concepts. Its implementation of code is more procedural than AREV. The need for procedural code explains the larger size required for Informix as compared to AREV. Therefore, in terms of size in lines of code, AREV has the smallest size followed by Informix, both of which are smaller than Cobol due to their non-procedural concepts.

LANGUAGE	(a) Size (LOC)	Effort in person-hour (b) Prototype	(c) Actual	(d) Total effort
AREV	1328	160	61	221
Informix	2250	110	130	240
Cobol	5406	-	116	116

note: 1. (d) = (b) + (c)

Table 5.3.1 Summary of development efforts in person-hours.

The implementation for Informix and AREV is slightly different to that for Cobol in that a prototype was built for the two 4GLs. As mentioned earlier, prototypes are implemented for a subset of the functions for the system using the two 4GLs in order to

gain familiarity with their language features as it is felt to be necessary even with incremental development. The actual system was then implemented so that a fairer view of the actual implementation effort can be obtained without the bias of the initial learning curve. Furthermore, it allows the removal of superfluous exploratory details which will tend to affect the overall line count. Table 5.3.2 summarizes the milestone time-log for the three implemented system in person-hours. These results reveals the relative efforts required for each increment. It can be seen from those results that AREV required substantially high effort in the prototype phase as compared to Informix for reasons which will be explained later in this section. Subsequent implementation from the first to fourth increment shows AREV consuming the least effort as compared to Informix or Cobol. In terms of effort, Informix is quite comparable to Cobol in some areas. Other areas seems to indicate more effort in implementing an equivalent Informix program. These are mainly in the testing of individual modules and the effects of module coupling and will be explained in greater detail below.

When comparing the total effort between the languages (column (d) in Fig. 5.3.1), Cobol shows the lowest effort required. This is because the Cobol implementation does not include a prototype. The total effort for Informix and AREV is substantially higher than Cobol because the prototype also forms part of the learning curve for the two 4GLs. This can be seen to suggest the learning curve required for the 4GLs. If the prototypes are excluded from the total efforts figure, ie. column (c) in Fig. 5.3.1, AREV is seen to require the least effort as compared to both Informix and Cobol. This is because the non-procedural implementation in AREV becomes relatively simple once the barrier of the learning curve has been overcome. This is more so for a less complex system. For more complex task, such as the third increment in Fig. 5.3.2, the amount of effort is quite comparable to Informix or Cobol. The difficulty with the structured approach which those form-filling features demand often requires substantial time in evaluating the required inputs. However, once sufficient familiarity with the language have been attained, the required inputs become quite

mechanical and, hence, the savings in terms of effort result. When more complex implementations are involved in which procedural code is required, the effort required will increase. Because AREV provides some utilities for constructing some of the query language statements, reasonable savings is observed in this area.

Milestone	Effort (person-hours)			
	Task Category	AREV	Informix	Cobol
Prototype		160	110	-
	Sub-total 1	160	110	0
1st increment	Set-up database	-	3	-
	Menus	4	5	8
	Item-type	4	25	16
	Supplier	6	22	14
	Date	4	5	4
2nd increment	Supplier - Item-type	4	15	14
	Item	4	15	14
3rd increment	Stock-on-hand	27	21	18
4th increment	Stock withdrawal	4	12	16
	Stock replenishment	4	10	12
	Sub-total 2	61	130	116
	Grand Total	221	240	116

Table 5.3.2 Milestone Time-log in person-hours.

Informix and Cobol have quite similar implementation effort when a prototype is not included for Informix. The difference in effort between these two language does not only lie in the language characteristics itself, where the features in Informix make it more productive in implementing similar functions in lesser lines of code. The main time consuming task in using Informix is the way in which modules need to be linked

before it can be run. The implementation in several modules meant that this linking phase is necessary before testing on the semantics of modules can be carried out. As inter-module coupling is unavoidable for the system implemented in this case study, the entire system or relevant subset of the system will have to be integrated, via compile and link option, before the processing logic can be tested. This forms a relatively large proportion of the total effort. Another area of difficulty is in the database set-up. When modifications to an existing database is required, the entire database will have to be re-constructed for the changes to be effected in the current database. This means that the effectiveness of testing in incremental development is dependent on the certainty of the structure of the database. Because test data are often used by subsequent increments, a change in the database implies that these existing test data will be lost. Hence, the inability of Informix to allow for modifications of only certain relations in the database results in the greater effort required in order to cope with such a change.

Micro-Focus level II Cobol provides an easy to use screen painting utility which simplified the screen design process. This becomes quite comparable with the two 4GLs in the area of forms design. The Animator also provides substantial savings where the program can be executed by individual statement or a group of statements using breakpoints. Variables can be inquired into to determine the accuracy of the processing. This is rather effective in reducing the efforts required in the testing of individual modules. The major time consuming task involves the lower level procedural code as compared to either Informix or AREV.

CHAPTER 6

SUMMARY AND CONCLUSIONS

Current issues of software measurement and sizing, presented in chapter 2, reveal the need for further research in this area so that more objective measures for software can be achieved. The perpetual change in form and style of programming languages highlights the need to standardize the basic unit of measure for quantifying software. This measure is becoming increasingly important in productivity studies as well as with related models of the software development process, especially the cost/effort prediction models. A method such as FPA with a language-independent size metric was seen to have the benefit of being used early in the software development process irrespective of the implementation language. However, its use has been limited by its lack of "universality" as the basic unit of measure. Lines of code (LOC), with its existing base of information, was found to be useful in extending the applicability of function points thereby allowing it to be applied with a wider range of programming languages and complementing other models of the software development process. The significance of using language expansion ratios as an extension to function points was an attempt to bridge the gap between function points and lines of code. Expansion ratios serve as a language-dependent size multiplier which produces an equivalent lines of code measure from function points. Such an effort would enable the continued use of LOC in software sizing with the convenience of using function points as the initial size measure.

An important objective of this research was to investigate the validity of *Language Expansion Ratios* and examine the factors affecting their accuracy when used in software sizing. Experience in software development points in the direction of the need

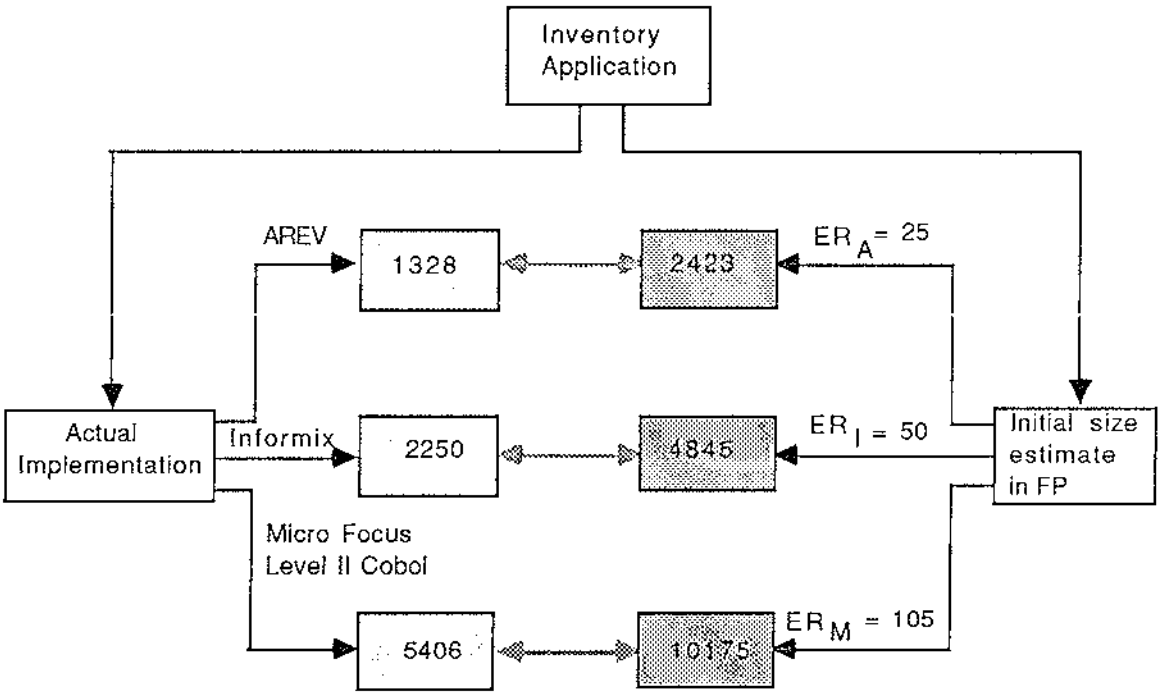
for more accurate effort estimates and the use of appropriate sizing methods to adequately capture the size for subsequent input to effort and scheduling models. The focus of this study was on the relevance of using expansion ratios in expressing the characteristic size of an application and their usefulness in software size prediction.

The case study, outlined in chapter 3, was used as the basis for the detailed design and it was implemented using the three selected languages; namely Advanced Revelation (AREV), Informix 4GL (Informix), and Micro Focus Level II Cobol (Cobol). This specific data-centred business application, together with the selected languages, was used for the comparative study because of the advantage it has in restricting the range of variability to the domain of interest.

The methodology adopted for the empirical study was presented in chapter 4 where emphasis was placed in ensuring a consistent implementation of the inventory application for each of the three languages with appropriate considerations in empirical data collection. A total of 24 programs were implemented for each of the three selected languages. The size in lines of code was measured and grouped into six categories, namely, system, module types, input/update transactions, reports, forms, and external inquiry. The line counts were based on the counting rule outlined in section 4.4. During the course of the analysis, a revised counting rule was established to separate file inquiry subroutines from the calling program due to the effects of outliers observed in the initial dataset. Linear regression was used to study the size relationships between the three languages based on the six categories of data mentioned above. The F-distribution test statistic (F-ratio) was used to assess the analysis of variance for the bivariate regression models. Three cases were analysed for each of the six categories of data. Case (a) relates Informix with AREV. Case (b) relates Cobol with AREV. Case (c) relates Cobol with Informix.

Chapter 5 provides a detailed discussion of the analysis of the empirical data which is summarized below.

The regression analyses conducted indicate that more than two-thirds of the variation in function size of one language can be related to a similar function size of another language and that the inter-language relationships for the function categories were statistically significant at a level of 0.025. These results imply that there was a size relationship between languages based on similar functions in lines of code.



actual size (LOC)

Initial size estimate (LOC)

comparison of initial size estimate (LOC) and actual size (LOC)

ER_A *Expansion Ratio for AREV*

ER_I *Expansion Ratio for Informix*

ER_M *Expansion Ratio for Micro Focus Level II Cobol*

Note : The expansion ratios used here were based on published values.

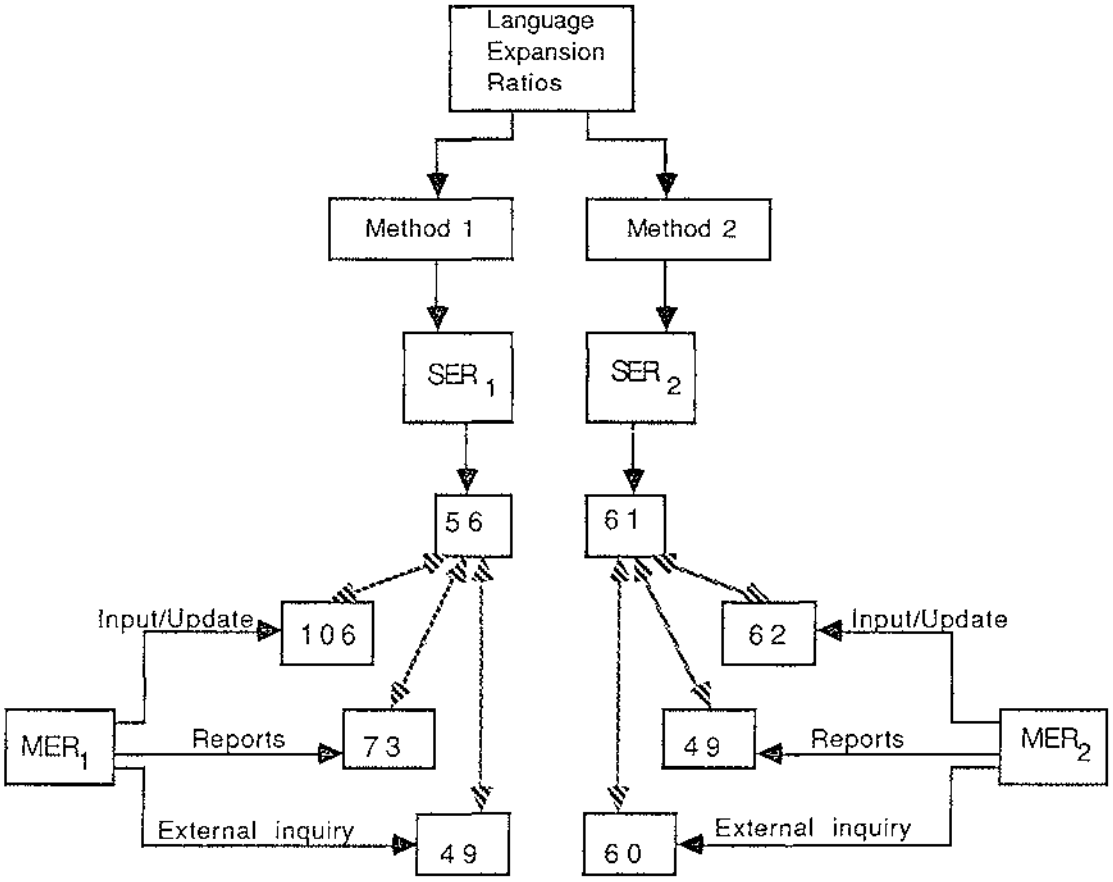
Fig. 6.1 Results of software size estimates.

The level of a language was also seen to be related to the relative size of the resulting system programmed in it. In this case study, AREV and Informix are two 4GLs which require about half the size of an equivalent Cobol implementation. The non-procedural concepts of the 4GLs effect a general savings in terms of effort and size in lines of code. However, the savings in effort do not extend, in real terms, to more complex systems. These observations were found to be consistent with the study of each module type category. Based on these analyses, it can be concluded that there were significant relationships of similar functions, based on their relative sizes in lines of code, between different languages and within a specified application domain. This result also indicates the validity of language expansion estimation of function size based on the ability of obtaining such functional categories for an application.

An initial estimate of the required size of the application, using published expansion ratios, in each of the languages used was found to have significant difference with the actual size. This is shown in Fig. 6.1 above. The expansion ratios used were published values which were considered to be conservative approximations of the actuals. Because each expansion ratio was not specifically calibrated for the languages used, differences were anticipated. However, this result indicates the subjectivity of these expansion ratios when used in its present form.

Analysis of the expansion ratios, calculated using the empirical size data, showed that the function value for each module type category was not adequately represented to reflect a consistent size equivalence in function points. The difference between the module type expansion ratio and the system expansion ratio was quite significant when Albrecht's FPA was used; both of which should be consistent with each other. This is shown in Fig. 6.2 where Cobol is used as an example.

The expansion ratios obtained, using the measured size data and Albrecht's function points as the common denominator, showed some inherent inconsistencies in the determination of function points. The similar patterns found for all three languages was seen to be the result of inadequate weighting factors when calculating the function points to capture the characteristic size of individual function categories.



Method 1	*Albrecht's FPA method*
Method 2	*Symons' MARK II method*
SER	*System Expansion Ratio*
MER	*Module type Expansion Ratio*
	Comparison of SER and MER for each method

Fig. 6.2 Results of Language Expansion Ratios for Micro Focus Level II Cobol.

In comparison, the expansion ratios obtained using the function points determined by the Mark II method showed some improvements in the results (see Fig. 6.2). It was found that the Mark II method applied a more objective measure in calculating the function points. These observations show that there is a need for the weighting factors associated with each function categories to be fine-tuned to allow a more representative size in function points to be determined. Also, a more objective measure of function value is necessary for the size in function points to be representative of the characteristic size of the software.

Language Expansion Ratios were found to give insufficient representation of the characteristic size of a system because of the lack of consistency in the weighting factors used in determining the function points. This does not allow greater accuracy in software size prediction. However, the study shows that more objective measures can be achieved to allow a representative measure to be developed. The use of expansion ratios was found to be valid for this case study where size relationships for similar functions between languages were statistically significant. This study provides an indication of the factors which affect the determination of the expansion ratios and considers them to be inadequate when applied in their present form. The usefulness of expansion ratios in sizing models should, however, encourage continuing research to calibrate them using a larger dataset so that they can be more confidently applied in software sizing.

List of References

[ALBR83] Albrecht, A. J. et al (1983) : Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, IEEE Trans. on Softw. Eng., Vol. SE-9, No. 6, pp. 639-652.

[BASI81] Basili et. al. (1981) : Programming Measurement and Estimation in the Software Engineering Laboratory, J. of Syst. and Softw., Vol. 2, pp. 47 -59.

[BASI83] Basili et al (1983) : Metric Analysis and Data Validation Across Fortran Projects, IEEE Trans. on Softw. Eng., Vol. SE-9, No. 6, pp. 652-663.

[BEHR83] Behrens, C. A. (1983) : The Productivity of Computer System Development Activities with Function Points, IEEE Trans. on Softw. Eng., Vol. SE-9, No. 6, pp. 648-652.

[BOEH73] Boehm, B. W. (1973) : Software and its Impact: A Quantitative Assessment, Reprinted from Datamation Magazine, May 1973, pp. 1-20.

[BOEH81] Boehm, B. W. (1981) : Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

[BOEH84] Boehm, B. W. (1984) : Software Engineering Economics, IEEE Trans. on Softw. Eng., Vol. SE-10, No. 1, pp. 1-29.

[BOLA87] Boland, R. J. et. al. (1987) : Critical Issues in Information Systems Research, pp. 113-134, John Wiley & Sons Ltd.

[BROO75] Brooks, F. P. (1975) : The Mythical Man-Month, Addison-Wesley.

[CARD87a] Card, D. N. et al (1987) : Resolving The Software Science Anomaly, The J. of Syst. and Softw., Vol. 7, pp. 29-35.

[CARD87b] Card, D. N. et al (1987) : Comments on Resolving The Software Science Anomaly, The J. of Syst. and Softw., Vol. 7, pp. 83-84.

[CONT86] Conte et al (1986) : Software Engineering Metrics and Models, The Benjamin/Cummings Publishing Company, Inc., 2727 Sand Hill Road, Menlo Park, California 94025.

[COSM87a] Cosmos, Inc. (1987) : Advanced Revelation - Technical Reference, Cosmos, Incorporated, USA.

[COSM87b] Cosmos, Inc. (1987) : Advanced Revelation - Installation & Introduction, Cosmos, Incorporated, USA.

[COSM87c] Cosmos, Inc. (1987) : Advanced Revelation - Using Advanced Revelation, Cosmos, Incorporated, USA.

[COSM87d] Cosmos, Inc. (1987) : Advanced Revelation - Tutorial, Cosmos, Incorporated, USA.

[COSM87e] Cosmos, Inc. (1987) : Advanced Revelation - Menus & Windows Reference, Cosmos, Incorporated, USA.

[CRUI82] Cruickshank, R. D. et al (1982) : Quantification of Software Projects, pp. 139-148.

[DACS87] Dacs (1987) : A Descriptive Evaluation of Software Sizing Models, IIT Research Institute, 4550 Forbes Boulevard, Suite 300 Lanham, Maryland 20706-4324.

[DEMA88] DeMarco, T. (1988) : An Algorithm For Sizing Software Products, Associate, Yourdon Inc., New York.

[FRYE84] Fryer, A. D. T. (1984) : Cobol on Microcomputers, Micro Focus Ltd, U.S.A.

[GANE77] Gane, C. P. (1977) : Structured System Analysis, Prentice-Hall, USA.

[HALS77] Halstead, M. H. (1977) : Elements of Software Science, Elsevier North-Holland, New York.

[INFO87a] Informix Softw., Inc. (1987) : Informix - 4GL, Version 1.10: User Guide, Informix Software, Incorporated., USA.

[INFO87b] Informix Softw., Inc. (1987) : Informix - 4GL, Version 1.10: Reference Manual, Volume One, Informix Software, Incorporated., USA.

[INFO87c] Informix Softw., Inc. (1987) : Informix - 4GL, Version 1.10: Reference Manual, Volume Two, Informix Software, Incorporated, USA.

[ITAK82] Itakura et al (1982) : A Model for Estimating Program Size and Its Evaluation, IEEE Trans. on Softw. Eng., June 1982, pp. 104-109.

[JEFF87] Jeffery, D. R. et al (1987) : Characterising Resource Data: A Model for Logical Association of Software Data, Technical Report TR-1848, NSG-5123, May1987, pp. 1-35.

[JEFF88] Jeffery, D. R. et al (1988) : The Validity, Reliability, and Practicality of Function Points as a Measure of Software Size, Dept. of Information Systems, University of New South Wales, P.O. Box 1, Kensington, N.S.W. 2033, Aust..

[JONE86] Jones, C. (1986) : Programming Productivity, McGraw-Hill Book Company, Printed in USA.

[LI87] Li, H. F. et al (1987) : An Empirical Study of Software Metrics, IEEE Trans. on Softw. Eng., Vol. SE-13, No. 6, pp. 697-708.

[LIM80] Lim, P. A. (1980) : A Guide to Structured Cobol with Efficient Techniques and Special Algorithms, Van Nostrand Reinhold Company, 135 West 50th Street, New York, N.Y. 10020.

[LOW88] Low, G. C. et al (1988) : Function Points in the Estimation and Evaluation of the Software Process, Dept. of Information Systems, University of New South Wales, P.O. Box 1, Kensington, N.S.W. 2033, Aust..

[MICR83a] Micro Focus Ltd (1983) : Personal Cobol, Version 1.1, Getting Started, Micro Focus Ltd, England.

[MICR83b] Micro Focus Ltd (1983) : Level II Cobol, Version 1.1, Language Reference , Manual, Micro Focus Ltd, England.

[NORM75] Norman, H. N. (1975) : Statistical Package for the Social Sciences, McGraw-Hill Book Company, Inc., USA.

[PRES87] Pressman, R. S. (1987) : Software Engineering, a Practitioner's Approach, McGraw-Hill Inc., USA.

[RUBI85] Rubin, H. A. (1985) : The Art and Science of Software Estimation : Fifth Generation Estimators, J. of Parametrics, Vol. 5, No. 2, pp. 50-65.

[SACH84] Sachs, L. (1984) : Applied Statistics, a Handbook of Techniques, Springer- Verlag New York Inc., USA.

[SCHN78] Schneider, V. (1978) : Prediction of Software Effort and Project Duration - Four New Formulas, ACM Sigplan Notices, pp. 49-59.

[SHEN83] Shen et al (1983) : Software Science Revisited: A Critical Analysis of The Theory and Its Empirical Support, IEEE Trans. on Softw. Eng., Vol. SE-9, No. 2, pp. 155-165.

[SHEP88] Shepperd, M. (1988) : A Critique of Cyclomatic Complexity as a Software Metric, Softw. Eng. J., March, pp. 30-36.

[SYMO88] Symons, C. R. (1988) : Function Point Analysis: Difficulties and Improvements, IEEE Trans. on Softw. Eng., Vol. 14, No. 1, pp. 2-10.

[VERN87] Verner, J. et al (1987) : A Model for Software Sizing, The J. of Syst. and Softw., Vol. 7, pp. 173-177.

[VERN88a] Verner, J. et al (1988) : Technology Dependence in Function Point Analysis: A Case Study and Critical Review, pp. 1-12, Computer Science Dept., Massey University, NZ.

[VERN88b] Verner, J. et al (1988) : Estimating Size and Effort in Fourth-Generation Development, IEEE Softw., pp. 15-22.

[WALS77] Walston, C. E. et al (1977) : A Method of Programming Measurement and Estimation, IBM Syst. J., Vol. 16, No. 1, pp. 54-73.

[YOUR79] Yourdon, E. (1979) : Managing the Structured Techniques, Prentice-Hall Inc., USA.

Appendices

Appendix A

EXAMPLE A

BUSINESS ANALYSIS AND SYSTEM DESIGN SPECIFICATIONS FOR AN

INVENTORY CONTROL AND PURCHASING SYSTEM

1. Introduction

This example gives a step by step illustration of the specifications for a typical Inventory Control and Purchasing application. The documentation from the Business Analysis phase is aimed at users and is hence intended to be user friendly. The documentation from the System Design phase is aimed at those building the system. It is intended to be informative, but above all definitive.

In order to distinguish comment on rationale from parts of the analysis and design documentation, all comment is printed in italics.

2. Inventory Control and Purchasing

A small company maintains a stock of various items for its own use. For the purpose of this illustration, all stock is kept at one location. As and when necessary, the company orders new supplies from one of its suppliers. It therefore keeps track of the kinds of supplies available from each supplier and it also keeps track of its current stock on hand for each kind of item it needs.

3. Business Analysis steps

The steps in the Business Analysis phase are as follows:

1. Business activity analysis
2. Entity type identification
3. Entity type relationship analysis
4. Attribute analysis
5. Population analyis
6. Relationship cardinality analysis
7. Business activity to entity type cross reference
8. Feasability of design alternatives

3.1 Business Activity analysis.

The top level activity is "Inventory Control and Purchasing". This can be categorized as "Acquisition of external resources". It can initially be broken down as follows:

1. Handling new types of stock
2. Ordering stock when required
3. Monitoring stock levels.

The first two of these can be broken down further as follows:

1.1 Categorization of new stock items
1.2 Investigation of possible suppliers

2.1 Select supplier
2.2 Send purchase order

This two level breakdown defines the area of interest for the Business Analysis. The hierarchical relationship between activities on one level and their subordinate activities on the next level down is given by a Business Activity numbering scheme as follows:

1. Handling new types of stock
1.1 Categorization of new stock items
1.2 Investigation of possible suppliers

2. Ordering stock when required
2.1 Select supplier
2.2 Send purchase order

3. Monitoring stock levels.

In practice, there would normally be several top level activities and in a large study they would be in more than one category. Furthermore, each top level activity covered might decompose into three or four levels instead of just two as illustrated here. It is also possible to indicate the hierarchical breakdown using a two column table in which the first column contains the name or number of a higher level activity and the other contains the name or number of one of its parts.

3.2 Entity type identification

The following entity types are relevant to the activity of Inventory Control. The sequence of these entity types has been chosen to aid comprehension when they are elabourated further. Each entity type is categorized in two ways, firstly according to its Entity Type Category and secondly according to its Dynamic Class.

Entity type number	Entity type name	Entity type category	Dynamic class
ET1	STOCK ITEM TYPE	Categorizing	< 1 per year
ET2	STOCK ITEM	Fundamental	< 1 per week
ET3	SUPPLIER	Fundamental	< 1 per week
ET4	SUPPLIER OF STOCK ITEM	Cross ref	< 1 per week
ET5	DATE	Fundamental	< 1 per month
ET6	STOCK ON HAND	Event	< 1 per hour
ET7	PURCHASE ORDER	Event	< 1 per hour
ET8	PURCHASE ORDER LINE	Cross ref	< 1 per hour

The Dynamic Class column places each entity type in one of eight frequency bands to give a rough indication of how often entities of that type are added or modified.

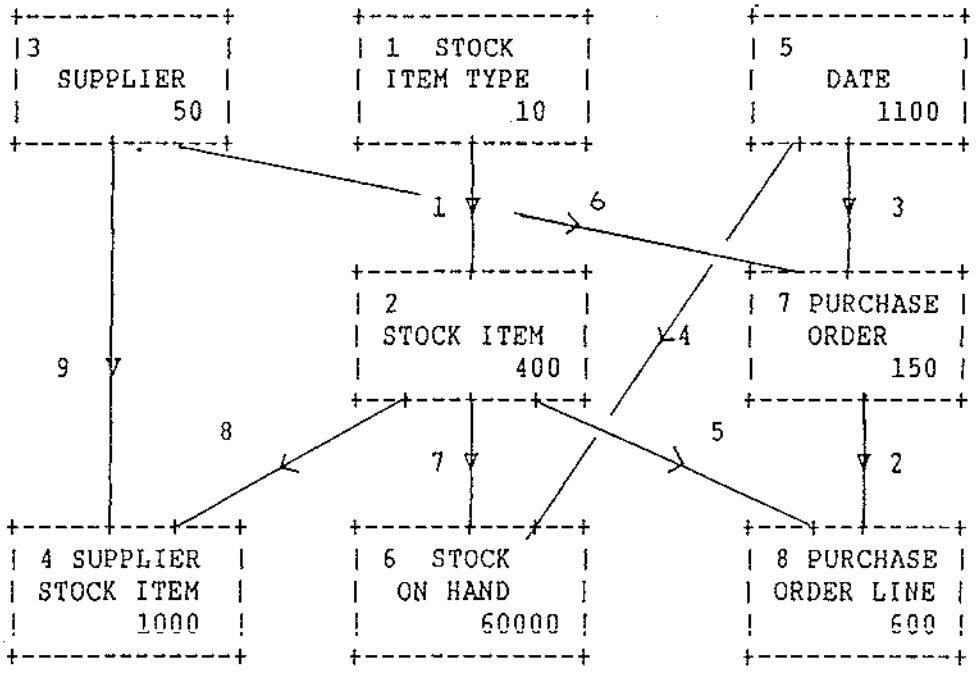
Normally, the list of Entity Types goes through several iterations before analysis is cut off.

3.3 Entity type relationship analysis

The Relationships between the eight Entity Types are shown in the data structure diagram on the following page. In this diagram, each rectangle represents an Entity Type and each arrow between two Entity Types represents a one to many relationship between those Entity Types.

The meaning of the one to many relationship is explained in terms of an example. The one chosen is the Relationship between SUPPLIER and PURCHASE ORDER. The Relationship means that each SUPPLIER has been sent zero one or more PURCHASE ORDERS and that each PURCHASE ORDER has been sent to one and only one SUPPLIER.

Other kinds of Relationship are possible in this and more complex examples, but it is chosen for simplicity to use only one kind of Relationship and one kind of arrow in this example.



The Entity Types and Relationships are each numbered in the above diagram for ease of further reference. The number in the bottom right hand corner of each rectangle is an estimate of the number of entities of the Entity Type which need to be recorded in the system at any one time.

The above data structure diagram is adequate for small applications. If the results of the analysis are being computerized, then the relationships would be recorded in a table and each would be given a name as follows

Relationship No	Relationship Type	Description
1	M1	Stock item in type
2	M1	Line in purchase order
3	M1	Date of purchase order
4	M1	Date of stock on hand
5	M1	Stock item reference on line
6	M1	Purchase order to supplier
7	M1	Stock on hand for stock
8	M1	Stock item available
9	M1	Supplier of stock item

Each Relationship involves two Entity Types. However, each Relationship is assymetric and it is therefore necessary to distinguish between the two Entity Types. This is done by identifying one as independent (I) and the other as dependent (D). Each Relationship are given an identifying number, which corresponds to one of those in the data structure diagram.

Relationship	Entity type	Dependency
1	ET3	I
1	ET2	D
2	ET5	I
2	ET6	D
3	ET4	I
3	ET5	D
4	ET4	I
4	ET7	D
5	ET2	I
5	ET6	D
6	ET1	I
6	ET5	D
7	ET2	I
7	ET7	D
8	ET2	I
8	ET8	D
9	ET1	I
9	ET8	D

3.4 Attribute analysis

The best way to present the results of the attribute analysis to users is by preparing one table for each entity type and representing each attribute as a column in a table. To clarify the meaning of the entity types and attributes, it is advisable to include some rows in the table which as far as possible should consists of "live" data. The user is used to thinking in these terms rather than in terms of abstractions such as entity types.

The following pages contain a description of each entity type and a table of examples for each entity type. In addition there is certain standard information about each entity type which needs to be documented as follows. The format for each entity type is as follows:

1. Description
2. Table of examples, indicating which attributes serve as unique identifiers of each entity type.
3. Estimated volume
4. Whether dynamic
5. Initial source
6. On-going responsibility for updating
7. Questions

The estimated volume indicates the number of entities of each type (i.e. the number of rows in the table). If the data is dynamic, then it indicates how many are created in a one year period.

The "whether dynamic" section indicates roughly how often the data is updated. (*This corresponds to the "dynamic class" in section 3.2*).

The "initial source" section indicates where in the company the first version of the data will come from. (*This is intended to start the user thinking about this problem*)

The "on-going authority for updating" indicates who or which organisation in the company will have the authority and hence the responsibility for making updates to this data. This information will be needed later for designing the security system (or access control system) needed to protect the data in the system.

The "questions" section allows the analyst to note issues relating to the entity type on which he has outstanding questions.

3.4.1 STOCK ITEM TYPE

1. Description.

This entity type is used to provide a simple categorization of STOCK ITEMS for budgeting and management control purposes.

2. Table of examples

Stock item type code	Description
E	Office Equipment (capital expense)
S	Stationery supplies
V	Vehicles (capital expense)
F	First aid supplies
K	Kitchen supplies

The stock item code must be unique.

3. Estimated volume

Less than 10.

4. Whether dynamic

Modified less than once per year

5. Initial source

Code adopted from existing manual system, but modified to ensure each STOCK ITEM TYPE is mutually exclusive.

6. On going updating authority

Financial director.

3.4.2 STOCK ITEM

1. Description

A STOCK ITEM is a class of article which is held in the stores and issued to members of the staff for consumption or use. A STOCK ITEM is not a physical objects, but there may be many physical objects associated with one STOCK ITEM in the stores at any one time.

2. Table of examples

Stock item type	Stock item code	Description
E	5632	Compaq Plus computer
E	6234	Pansonic Printer
E	2357	Canon PC24 photocopier
S	3214	Xerox copying paper
S	1234	Parker fibre tip refill
F	2389	Paracetemol

Each STOCK ITEM has a unique Stock Item Code.

3. Estimated volume

Between 300 and 400

4. Dynamic class

About 50 new STOCK ITEMS are added every year.

5. Initial source

Existing manual system

6. On-going updating authority

Stores manager

3.4.3 SUPPLIER

1. Description

A SUPPLIER is a company who is able to sell the kind of STOCK ITEMS the company keeps in the stores. A SUPPLIER can be recorded as a SUPPLIER without having sold anything to the company.

2. Tables of examples

Supplier number	Supplier name	Supplier address	Phone
43	Rymans	23 South St, Walton	234567
32	Boots	12 High St, Weybridge	678912
12	Inmac	17 First St, London	1234566
13	Mayfair Micros	23 West St, Putney	2452456

Each Supplier Number must be unique.

3. Estimated volume

About 50 in the existing system

4. Whether dynamic

There are about 10 new Suppliers added to the list every year.

5. Initial source

Existing system

6. On going updating authority

Head of purchasing

7. Questions

What happens when a supplier goes out of business?

3.4.4 SUPPLIER OF STOCK ITEM

1. Description

This cross reference indicates which STOCK ITEMS a SUPPLIER is able to supply and at the same time it indicates for each STOCK ITEM the SUPPLIERS from which it can be obtained.

2. Table of examples

Supplier number	Stock item code
43	3214
32	2389
12	3214
13	5632
13	6234
43	1234

The combination of Supplier number and Stock item code must be unique.

3. Estimated volume

Each of 50 SUPPLIERS sells on average about 20 STOCK ITEMS of interest to the company, and this means the estimated volume is about 1000 and that each stock item can be obtained from between 2 and 3 SUPPLIERS.

4. Whether dynamic

Since the number of STOCK ITEMS increases by about 50 per year and each can be obtained from about 2 SUPPLIERS, the average growth is about 100 per year.

5. Initial source

The existing system gives the primary SUPPLIER of each STOCK ITEM and data about other SUPPLIERS will need to be built up.

6. On-going updating authority

Head of purchasing

7. Questions

Should information about each SUPPLIER'S price for the named STOCK ITEM be maintained? If so, should the validity date of the price also be recorded? Who will keep this information up to date?

3.4.5 DATE

1. **Description**

The entity type DATE is included to emphasize the importance of analysing carefully all time dependent data in any information system.

2. Table of examples

Any valid calendar date is a valid example of the entity type DATE. Most computerized systems record dates in the format year, month, date (e.g. 860305 instead of 5 March 1986) in order to make it easier to sort output on dates.

3. Estimated volume

It is assumed that the system will contain 3 years data at any time and the estimated volume is 3 x 365 = 1095 or 1000 as a working figure.

4. Whether dynamic

Since no dates need to be recorded in advance, there will be one date added every work day.

5. Initial source

The dates of purchases orders which are still open when the system goes live will need to be recorded.

6. On-going updating authority

The system will have its own internal calendar and dates will be generated as and when necessary.

7. Questions

Will it be necessary to ask questions from a terminal such as "Which purchase orders were issued on 3 Jan 85?".

3.4.6 STOCK ON HAND

1. Description

This entity type represents the quantity of stock of a given STOCK ITEM which is available in the stores on a given date and at a given time. The quantity is updated each time there is a withdrawal from the stores, each time goods are returned, and each time the stock level is replenished.

2. Table of examples

Stock item code	Date	Time	Quantity
5632 (Compaq Plus computer)	860305	14.45	3
6234 (Pansonic Printer)	860203	10.45	2
6235 (Pansonic Printer)	860401	8.45	1
2357 (Canon PC24 photocopier)	860704	9.30	1
3214 (Xerox copying paper)	861003	17.30	25
1234 (Parker fibre tip refill)	870304	9.40	237
2384 (Paracetemol)	870402	14.30	531

The combination of Stock item code, date and time must be unique. There may be more than one movement for a given STOCK ITEM on any DATE.

3. Estimated volume

Each of 400 STOCK ITEMS has on average 1 movement per week and hence the number of occurrences of this entity type per year is about 20,000.

4. Whether dynamic

There are 400 movements per week, which is about 10 per working hour.

5. Initial source

Initial stock taking before new system goes live. Stock movements prior to that date will not be recorded in the new system.

6. On going updating authority

Store room clerk

7. Questions

What about re-ordering levels?

3.4.7 PURCHASE ORDER

1. Description

A PURCHASE ORDER is issued to a SUPPLIER on a given DATE to order a new supply of one or more STOCK ITEMS.

2. Table of examples

Purchase order no	Date	Supplier number
74321	860406	12
74322	860406	32
74323	860408	13
74324	860408	12

Each Puchase Order number is unique.

3. Estimated volume

There are about 50 PURCHASE ORDERS issued every year.

4. Whether dynamic

Less than 1 per week.

5. Initial source

When the system goes live at the beginning of the Financial Year, all open purchase orders will be recorded in the new system.

6. On-going updating authority

Purchasing department.

7. Questions

3.4.8 PURCHASE ORDER LINE

1. Description

This cross reference entity type indicates the STOCK ITEMS ordered on each PURCHASE ORDER and the quantity of each ordered.

2. Table of examples

Purchase order number	Line no.	Stock Item number	Unit Price	Quantity	Price
74321	1	5632	3000	2	6000
74321	2	6234	250	4	1000
74322	1	2384	2	20	40
74323	1	5632	2800	1	2800

3. Estimated volume

Each of the 50 PURCHASE ORDERS per year contains on average 4 PURCHASE ORDER LINES and hence the estimated volume is 200 per year.

4. Whether dynamic

Less than one per day.

5. Initial source

As for PURCHASE ORDERS

6. On-going updating authority

As for PURCHASE ORDERS.

7. Questions

3.5 Population analysis

This example does not contain entity types which would make the kind of analysis required in this step meaningful.

3.6 Relationship cardinality analysis

The cardinality of a relationship is the average number of entities of one type which correspond to one entity of the other type. The cardinality of each relationship from step 3 (entity type relationship analysis) is shown in the following table.

Relationship No	Cardinality	Description
1	40	Stock item in type
2	4	Line in purchase order
3	0.2	Date of purchase order
4*	60	Date of stock on hand
5*	1.5	Stock item reference on line
6	3	Purchase order to supplier
7*	150	Stock on hand for stock
8	2.5	Stock item available
9	20	Supplier of stock item

This table is one of the most important pieces of analysis with respect to the performance of the system to be designed. Although it shows only average cardinalities, the figures are accurate enough to pin-point potential problem areas and to take evasive action if necessary.

The above figures are predicated on the assumption that three years data will be held in the system at any one time. The cardinality of some of the relationships is proportional to the amount of time dependent data that needs to be held.

For example, the number of instances of STOCK ON HAND generated per year has been estimated at 20,000 (see section 3.4.6). This means that for each STOCK ITEM, there will be on average 150 movements every 3 years. This may or may not be a problem. If it is clear that any querying of the entity type STOCK ON HAND will always be time related (for example "how many withdrawals between 860401 and 860630?"), then there is no problem.

In general, any relationship with an average cardinality of more than 40 will have to be examined carefully when the data base is designed.

3.7 Step 7: Business activity to entity type cross reference.

The aim of this step is to check that the business activities specified in step 1 are consistent with the entity types from step 2.

The reason that this step is not included earlier is that, in practice, steps 3 to 6 can cause a re-iteration of step 2 and even of step 1. Some analysts might, however, find it helpful to prepare this cross reference after completion of step 2.

The entity types are as follows:

Entity type number	Entity type name
ET1	STOCK ITEM TYPE
ET2	STOCK ITEM
ET3	SUPPLIER
ET4	SUPPLIER OF STOCK ITEM
ET5	DATE
ET6	STOCK ON HAND
ET7	PURCHASE ORDER
ET8	PURCHASE ORDER LINE

The cross reference table then indicates which Business Activities Update each Entity Type and which Retrieve information about it.

Entity type / Business activity	ET1	ET2	ET3	ET4	ET5	ET6	ET7	ET8
1.1 Categorisation of new stock items	R	U						
1.2 Investigation of possible suppliers			U	U				
2.1 Select supplier		R	R	R				
2.2 Send purchase order					R		U	U
3. Monitor stock levels		R			R	R		

This cross reference table is intended as an illustration of inconsistency in order to provide a basis for the following questions which would need to be asked of the user.

The following questions arise from this step.

1. Which activity creates STOCK ITEM TYPE?
2. Which activity creates STOCK ON HAND?
3. Which activity makes use of PURCHASE ORDER?

There are three ways of handling these questions. One is to revise the results of steps 1 to 7 in order to have a consisitent analysis. The second and other extrememm is to leave the problems to be solved in the System Design phase. The third is to redo steps 1 and 7 before going on to the analysis.

To answer these questions, a revised set of Business Activities is needed. These are as follows:

1. System start up

2. Handling new types of stock
   2.1 Categorization of new stock items
   2.2 Investigation of possible suppliers

3. Monitoring stock levels
   3.1 Keeping stock on hand up to date
   3.2 Establishing re-order levels

4. Ordering stock when required
   4.1 Select supplier
   4.2 Send purchase order

5. Purchase order follow up

The cross reference table between Business Activity and Entity Types is then as follows:

Entity type / Business activity	ET1	ET2	ET3	ET4	ET5	ET6	ET7	ET8
1. System start-up	U				U			
2.1 Categorisation of new stock items	R	U						
2.2 Investigation of possible suppliers			U	U				
3.1 Keep stock on hand up to date	R	R			R	U		
3.2 Establish re-order levels	R	U						
4.1 Select supplier		U	U	U				
4.2 Send purchase order					R	R	U	U
5. Purchase order follow up					R		R	R

This cross reference table now indicates that each Entity Type is updated by a Business Activity and furthermore each Entity Type is used by at least one Business Activity.

4. System Design steps

The steps in the System Design phase are as follows:

1. Entity type to table assignment
2. Attribute to column assignment
3. Table specification
4. Definition of referential constraints
5. Task definition
6. Specification of stepping stones
7. Task to menu assignment
8. Task to table cross reference
9. Assignment of tasks to access control classes

It is assumed that the questions raised in section 3.4.3, 3.4.4, 3.4.5, and 3.4.6 have been answered by the user and the designer is required to take these answers into account.

4.1 Step 1: Entity Type to Table assignment

It is felt to be advisable to make a number of changes to the results of the Business Analysis in order to provide a better System Design. The design consists of the following ten Tables.

Table number	Description
T1	STOCK ITEM TYPE
T2	STOCK ITEM
T3	SUPPLIER
T4	SUPPLIER OF STOCK ITEM TYPE
T5	CALENDAR DATE
T6	STOCK ON HAND
T7	PURCHASE ORDER
T8	PURCHASE ORDER ITEM
T9	STOCK WITHDRAWAL
T10	STOCK REPLENISHED

The changes will be clarified shortly. The data structure diagram in Figure A.3 shows the above ten Tables and how they are inter-dependent. (The rectangles in this diagram should now be interpreted as relational tables. Each arrow is to be interpreted as a referential constraint on the table pointed to. The contraint refers in each case to matchable columns in the table from which the arrow points.)

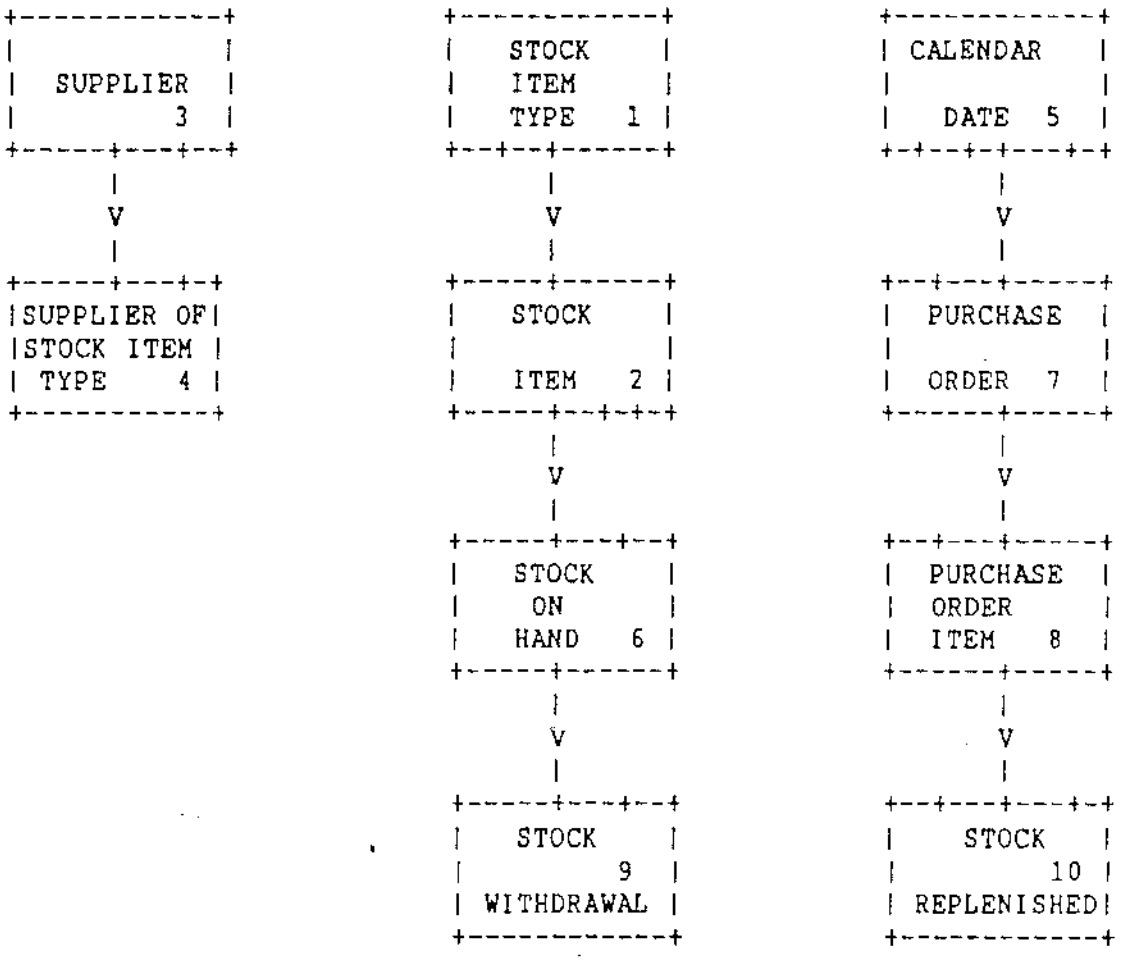


Figure A.3 Revised data structure diagram for System Design

The cross reference showing how the Entity Types from the Business Analysis phase have become Tables in the System Design phase is as follows:

ENTITY TYPE	TABLE 1	2	3	4	5	6	7	8	9	10
1			1							
2		1								
3	1									
4					1					
5							1			
6								1		
7						1				
8										

The use of the figure "1" in the above cross reference, indicates in each case that one Entity Type has been converted into one Table. In more complex examples it is sometimes necessary to convert two or more Entity Types into one Table, or one Entity Type into two or more Tables.

It should be noted that this example indicates that one of the Entity Types, namely number 8, SUPPLIER OF STOCK ITEM has been omitted on the basis that it is impossible to keep such information up to date. Three Tables not resulting from the Business Analysis are included as follows:

4. SUPPLIER OF STOCK ITEM TYPE
9. STOCK WITHDRAWAL
10. STOCK REPLENISHED

4.2 Step 2: Attribute to Column assignment

The following list of Column Names consists of the Attribute Names from step 4 of the Business Analysis phase plus others necessitated by the introduction of the three Tables which are not from the results of the Business Analysis stage.

STOCK CODE
STOCK ITEM NAME
CONSIGNMENT
DELIVERY
ISSUE
PRICE
SUPPLIER NUMBER
SUPPLIER NAME
STOCK ITEM TYPE CODE
STOCK ITEM TYPE DESCRIPTION
DATE OF PURCHASE
DATE OF DELIVERY
PURCHASE ORDER NUMBER
PURCHASE ORDER ITEM NUMBER

4.3 Step 3: Table specification

Each Column must belong to a Table. This means that each of the above Column Names should be assignable to one of the ten Tables already identified. It may transpire that some of these Column Names cannot be assigned to a Table. The following cross reference table indicates the initial assignments.

TABLE / COLUMN NAME	1	2	3	4	5	6	7	8	9	10
Stock Code		X				X		X	X	X
Stock Item Name		X								
Delivery										X
Issue									X	
Price								X		
Supplier Number			X		X		X			
Supplier Name			X							
Stock Item Type Code	X	X								
Stock Item Type Description	X									
Date of Purchase							X			
Date of Delivery										X
Date of Withdrawal									X	
Purchase Order Number							X	X		
Purchase Order Item Number								X		
Quantity of Stock						X			X	X
Date					X					

It is normally not useful to prepare a detailed cross reference between Attributes and Columns.

4.4 Step 4: Constraint Definition

There should be at least one uniqueness Constraint defined for each of the ten Tables. Each such Constraint is expressed either for a single Column or for a Column Group of that Table. The uniqueness Constraints for the ten Tables are as follows:

Table	Column or Column Group
STOCK ITEM TYPE	Stock Item Type Code
STOCK ITEM	Stock Item Code
SUPPLIER	Supplier Number
SUPPLIER OF STOCK ITEM TYPE	Supplier Number Stock Item Type Code
CALENDAR DATE	Date
STOCK ON HAND	Stock Item Code Stocktaking Date
PURCHASE ORDER	Purchase Order Number
PURCHASE ORDER ITEM	Purchase Order Number Purchase Order Item Number
STOCK WITHDRAWAL	Stock Item Code Date of Withdrawal Withdrawal Number
STOCK REPLENISHED	Stock Item Code Date of Delivery Replenishment Number

Since there is only one Uniqueness Constraint on each of the 10 Tables, the constraint can also be referred to as a "primary key".

As indicated, each of the fourteen arrows depicted in the data structure diagram represents a referential Constraint. Each of these is a Constraint on the values of one or more Columns in one of the Tables.

The Columns to which the Constraints apply are always "matchable" with the values of Columns in the other Table at the tail of the arrow. The Columns in the Table referred to are the Primary Key for that Table.

The following table indicates the Table and the constrained Columns for each of the fourteen Constraints.

Table no	Constraint type	Column or Column Group	Referred Table no	Column or Column Group
2	R	Stock Item Type Code	1	Stock Item Type Code
4	R	Supplier No	3	Supplier No
4	R	Stock Item Type Code	1	Stock Item Type Code
6	R	Stock Item Code	2	Stock Item Code
6	R	Stocktaking Date	5	Date
7	R	Supplier No	3	Supplier No
7	R	Date of Purchase	5	Date
8	R	Purchase Order Number	7	Purchase Order Number
8	R	Stock Item Code	2	Stock Item Code
9	R	Date of Withdrawal	5	Date
9	R	Stock Item Code	2	Stock Item Code
10	R	Date of Delivery	5	Date of Delivery
10	R	Stock Item Code	2	Stock Item Code
10	R	Purchase Order Number	8	Purchase Order Number
		Purchase Order Item Number	8	Purchase Order Item Number

It should be noted that the first 13 of the above list of referential Constraints are based on individual Columns. The last one is based on a Column Group. All 14 referential Constraints are based on value equality.

As an example, the Constraint on Table 8 (Purchase Order Item) implies that the value in the Column called Purchase Order Number in the Table PURCHASE ORDER ITEM must correspond to an extant value in the Column called Purchase Order Number, but in Table 7 (PURCHASE ORDER).

It should also be noted that there are no referential Constraints on Tables 1, 4 and 5. It will be seen in the structure diagram these three Tables, namely

1. STOCK ITEM TYPE
3. SUPPLIER
5. CALENDAR DATE

are not dependent on any other Tables.

4.5 Step 5: Task Definition

The following design decisions are taken at this point. It is assumed that the Tasks will be defined by first examining each of the ten Tables and deciding which of the following four Task Types are needed:

Add
Modify
Delete
Retrieve

The results of this examination is shown in the following cross reference which identifies 25 different Tasks.

It should be noted that the two Tables PURCHASE ORDER and PURCHASE ORDER ITEM are processed by the same TASKS on the basis that every PURCHASE ORDER has at least one PURCHASE ORDER LINE. It is not meaningful to add a PURCHASE ORDER without adding at least one PURCHASE ORDER ITEM. Furthermore, any retrieval Task which acceses one of these two Tables will also access need to access the other.

Table no	Description	Add	Delete	Modify	Retrieve
1	STOCK ITEM TYPE	1		2	3
2	STOCK ITEM	4		5	6
3	SUPPLIER	7		8	9
4	SUPPLIER OF STOCK ITEM TYPE	10			11, 12
5	DATE	13			
6	STOCK ON HAND	14		15	16, 17,18
7	PURCHASE ORDER	19			20, 21
8	PURCHASE ORDER ITEM	19			20, 21
9	STOCK WITHDRAWAL	22			23
10	STOCK REPLENISHED	24			25

It is also possible to categorize each of these 25 Tasks in terms of the following three categories, for each of which two or three coded alternatives are identifed.

1.	On-Line	O	Batch	B
	Neither	N		
2.	Homing-in	H	Out of the blue	B
3.	Menu driven	M	Fast track	F
	System initiated	S	Calendar initiated	C

Table no	Description	Add	Delete	Modify	Retrieve
1	STOCK ITEM TYPE	1 OHM		2 OHM	3 OHM
2	STOCK ITEM	4 OHM		5 OHM	6 OHM
3	SUPPLIER	7 OOM		8 OOM	9 OOM
4	SUPPLIER OF STOCK ITEM TYPE	10 OHM			11 OHM
					12 OHM
5	DATE	13 OOM			
6	STOCK ON HAND	14 OHF		15 OHF	16 OHM
					17 OHM
7	PURCHASE ORDER	18 OHM			20 OHM
		19 SOS			21 OOM
8	PURCHASE ORDER ITEM	18 OHM			20 OHM
					21 OOM
9	STOCK WITHDRAWAL	22 OHM			23 OHM
10	STOCK REPLENISHED	24 OHM			25 OHM

4.6 Step 6: Specification of stepping stones between tables

In order to assess the need for stepping stones, it is first necessary to update the results of the Cardinality Analysis step from the Business Analysis phase. The revised analysis is presented in the annotated data structure diagram in Figure A.4.

As in Figure A.2, each rectangle contain an estimate in the bottom right hand corner of the number of rows expected in that Table. The number in the top left hand corner is the Table number. The cardinality of the relationship (or in design terms more correctly the constraint) is included in the diagram for each of the 14 referential Constraints.

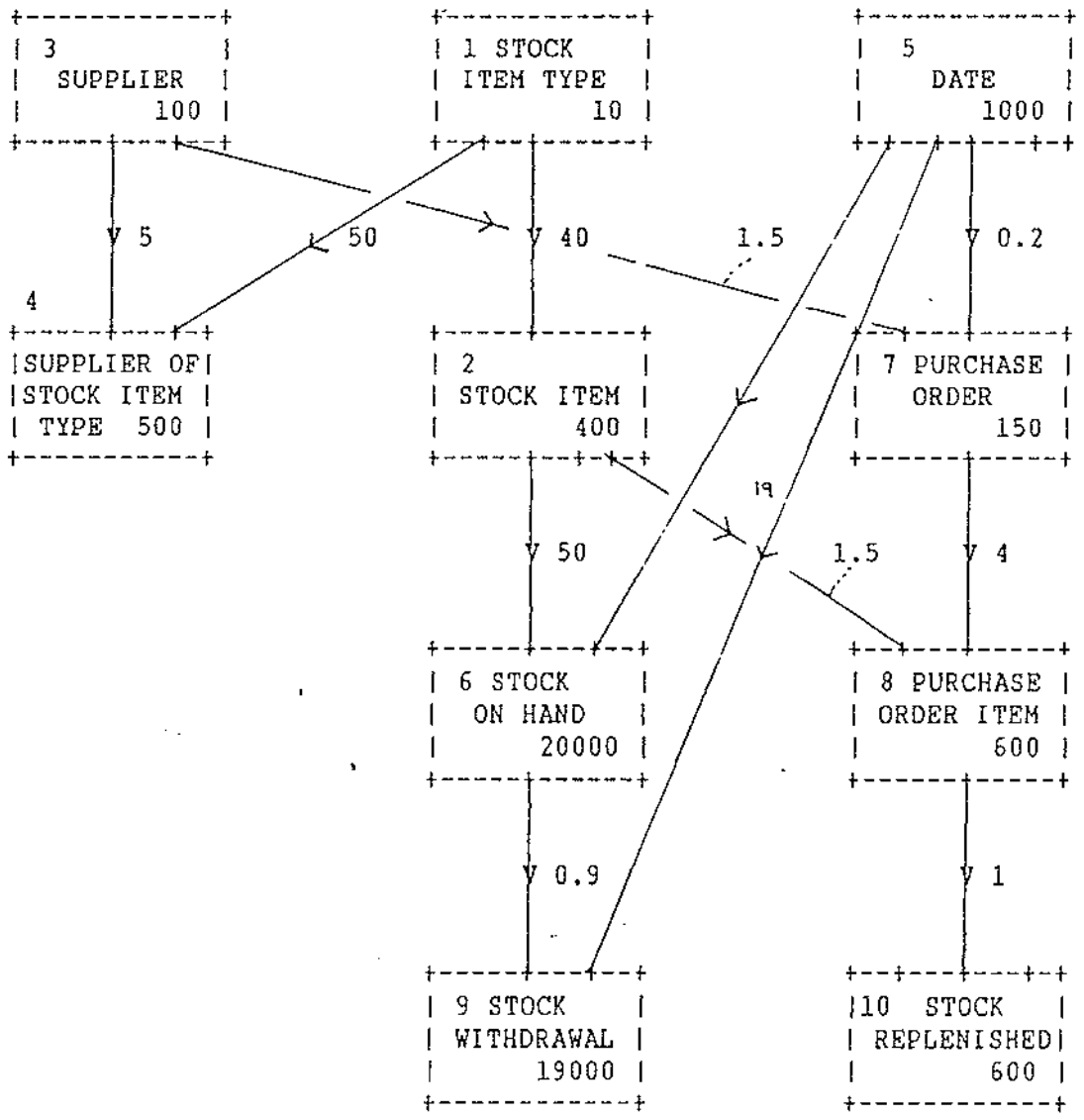


Figure A.4 Cardinalties for System Design tables and constraints

The highest cardinality shown appears to be 50 and it occurs twice. The two instances are the following:

1. From STOCK ITEM TYPE to SUPPLIER OF STOCK ITEM TYPE
2. From STOCK ITEM to STOCK ON HAND.

A cardinality of 50 is not likely to cause tremendous problems and hence no stepping stones need to be introduced.

4.7 Step 7: Task to menu assignments

Each of the 25 Tasks identified in Step 5 now needs to be assigned to a menu. It is proposed to have a two level menu structure, with one top level and four second level menus. The starting point for designing the menu structure is the revised breakdown of Business Activities from the step 8 in the Business Activity phase.

1. System start up

2. Handling new types of stock
   - 2.1 Categorization of new stock items
   - 2.2 Investigation of possible suppliers

3. Monitoring stock levels
   - 3.1 Keeping stock on hand up to date
   - 3.2 Establishing re-order levels

4. Ordering stock when required
   - 4.1 Select supplier
   - 4.2 Send purchase order

5. Purchase order follow up

On the basis of this breakdown, it is proposed to design the top level menu as follows:

```
+-----------------------------------------------------+

          XYZ COMPANY'S INVENTORY CONTROL SYSTEM

                  T O P   L E V E L   M E N U

      1.  System start up

      2.  Handling new types of stock

      3.  Monitoring stock levels

      4.  Purchase order processing and follow-up

      Indicate choice by typing appropriate integer____

+-----------------------------------------------------+
```

Of the 25 Tasks from Step 5, 23 are initiatable from a terminal. Each of these can now be assigned to one of the four second level menus. The assignment is shown in the annotated version of the cross reference table from Step 5 as follows. An S indicates that the Task is system initiated.

Table no	Description	Add	Delete	Modify	Select
1	STOCK ITEM TYPE	1 OHM 1		2 OHM 1	3 OHM 1
2	STOCK ITEM	4 OHM 1		5 OHM 2	6 OHM 2
3	SUPPLIER	7 OOM 1		8 OOM 2	9 OOM 2
4	SUPPLIER OF STOCK ITEM TYPE	10 OHM 1			11 OHM 4 12 OHM 4
5	DATE	13 OOM 1			
6	STOCK ON HAND	14 OHF S		15 OHF 3	16 OHM 3 17 OHM 3
7	PURCHASE ORDER	18 OHM 4 19 SOS S			20 OHM 4 21 OOM 4
8	PURCHASE ORDER ITEM	18 OHM 4			20 OHM 4 21 OOM 4
9	STOCK WITHDRAWAL	22 OHM 3			23 OHM 3
10	STOCK REPLENISHED	24 OHM 3			25 OHM 3

The integer to the right of each of the four rightmost columns indicates the menu to which the Task has been assigned.

As a result of this the four second order menus now appear as follows. The numbers against each Task correspond to the Task numbers assigned in Step 5. This is for documentation purposes only and the system should number its own entries when it generates a menu for a user in a given Access Class.

XYZ COMPANY'S INVENTORY CONTROL SYSTEM

M E N U F O R S Y S T E M S T A R T - U P

1. Add one or more Stock Item Types

2. Modify one or more existing Stock Item Types

3. Display available Stock Item Types

4. Add inititial list of Stock Items

7. Add initial list of suppliers

10. Add list of Stock Item Types carried by a supplier

13. Enter range of dates for which records will be kept

Indicate choice by typing appropriate integer_____

XYZ COMPANY'S INVENTORY CONTROL SYSTEM

M E N U F O R N E W T Y P E S O F S T O C K

5. Modify an exisitng stock item

6. Display list of stock items

8. Modify an existing supplier

9. Display list of suppliers

Indicate choice by typing appropriate integer_____

XYZ COMPANY'S INVENTORY CONTROL SYSTEM

MENU FOR MONITORING S T O C K L E V E L S

15. Manual recording of stock on hand

16. Display stock on hand profile for a given stock item between given dates

17. Display stock on hand on a given date for all stock items of a given type

22. Record a stock withdrawal

23. Display list of stock withdrawals for a given stock item between two dates

24 Record stock replenished

25. Display list of replenishments for a given stock item between two dates

Indicate choice by typing appropriate integer_____

+----------+

XYZ COMPANY'S INVENTORY CONTROL SYSTEM

MENU FOR PURCHASE ORDER PROCESSING AND FOLLOW UP

11. Display all suppliers of a given type of stock

12. Display all types of stock available from a given supplier

18. Generate a purchase order manually

20. Display all purchase orders issued on a given date

21. Display a purchase order with a given number

+----------+

It should be noted that Tasks 14 and 19 do not appear on any of these menus as they are initiated by the system automatically. Task 14 is an automatic updating of the Stock on Hand when there is a withdrawal or a replenishment. Task 19 is an automatic creation of a Purchase Order when the Stock on Hand drops below the Re-order level for that Stock Item.

4.8 Step 8: Assignment of Tasks to Access Control Classes

For the purposes of Access Control, the potential users are grouped into the following mutually exclusive classes.

A. Data administrator
B. Purchasing manager
C. Purchase department staff
D. Warehouse personnel

Each of the 23 Tasks initiatable from a terminal is assigned to one or more of these four classes. The assignments are shown in the following cross reference table.

Task	Access Control Class			
	A	B	C	D
1✓	Y			
2✓	Y			
3✓	Y	Y	Y	Y
4✓		Y		
5✓		Y		
6✓	Y	Y	Y	Y
7✓		Y		
8✓		Y	Y	
9✓	Y	Y	Y	Y
10✓		Y	Y	
11✓	Y	Y	Y	Y
12✓	Y	Y	Y	Y
13✓	Y			
15✓				Y
16✓	Y	Y	Y	Y
17✓	Y	Y	Y	Y
18✓		Y	Y	
20✓	Y	Y	Y	Y
21✓	Y	Y	Y	Y
22✓				Y
23✓	Y	Y	Y	Y
24✓				Y
25✓	Y	Y	Y	Y

It should be noted that all of the retrieval tasks are regarded as being available to all users, becuase none of the information in the system is felt to be confidential. The two system initiated tasks are incluided in the above table for completeness.

4.9 Step 9: Task to table cross references

The following cross reference table illustrates Task Uses Table with the categorization of each Table used by a Task in terms of its Role in the information path.

Table	Task	STOCK ITEM TYPE	STOCK ITEM	SUPPLIER	SUPPLIER OF STOCK ITEM TYPE	DATE	STOCK ON HAND	PURCHASE ORDER	PURCHASE ORDER LINE	STOCK WITHDRAWAL	STOCK REPLENISHED
STOCK ITEM TYPE	1	AS									
	2	MS									
	3	SS									
STOCK ITEM	4	SS	AT								
	5	SS	ST								
	6	SS	ST								
SUPPLIER	7			AS							
	8			MSB							
	9			SS							
SUPPLIER OF STOCK	10	SS		V	AT						
ITEM TYPE	11	SS		R	ST						
	12	R	SS		ST						
DATE	13					AS					
STOCK ON HAND	14 S										
	15	SS	SI			SS	MT				
	16	SS	SI			SS	ST				
	17	SS	SI			SS	ST				
PURCHASE ORDER	18					SS		AT	AT		
	19 S										
	20					SS		ST	ST		
	21							SS	SR		
PURCHASE ORDER ITEM											
STOCK WITHDRAWAL	22	SS	SI			SS	SI			AT	
	23	SS	SI			SS	SI			ST	
STOCK REPLENISHED	24	SS	SI			SS	AI				AT
	25	SS	SI			SS	AI				RT

The codes used in this cross reference table are as follows:

TASK TYPE

- S Select
- A Add
- M Modify
- D Delete

ROLE

- S Starting point
- T Target
- P Intermediate point
- R Retrieval reference
- V Validation reference
- F Inflection point

HOMING IN OR OUT OF THE BLUE

- B Out of the blue
- blank Homing in

This table provides a compact and highly codified definition of the 25 Tasks and of the path to be used by each Task from its starting point or points to its target or targets. It shows which Tables each Task needs to access. The sequence is defined in terms of the starting points, possible intermediate points and tables referenced, through to an end point (target) for each path.

As an example, Task 15 can be described in detail. It accesses the following four Tables:

Starting points:	DATE, STOCK ITEM TYPE
Intermediate point:	STOCK ITEM
Target point:	STOCK ON HAND.

This Task homes in on a Stock Item from Stock Item Type. The terminal user specifies a Date (or the system takes the date from the calendar clock). A row in the Stock on Hand table is then modified.

This example does not cover all the aspects of the design of Tasks. Some Tasks, such as 22 and 24 will need to perform numeric computation in order to modify the content of the Stock on Hand Table.

It should be noted that the content of this cross reference table could also be represented as a series of data flow diagrams in which some or all of the Tables are represetned as "data stores".

Appendix B

FIG. B.1 System Start- up (DFD1)

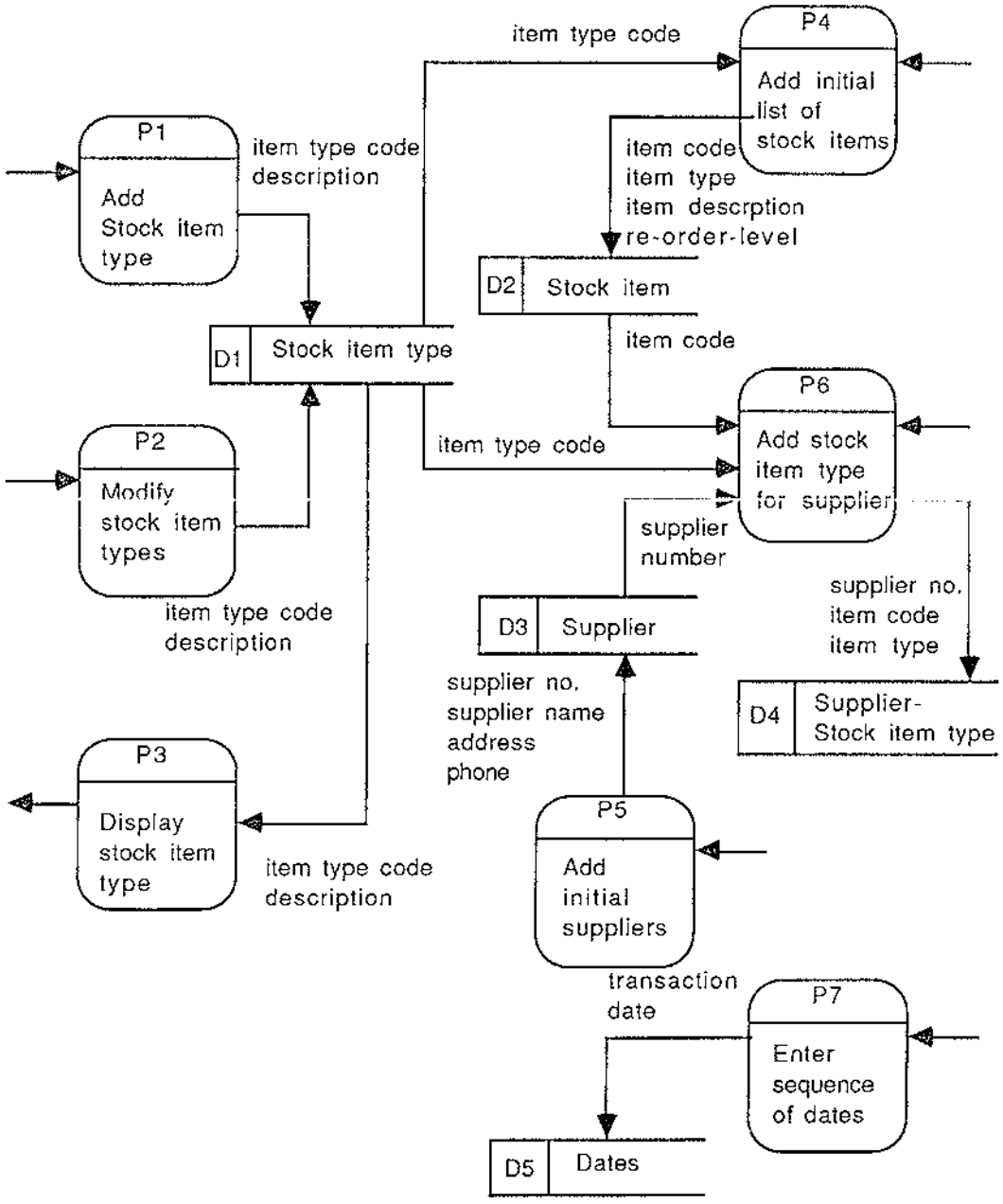


Fig. B.1.1 Process P1

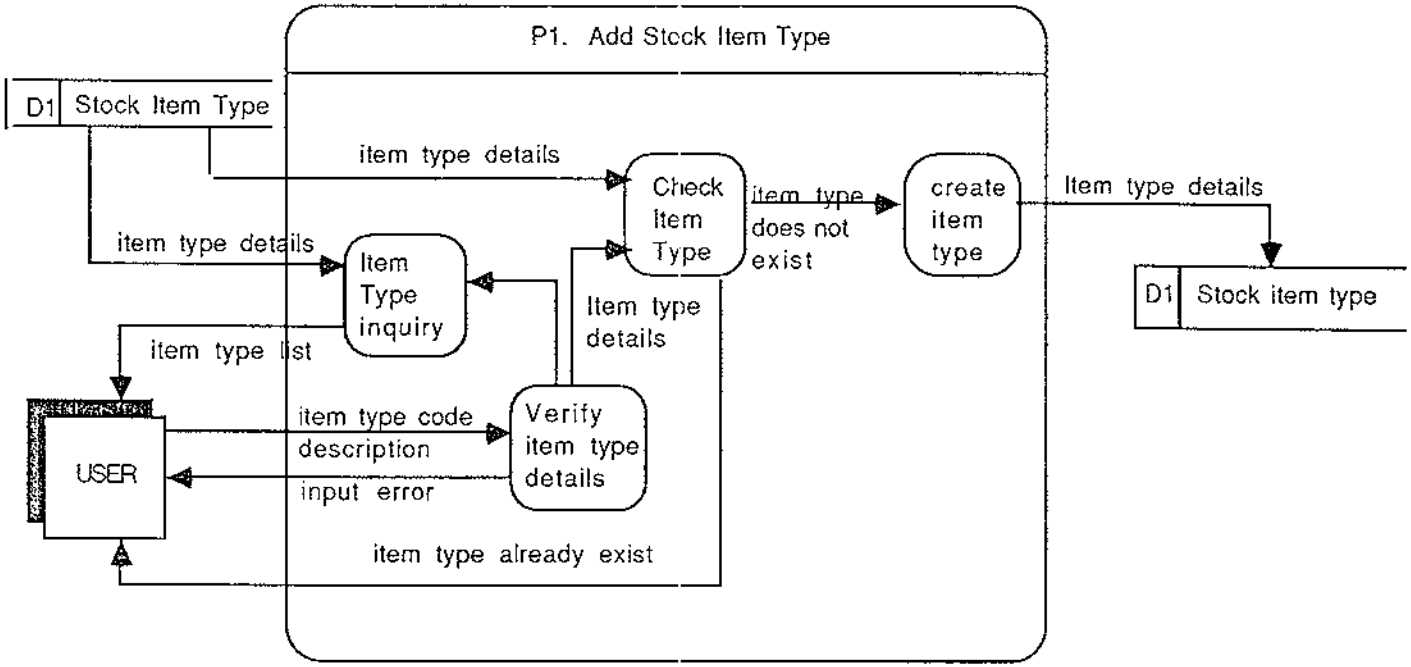


Fig. B.1.2 Process P2

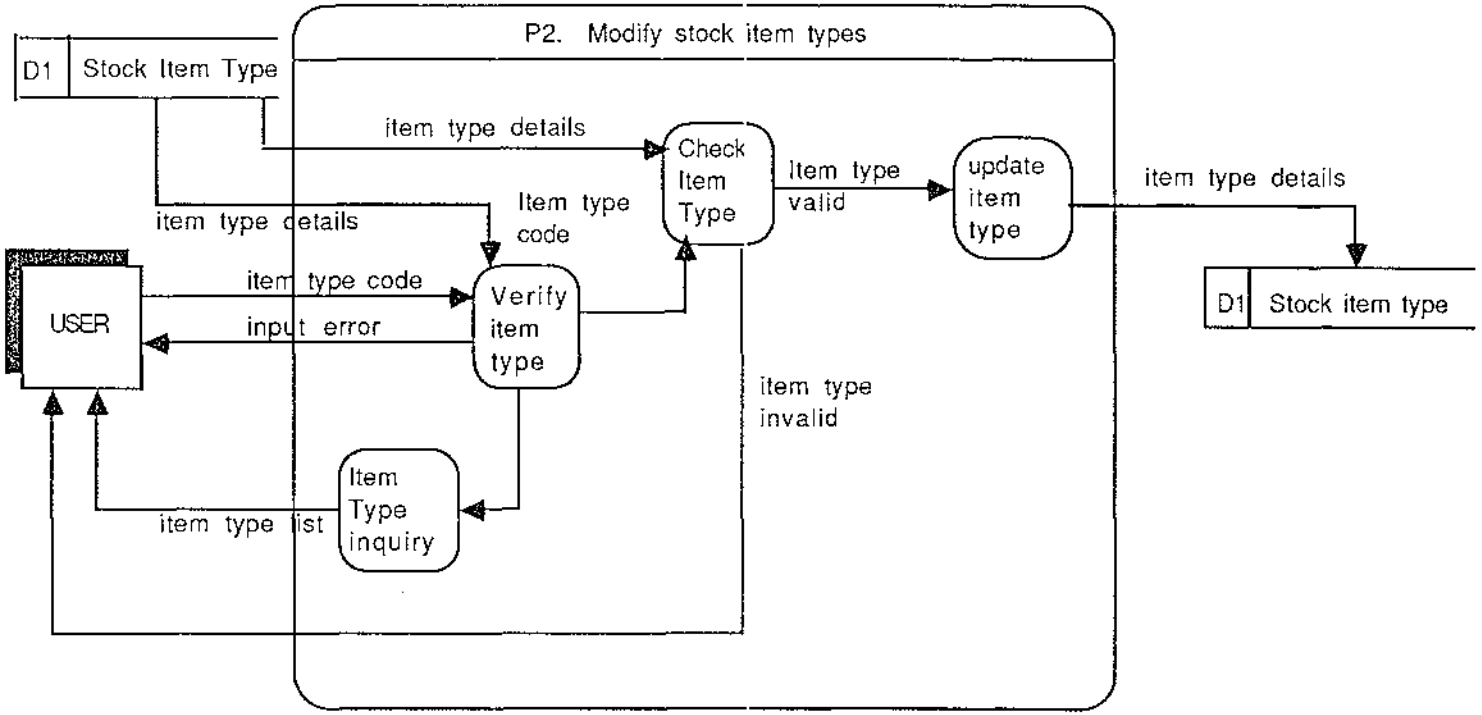


Fig. B.1.3 Process P3

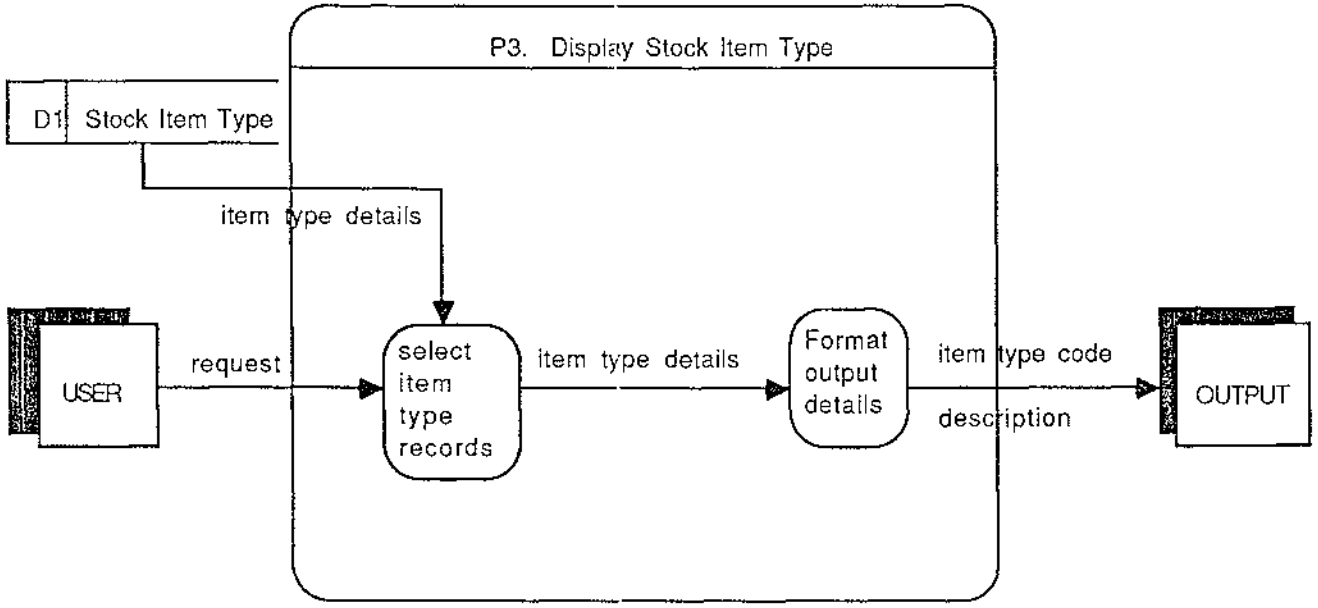


Fig. B.1.4 Process P4

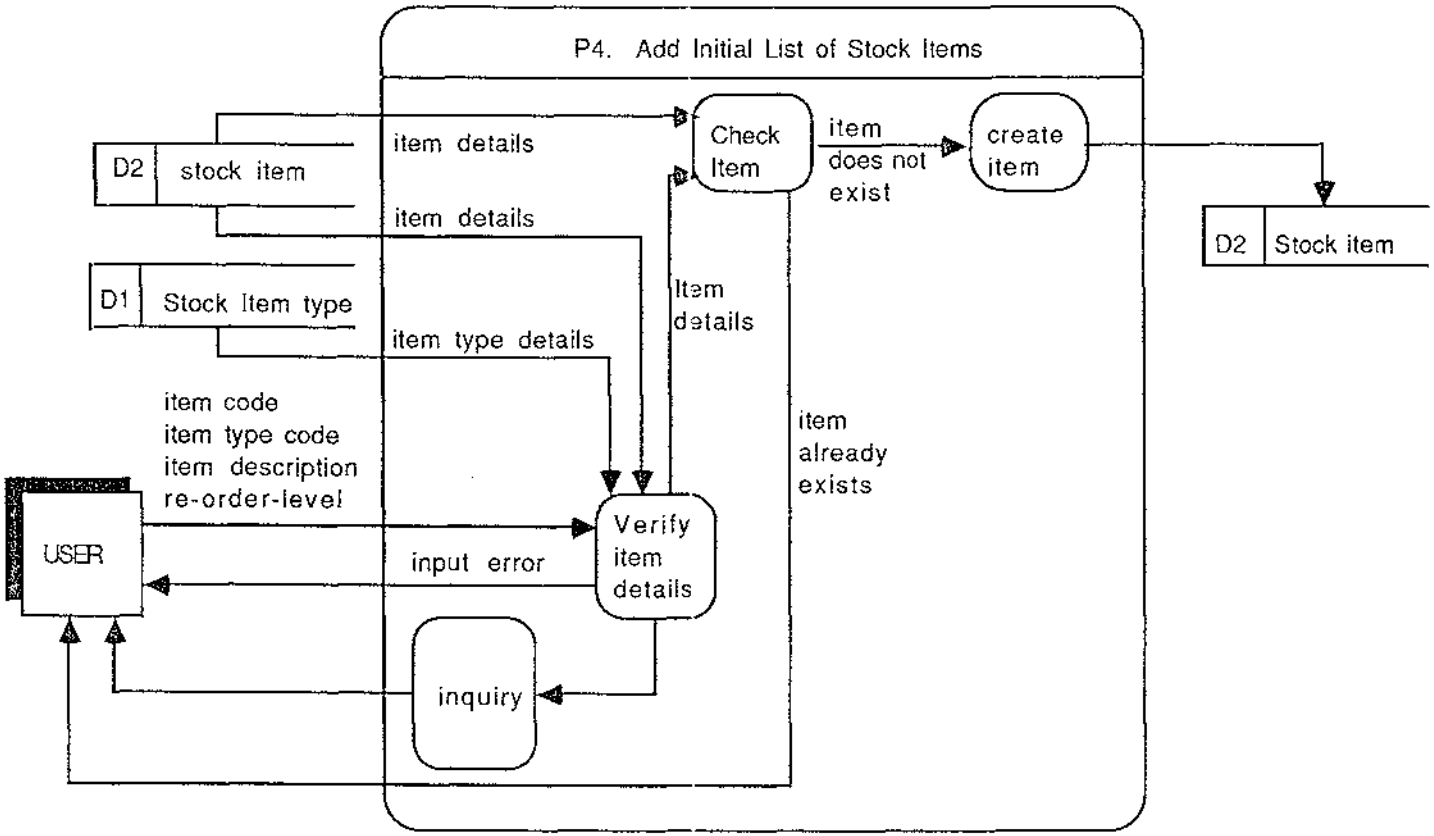


Fig. B.1.5 Process P5

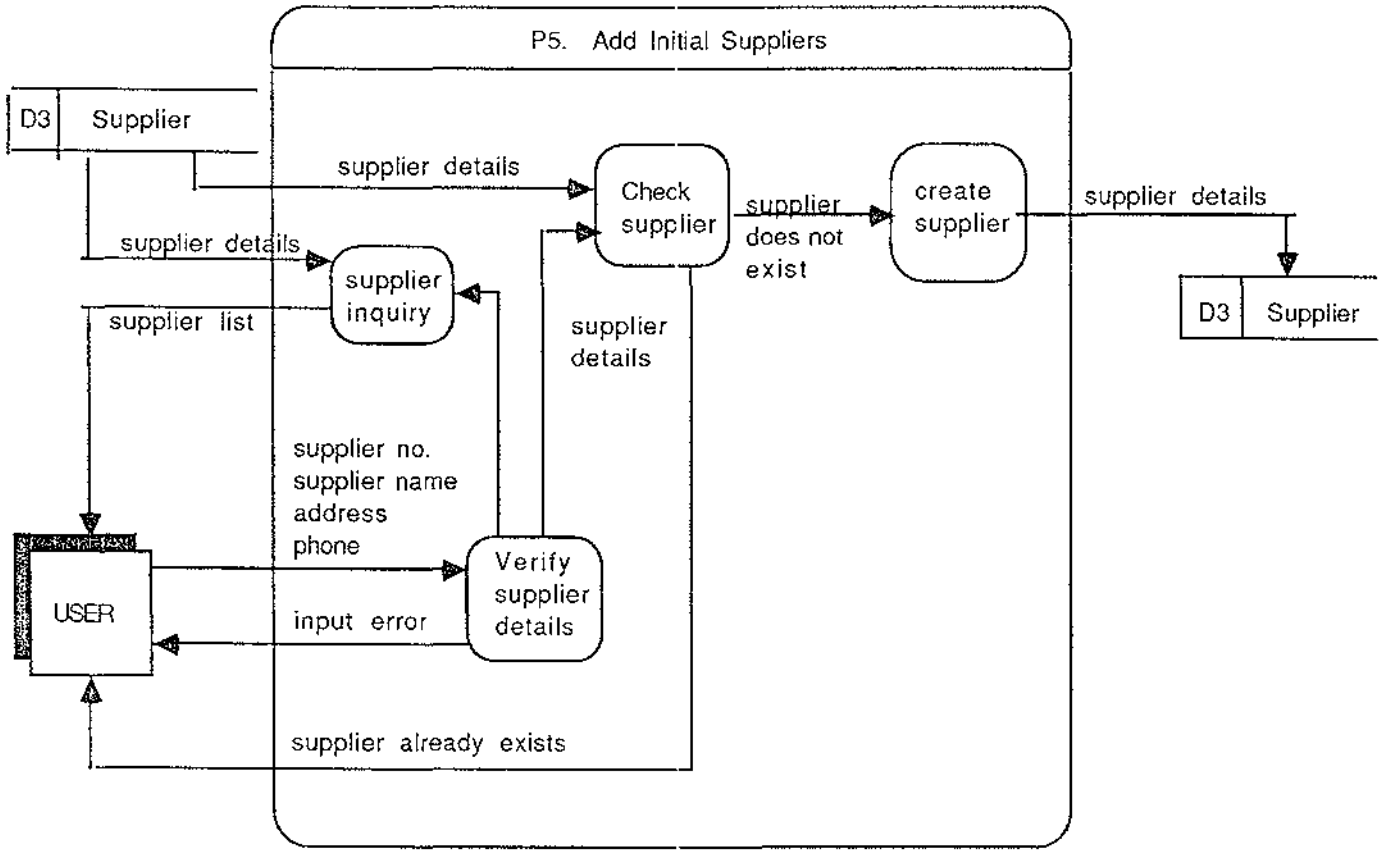
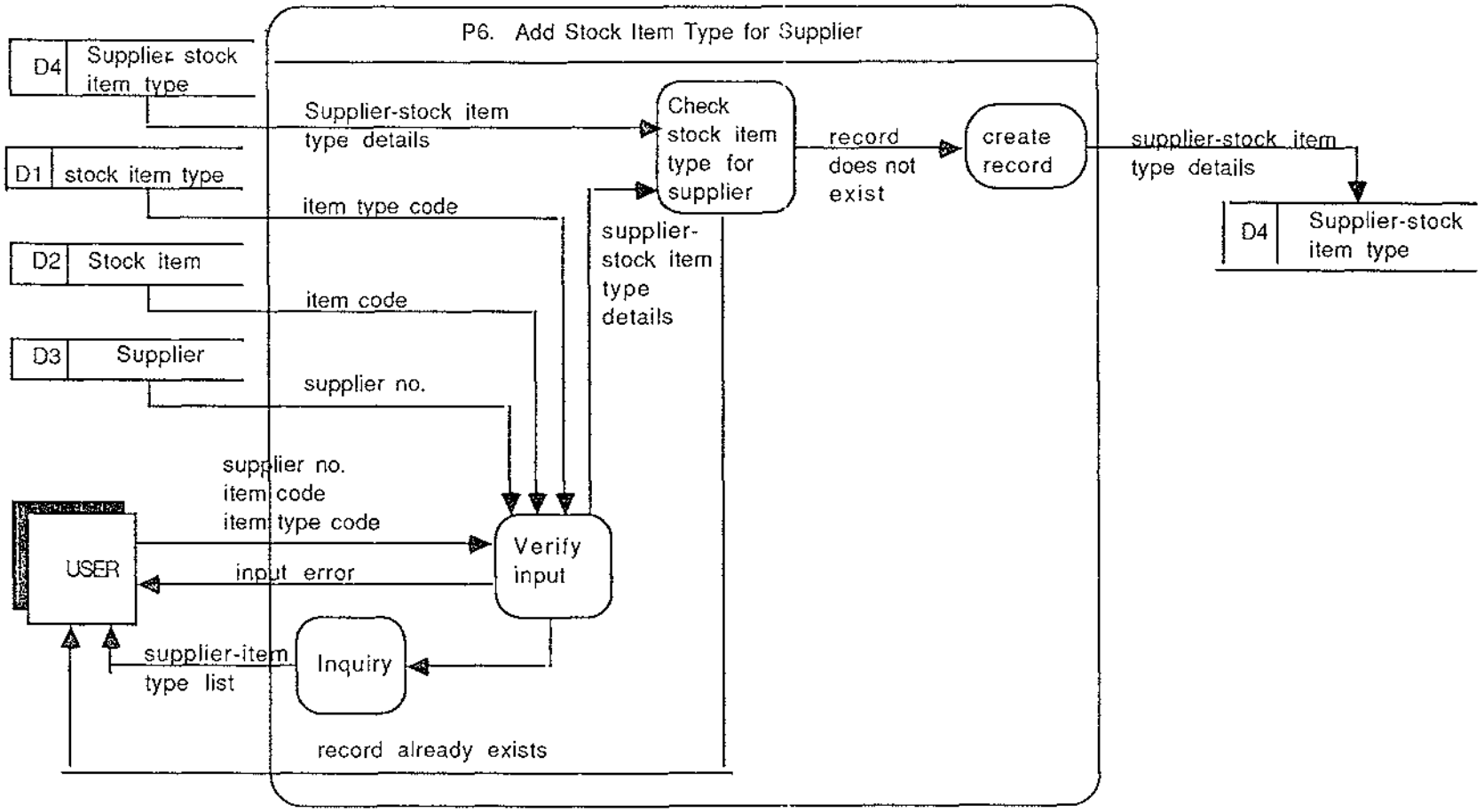


Fig. B.1.6 Process P6



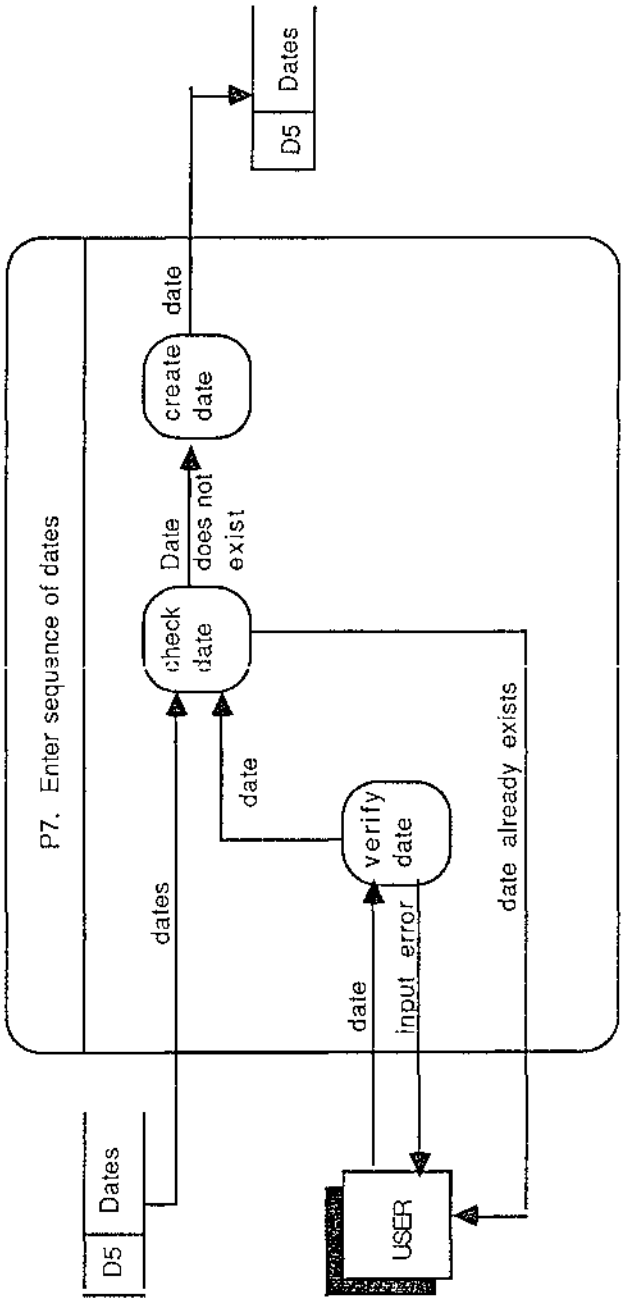


Fig. B.1.7 Process P7

FIG. B.2 New Types of Stock (DFD1)

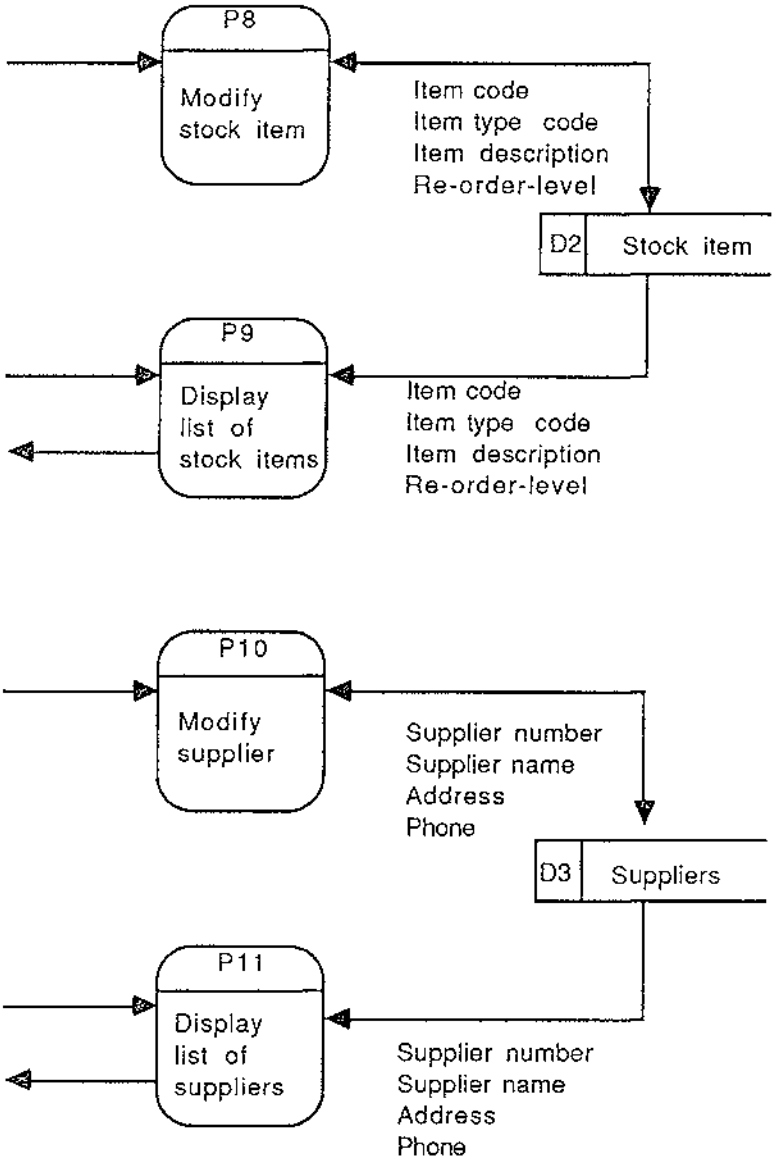


FIG. B.2.1 Process P8

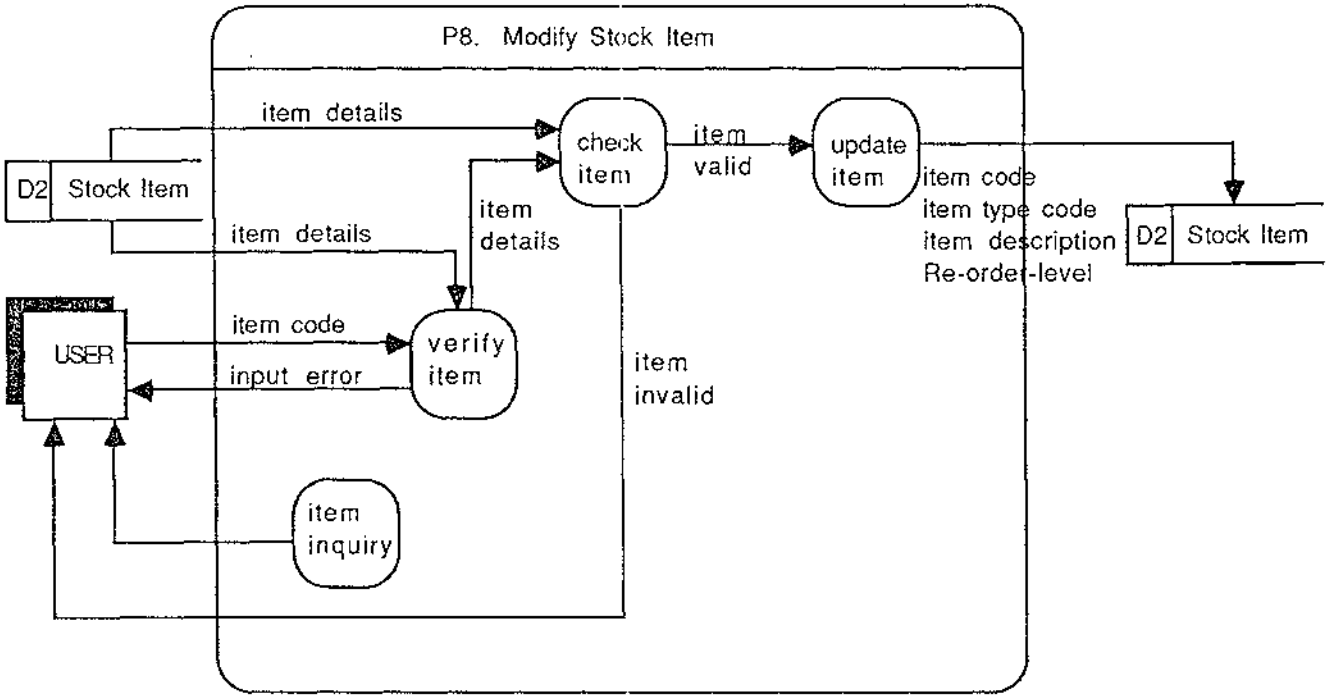


FIG. B.2.2 Process P9

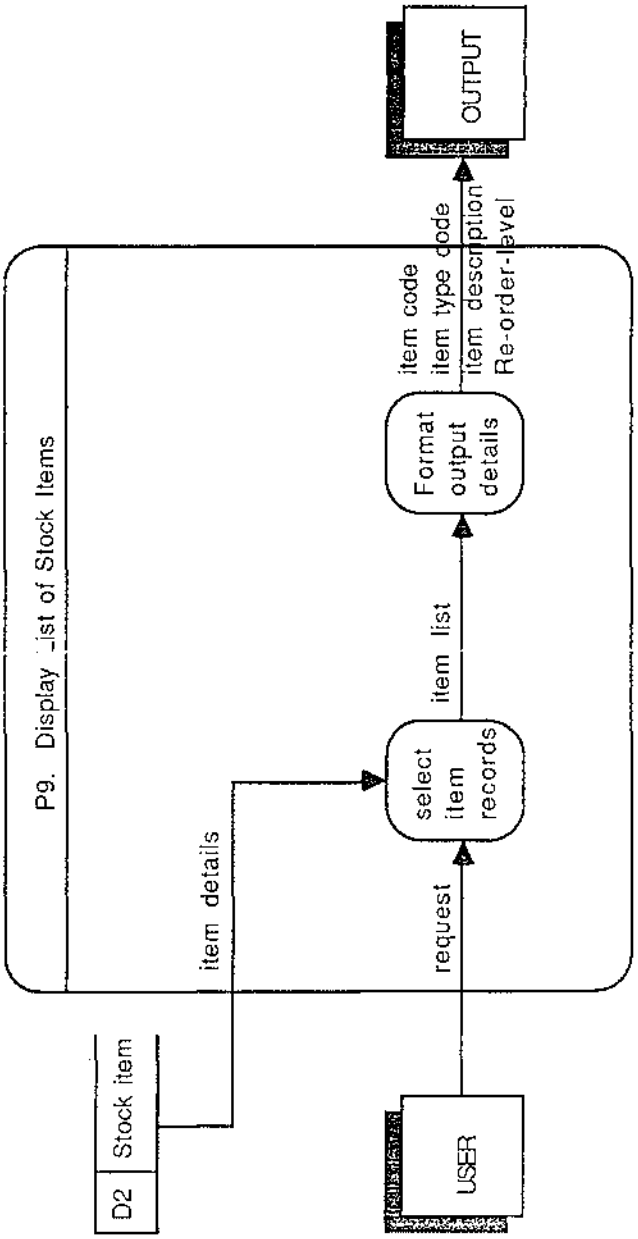
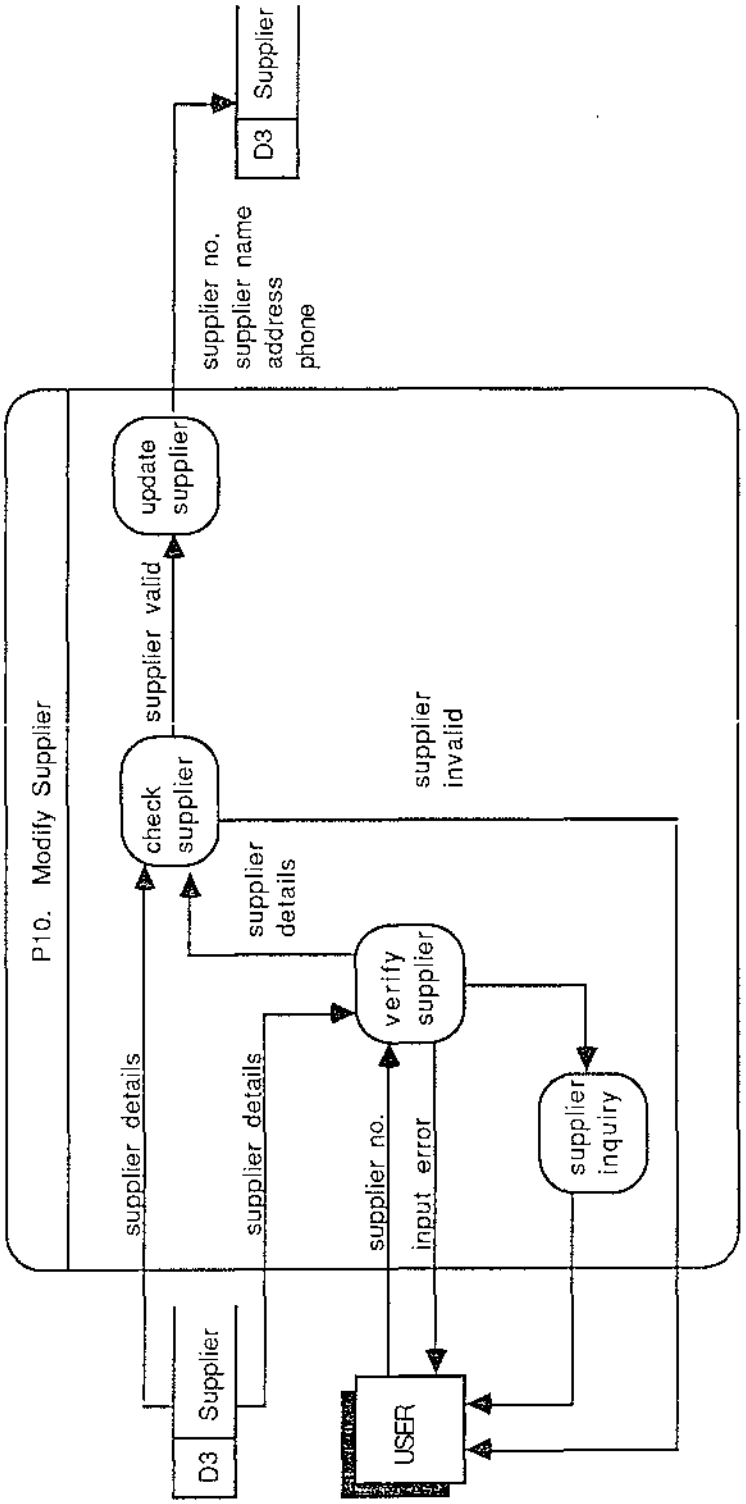


FIG. B.2.3 Process P10



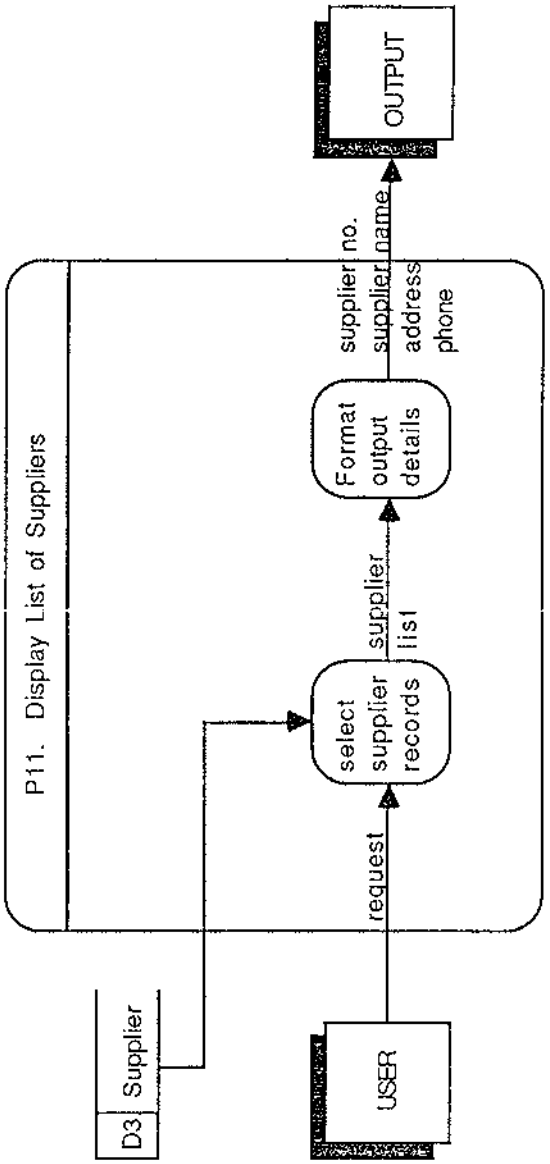


FIG. B.2.4 Process P11

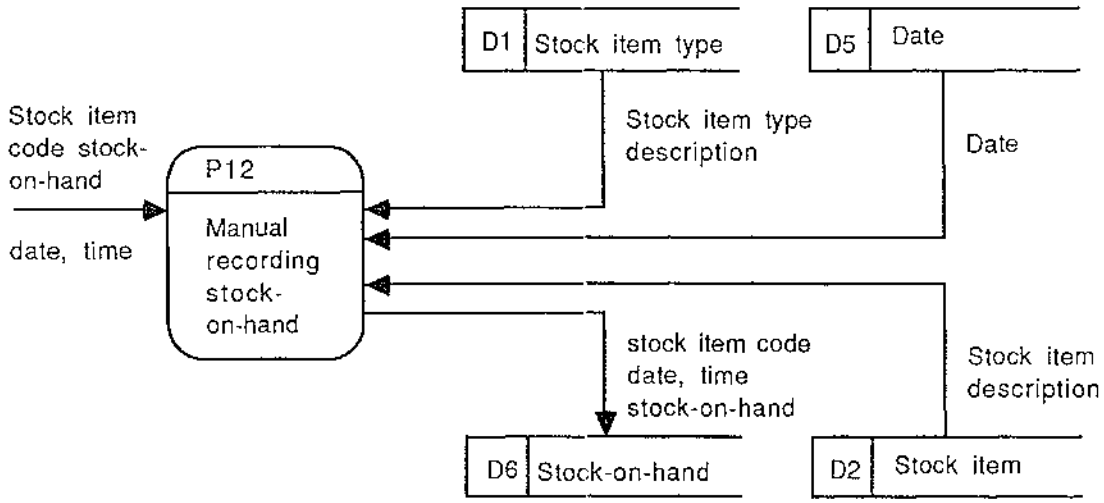
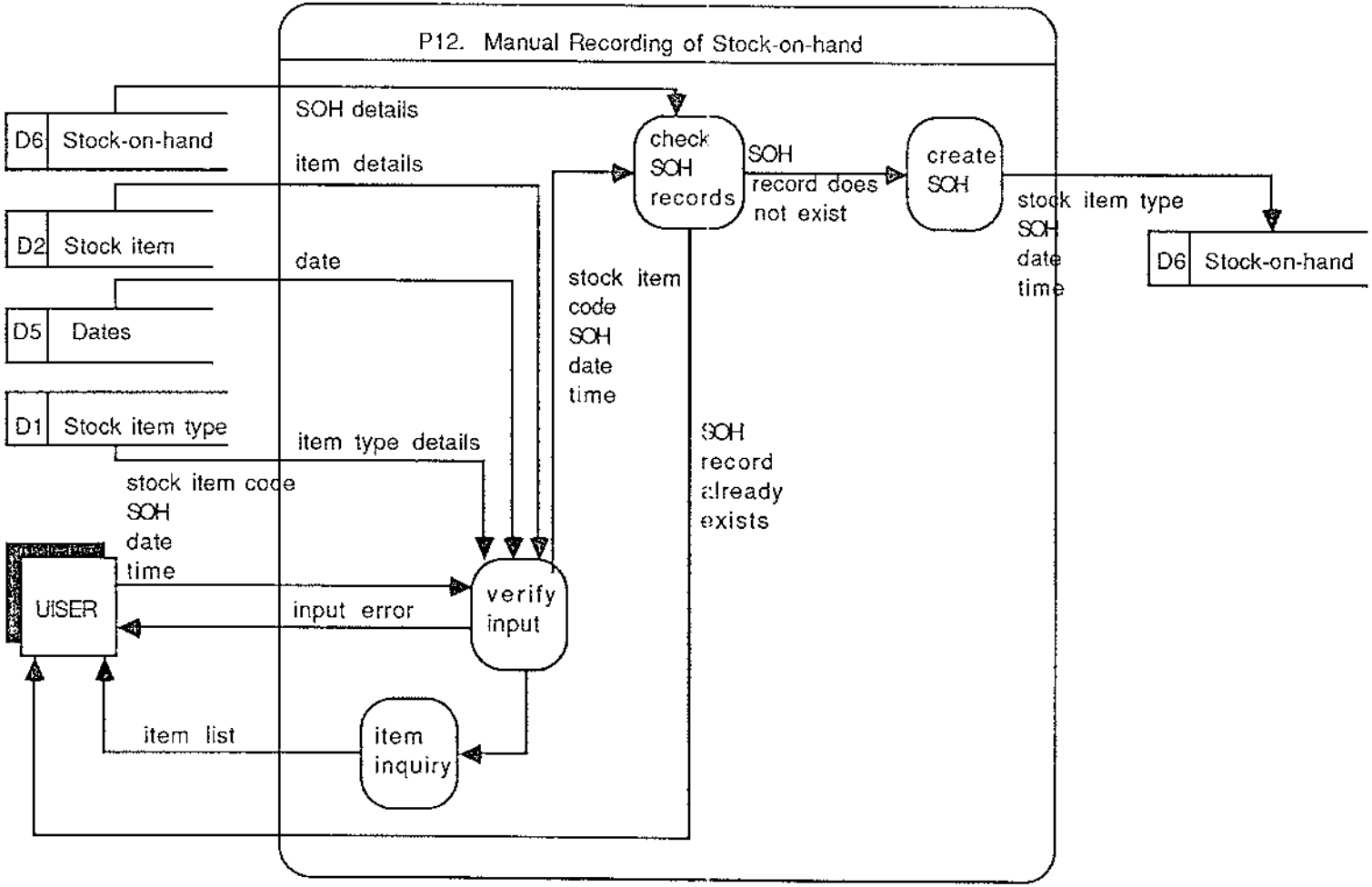


FIG. B.3 Monitoring Stock Levels (SOH) (DFD1)

FIG. B.3.1 Process P12



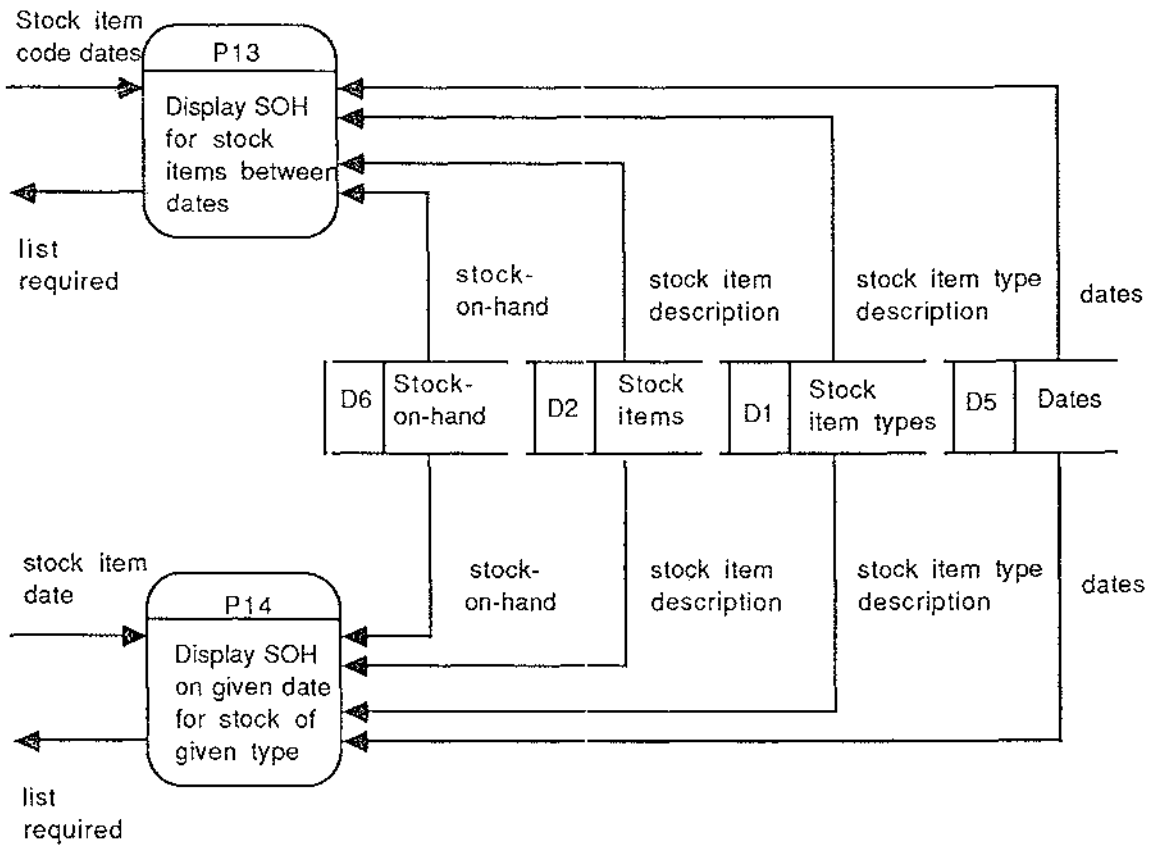


FIG. B.4 Monitoring Stock Levels (DFD1)

FIG. B.4.1 Process P13

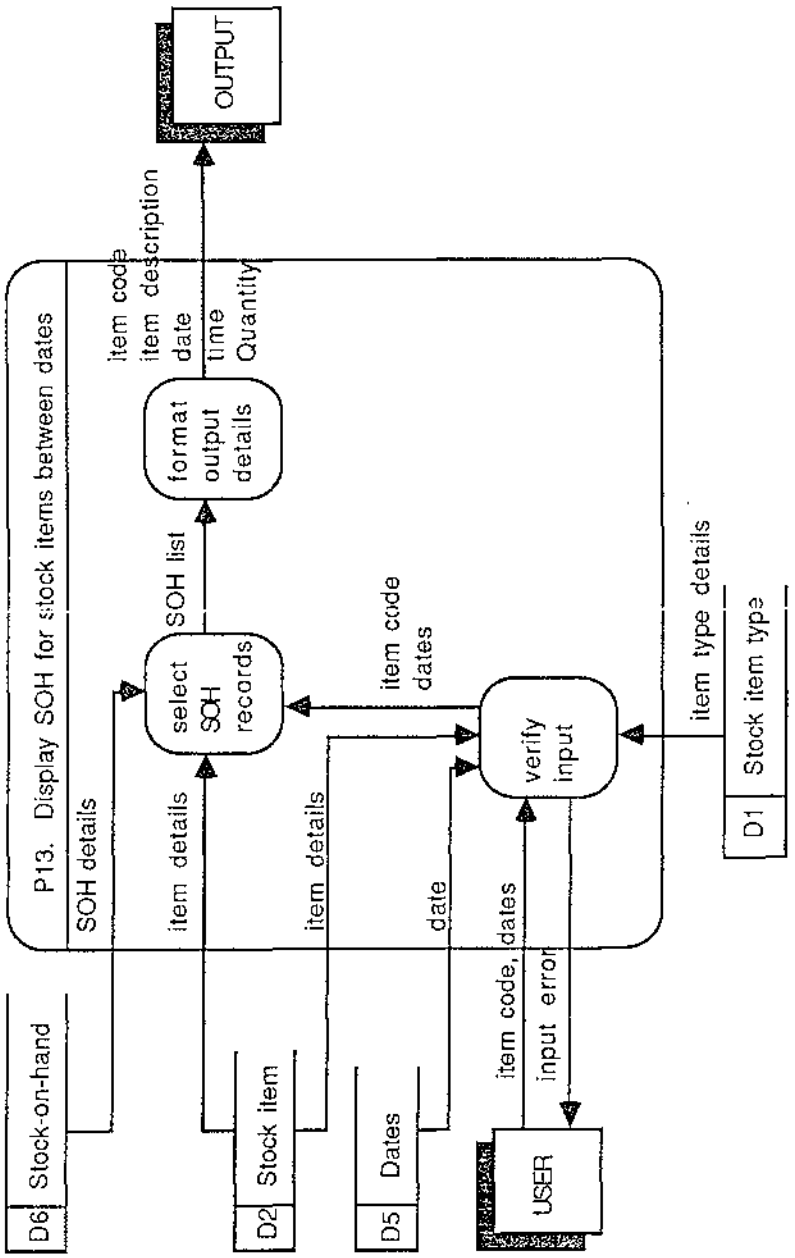
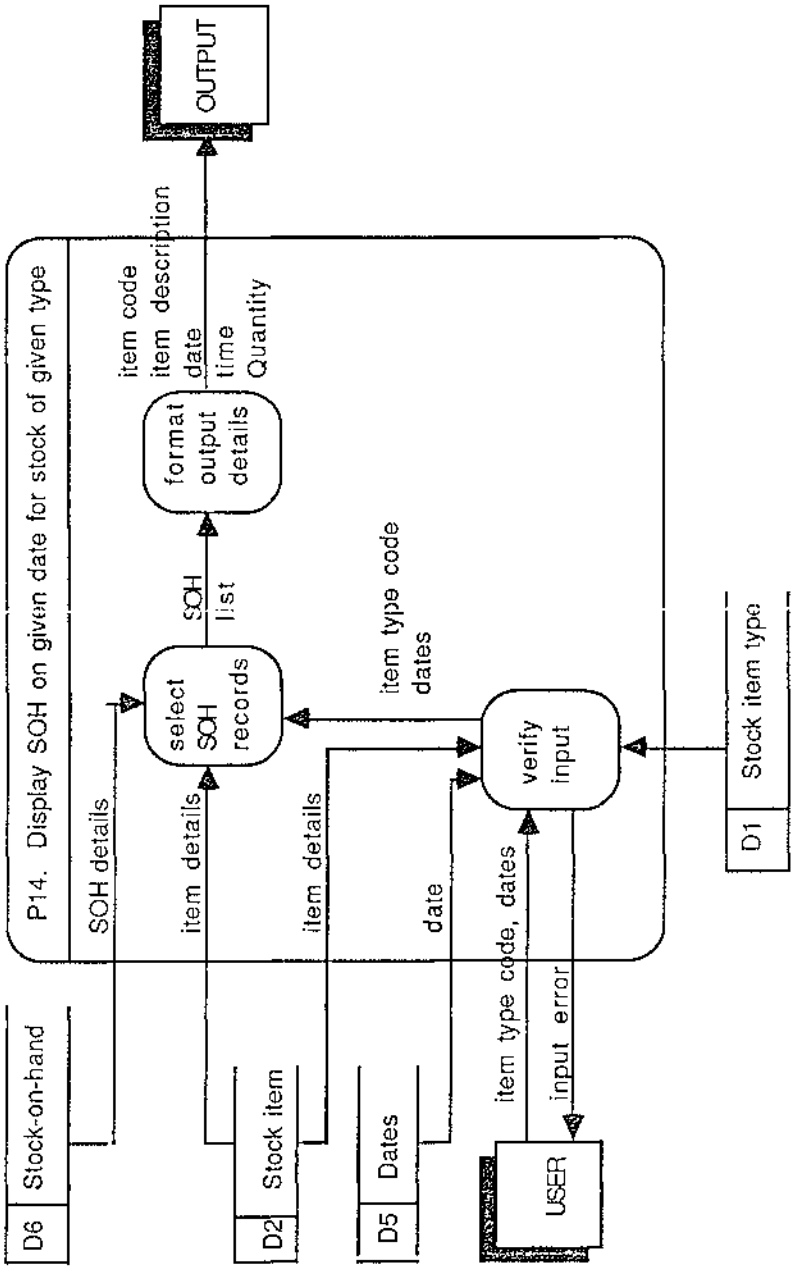


FIG. B.4.2 Process P14



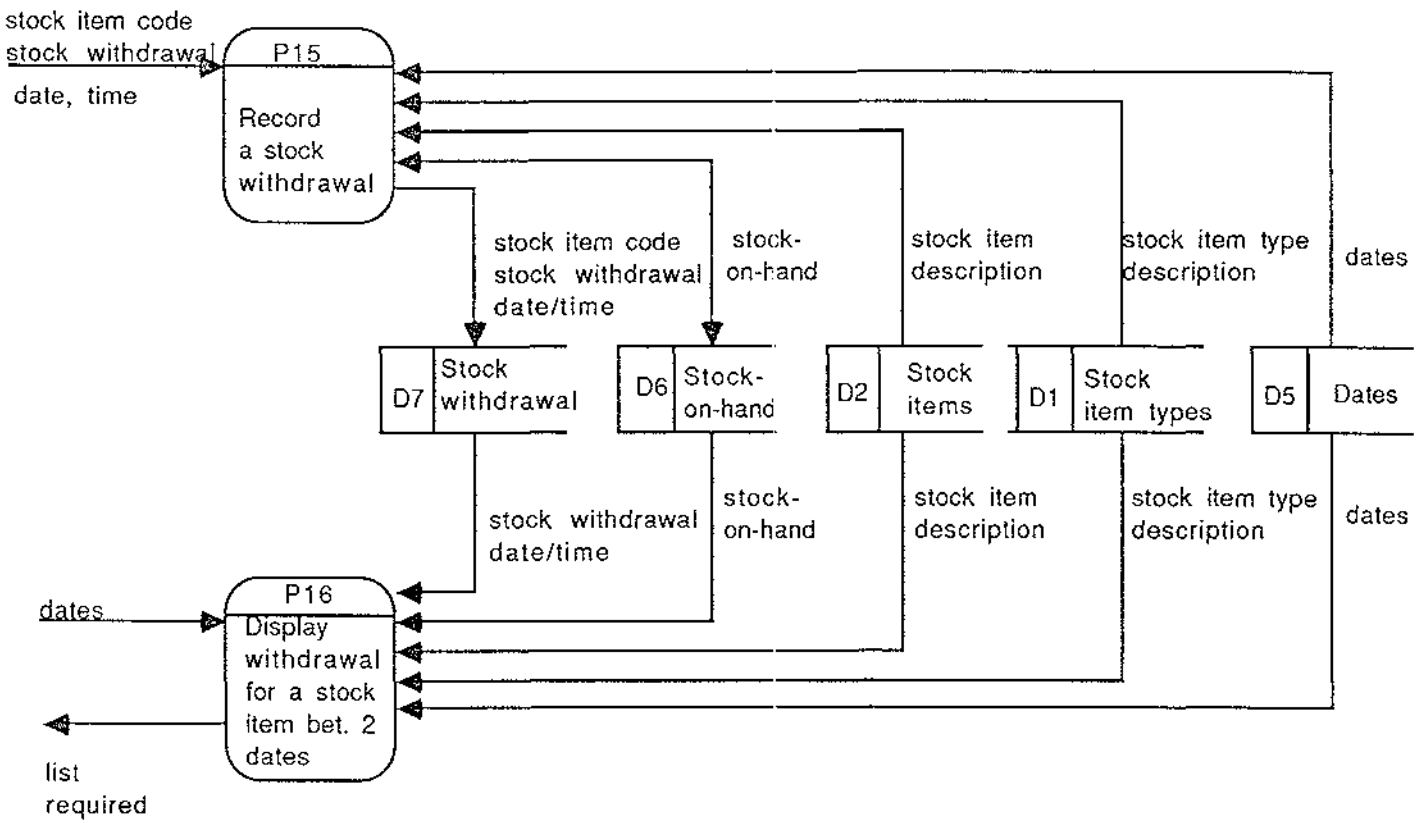


FIG. B.5 Monitoring Stock levels (DFD1)

FIG. B.5.1 Process P15

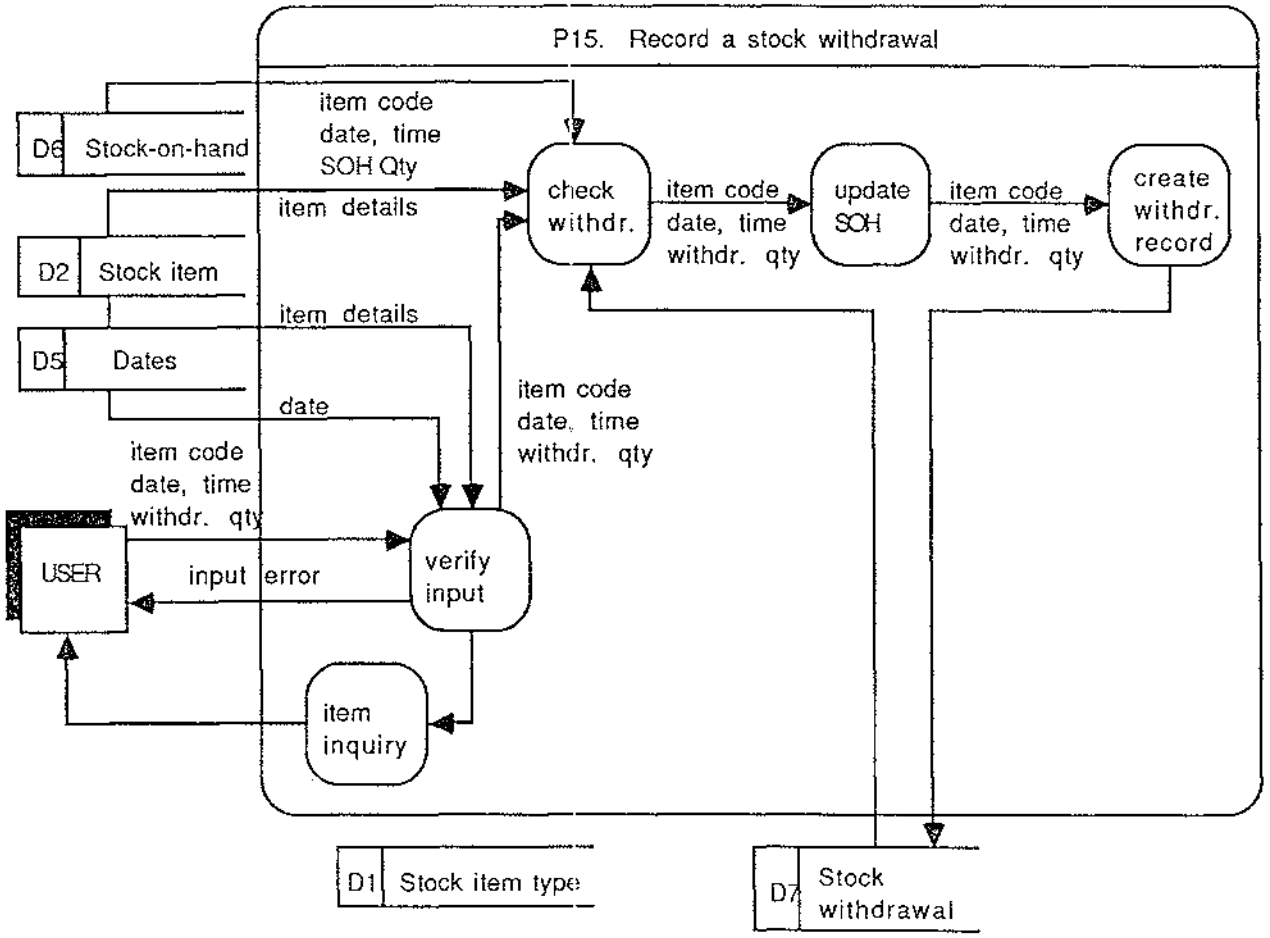
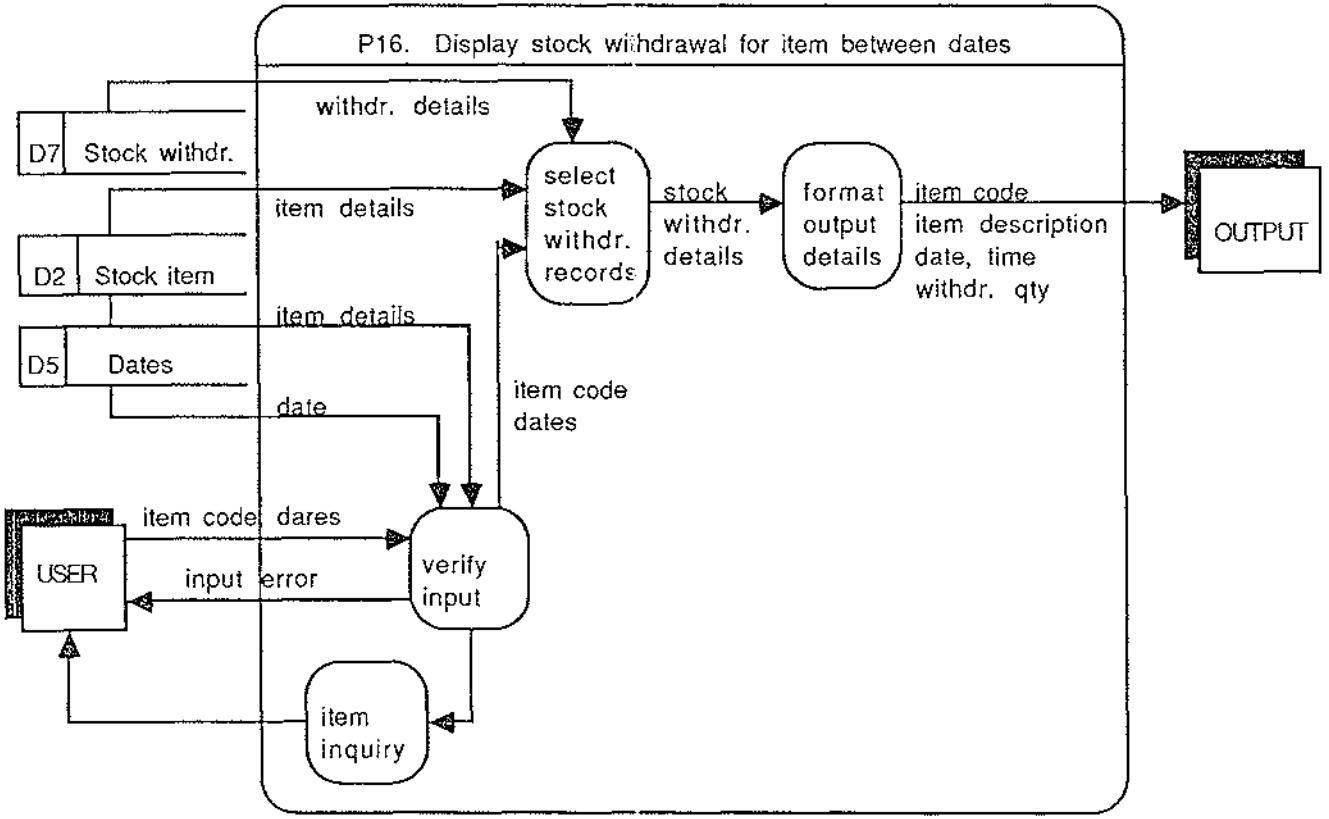


FIG. B.5.2 Process P16



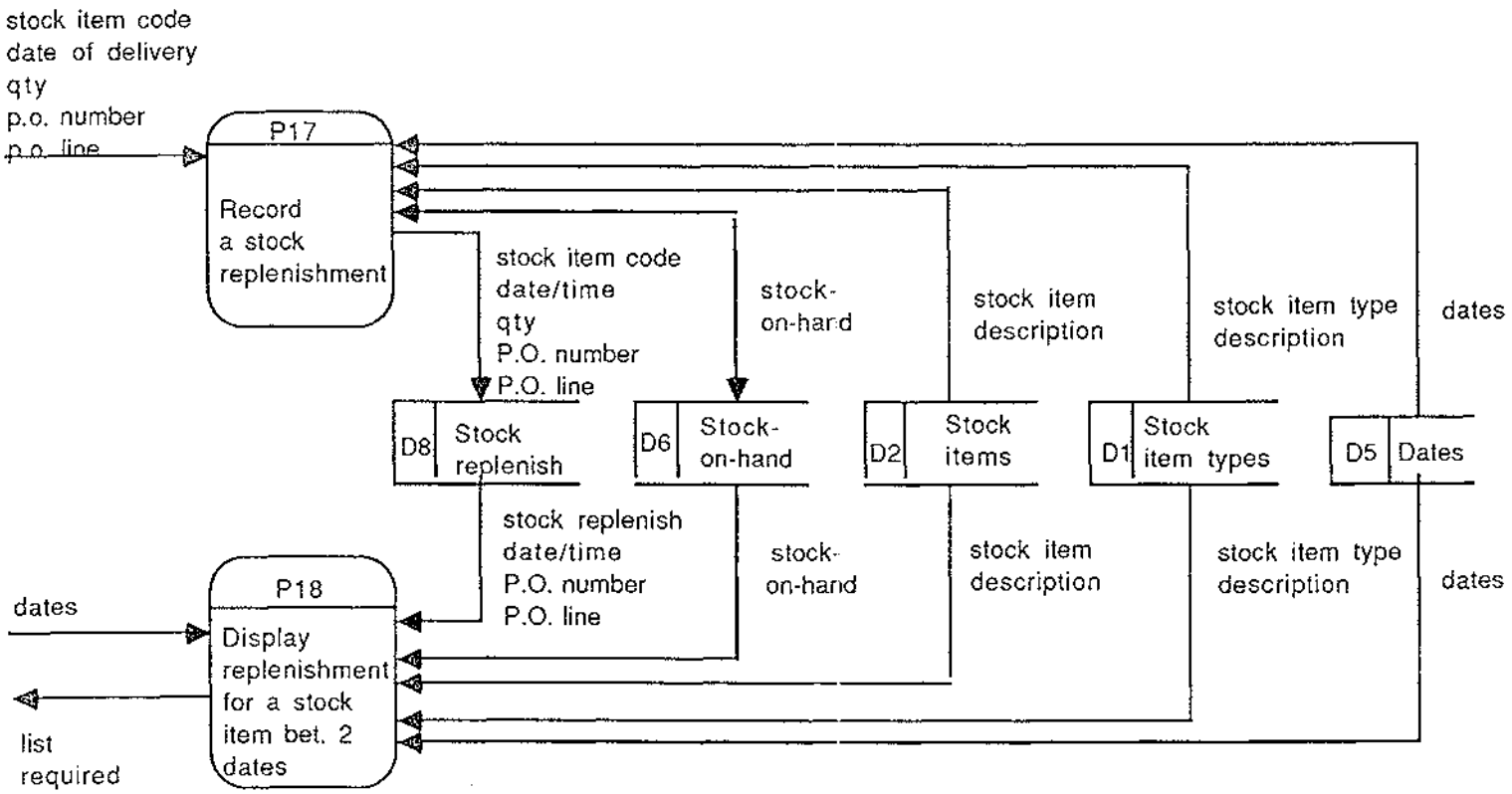


FIG. B.6 Monitoring Stock Levels (DFD1)

FIG. B.6.1 Process P17

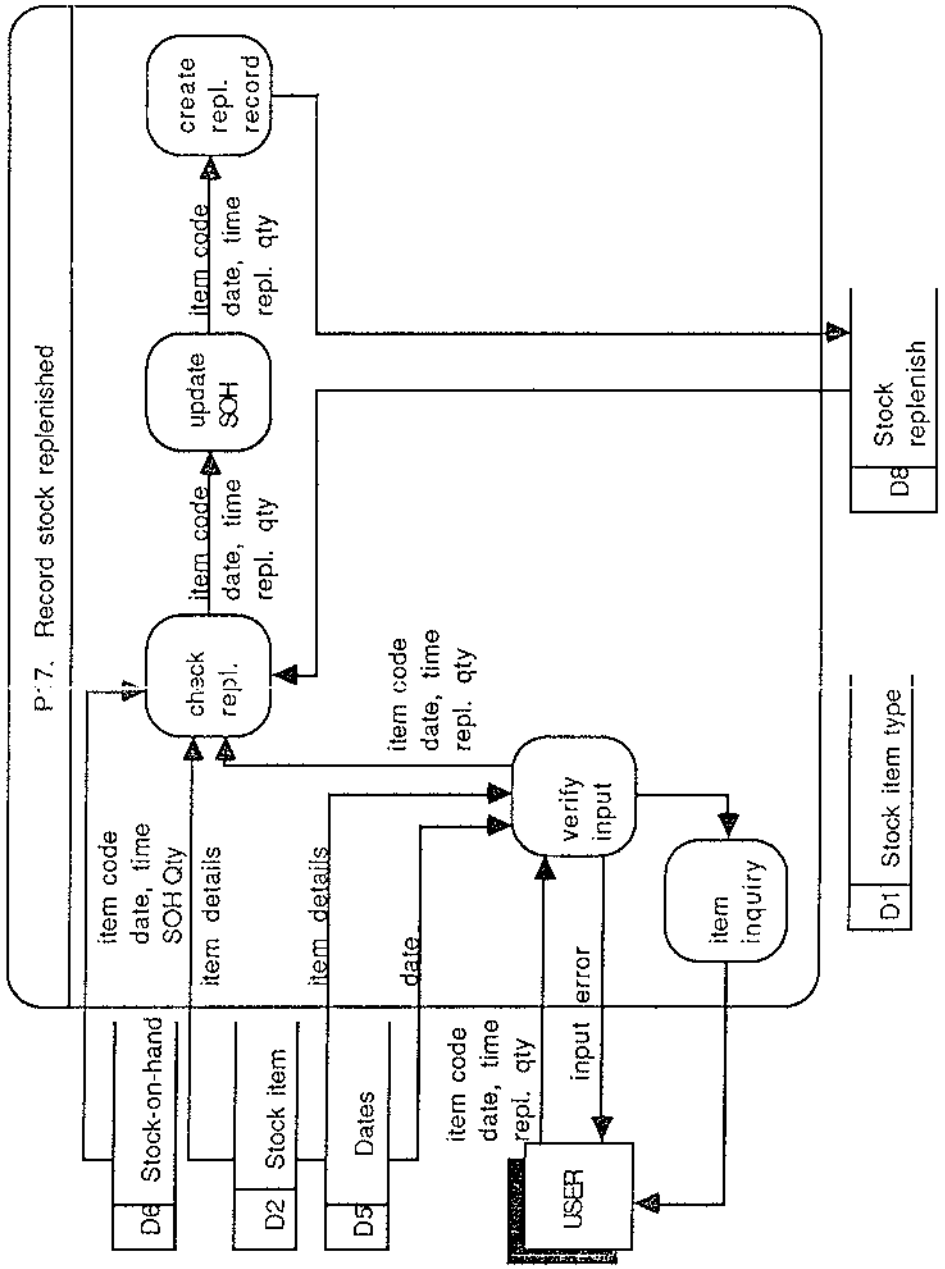


FIG. B.6.2 Process P18

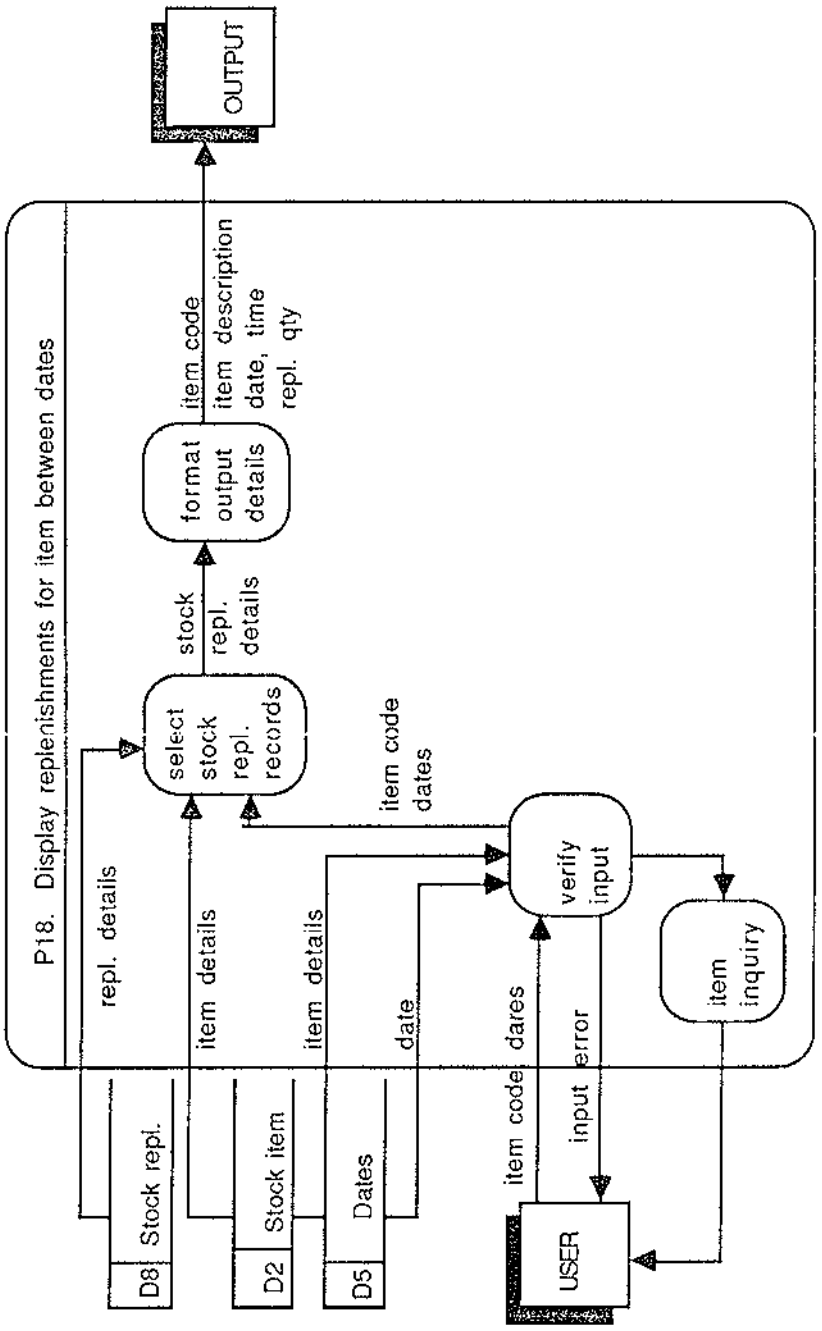
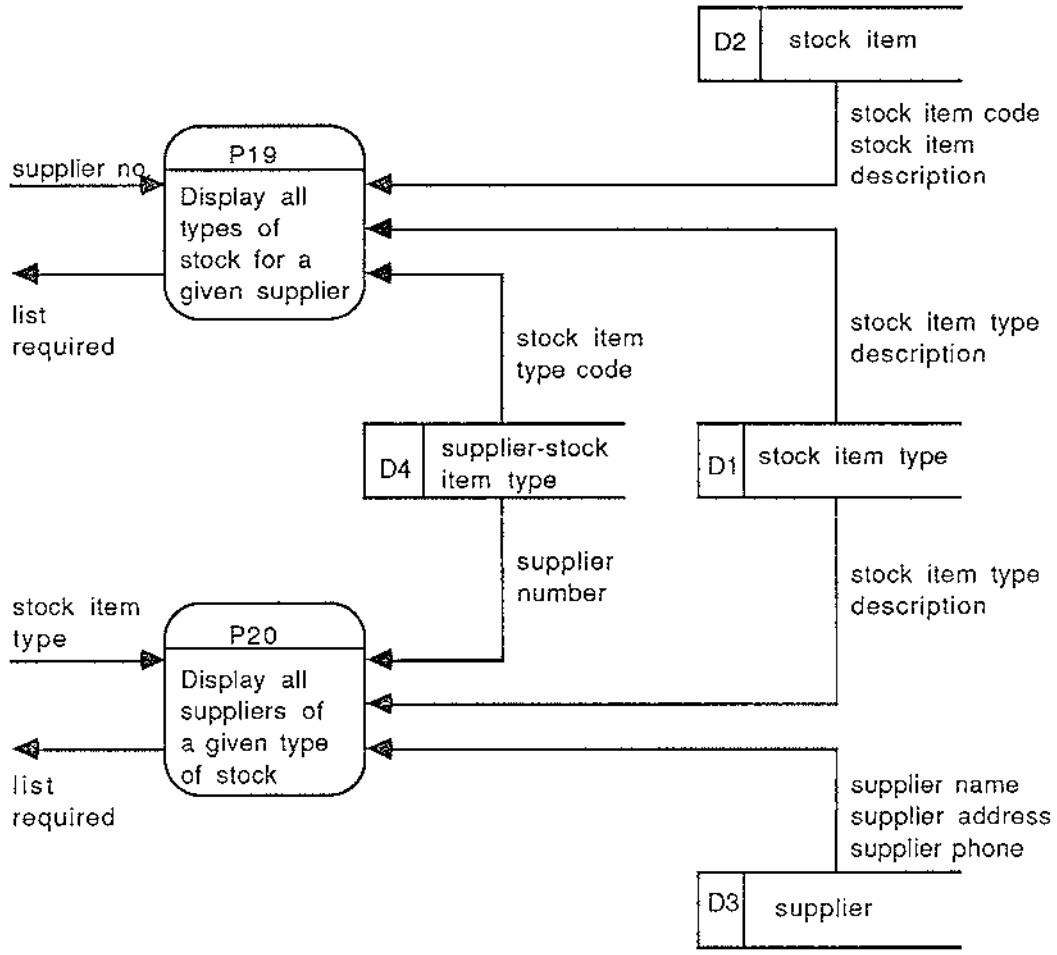
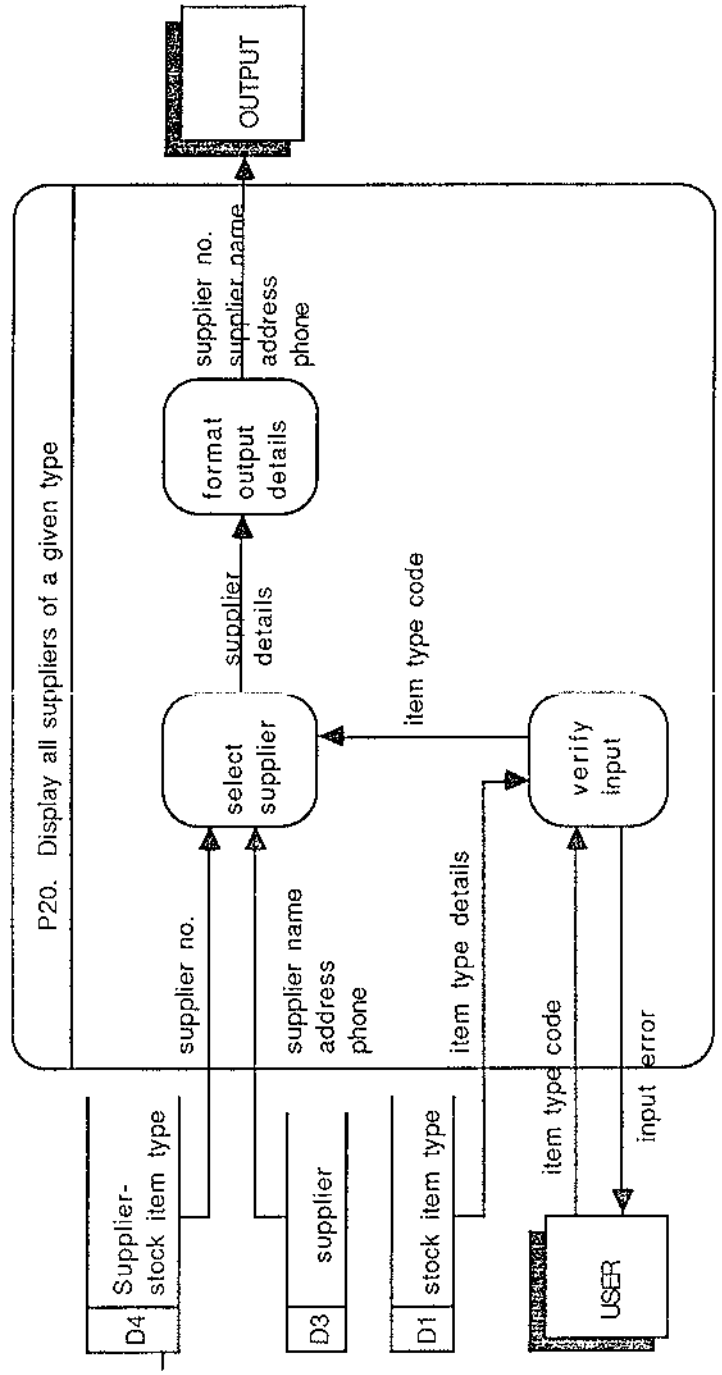


FIG. B.7 Purchase order processing and follow-up



FIG> B.7.1 Process P20



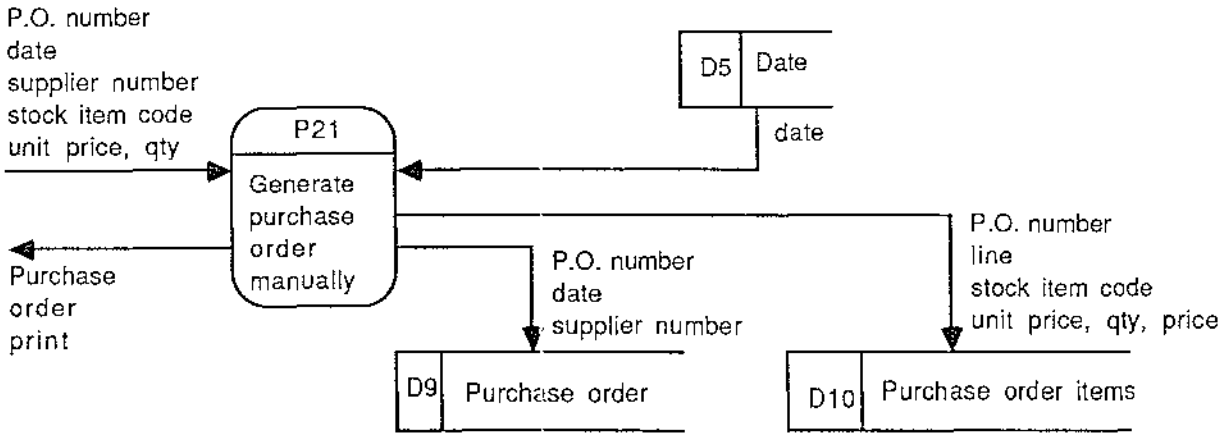


FIG. B.8 Purchase order processing and follow-up (DFD1)

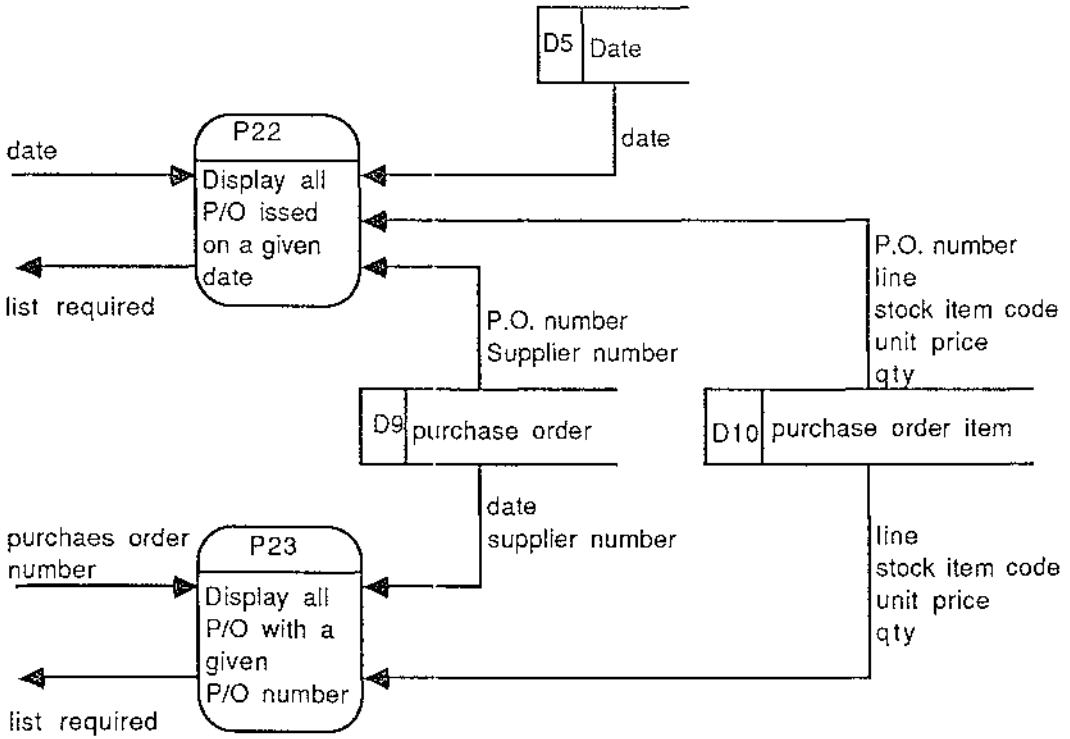


FIG. B.9 Purchase order processing and follow-up (DFD1)

Appendix C

Table C.1 List of Programs

No.	Program-ID	Description
1	ML1	Top level menu
2	ML2	Menu for system start-up
3	ML3	Menu for New types of stock
4	ML4	Menu for monitoring stock levels
5	ML5	Menu for purchase order processing and follow-up
6	MSL1	Manual recording of stock on hand
7	MSL2	Display stock on hand profile for a given stock item between given dates
8	MSL3	Display stock on hand on a given date for all stock items of a given type
9	MSL4	Record a stock withdrawal
10	MSL5	Display list of stock withdrawals for a given stock item between two dates
11	MSL6	Record stock replenished
12	MSL7	Display list of replenishments for a given stock item between two dates
13	NTS1	Modify an existing stock item
14	NTS2	Display list of stock items
15	NTS3	Modify an existing supplier
16	NTS4	Display list of suppliers
17	PO1	Display all suppliers of a given type of stock
18	SSU1	Add one or more stock item types
19	SSU2	Modify one or more existing stock item types
20	SSU3	Display available stock item types
21	SSU4	Add initial list of stock items
22	SSU5	Add initial list of suppliers
23	SSU6	Add list of stock item types carried by a supplier
24	SSU7	Enter sequence of dates for which records will be kept

Table C.2 Results of Function Point Analysis

Function Point Analysis		
Function Type	Function	Complexity Low/Average/High
External Input type	1. SSU1	Low
	2. SSU2	Low
	3. SSU4	Low
	4. SSU5	Low
	5. SSU6	Low
	6. SSU7	Low
	7. NTS1	Low
	8. NTS3	Low
	9. MSL1	Low
	10. MSL4	Low
	11. MSL6	Low
External output types	1. MSL2	Low
	2. MSL3	Low
	3. MSL5	Low
	4. MSL7	Low
	5. PO1	Low
	6. SSU3	Low
	7. NTS2	Low
	8. NTS4	Low
Logical Internal file type	1. Purchase order	Low
	2. Stock data	Low
	3. Supplier data	Low
	4. Date data	Low
External Inquiry type	1. ML2	Low
	2. ML3	Low
	3. ML4	Low
	4. ML5	Low
	5. INQ1	Low
	6. INQ2	Low
	7. INQ3	Low

Description	Complexity			Total
	Low	Average	High	
External Input	11 x 3 = 33	0 x 4 = 0	0 x 6 = 0	33
External Output	8 x 4 = 32	0 x 5 = 0	0 x 7 = 0	32
Logical Iternal File	4 x 7 = 28	0 x 10 = 0	0 x 15 = 0	28
External Interface File	0 x 5 = 0	0 x 7 = 0	0 x 10 = 0	0
External Inquiry	7 x 3 = 21	0 x 4 = 0	0 x 6 = 0	21
			Total Unadjusted Function Points	114

No.	Characteristic	DI	No.	Characteristic	DI
1	Data Communications	0	8	Online Update	4
2	Distributed Functions	0	9	Complex Processing	2
3	Performance	1	10	Reuseability	0
4	Heavily Used Configuration	1	11	Installation Ease	0
5	Transaction Rate	1	12	Operational Ease	1
6	Online Data Entry	3	13	Multiple Sites	0
7	End User Efficiency	3	14	Facilitate Change	4
				Total Degree of Influence	20

- o Not Present, or no influence = 0
- o Insignificant influence = 1
- o Moderate influence = 2
- o Average influence = 3
- o Significant influence = 4
- o Strong influence, throughout = 5

Adjustment factor = 0.65 + 0.01 x 20
= 0.85

Function Point Count = Total Unadjusted Function Points x Adjustment factor
= 114 x 0.85
= 96.9 FP

Table C.3 Line count for System 1

SYSTEM 1	(LOC)		
program-id	AREV	INFORMIX	COBOL
SSU1	48	69	143
SSU2	59	171	343
SSU4	82	84	197
SSU5	81	85	288
SSU6	68	100	225
SSU7	55	66	152
NTS1	92	174	427
NTS3	92	182	408
MSL1	75	115	232
MSL4	116	116	282
MSL6	111	124	278
MSL2	92	104	307
MSL3	84	104	303
MSL5	81	105	307
MSL7	81	110	308
PO1	55	71	264
SSU3	4	60	148
NTS2	5	68	172
NTS4	5	74	174
ML1	7	23	86
ML2	10	31	96
ML3	7	20	86
ML4	10	33	94
ML5	8	27	86

Table C.4 Line count for System 2

SYSTEM 2	(LOC)		
program-id	AREV	INFORMIX	COBOL
SSU1	48	69	143
SSU2	51	101	217
SSU4	82	84	197
SSU5	81	85	174
SSU6	68	100	225
SSU7	55	66	152
NTS1	82	102	280
NTS3	83	108	262
MSL1	75	115	232
MSL4	116	116	282
MSL6	111	124	278
MSL2	92	104	307
MSL3	84	104	303
MSL5	81	105	307
MSL7	81	110	308
PO1	55	71	264
SSU3	4	60	148
NTS2	5	68	172
NTS4	5	74	174
ML1	7	23	86
ML2	10	31	96
ML3	7	20	86
ML4	10	33	94
ML5	8	27	86
INQ1	8	70	126
INQ2	10	72	147
INQ3	9	74	146

Table C.5 Line count for Input/update module - System 1

INPUT/UPDATE 1	(LOC)		
program-id	AREV	INFORMIX	COBOL
SSU1	48	69	143
SSU2	59	171	343
SSU4	82	84	197
SSU5	81	85	288
SSU6	68	100	225
SSU7	55	66	152
NTS1	92	174	427
NTS3	92	182	408
MSL1	75	115	232
MSL4	116	116	282
MSL6	111	124	278

Table C.6 Line count for Input/update - System 2

INPUT/UPDATE 2	(LOC)		
program-id	AREV	INFORMIX	COBOL
SSU1	48	69	143
SSU2	51	101	217
SSU4	82	84	197
SSU5	81	85	174
SSU6	68	100	225
SSU7	55	66	152
NTS1	82	102	280
NTS3	83	108	262
MSL1	75	115	232
MSL4	116	116	282
MSL6	111	124	278

Table C.7 Line count by Module Type Categories

MODULE TYPE	(LOC)		
	AREV	INFORMIX	COBOL
INPUT/UPDATE	879	1286	2975
REPORT	407	696	1983
MENUS	42	134	448
INQUIRY	27	216	419
FORMS	62	187	258

Table C.8 Line count for Reports Module

REPORT	(LOC)		
program-id	AREV	INFORMIX	COBOL
MSL2	92	104	307
MSL3	84	104	303
MSL5	81	105	307
MSL7	81	110	308
PO1	55	71	264
SSU3	4	60	148
NTS2	5	68	172
NTS4	5	74	174

Table C.9 Line count for Menus Module

MENUS	(LOC)		
program-id	AREV	INFORMIX	COBOL
ML1	7	23	86
ML2	10	31	96
ML3	7	20	86
ML4	10	33	94
ML5	8	27	86

Table C.10 Line count for Inquiry Module

INQUIRY	(LOC)		
program-id	AREV	INFORMIX	COBOL
INQ1	8	70	126
INQ2	10	72	147
INQ3	9	74	146

Table C.11 Line count for Forms Module

FORMS	(LOC)		
	AREV	INFORMIX	COBOL
ITEMTYPE.ENTRY	5	19	21
ITEM.ENTRY	7	23	23
SUPPLIER.ENTRY	9	28	50
ITEMTYPE-SUPP.ENTRY	6	22	29
STOCKONHAND.ENTRY	9	24	36
WITHDRAWAL.ENTRY	9	25	38
REPLENISH.ENTRY	11	29	38
DATE.ENTRY	6	17	23

Appendix D

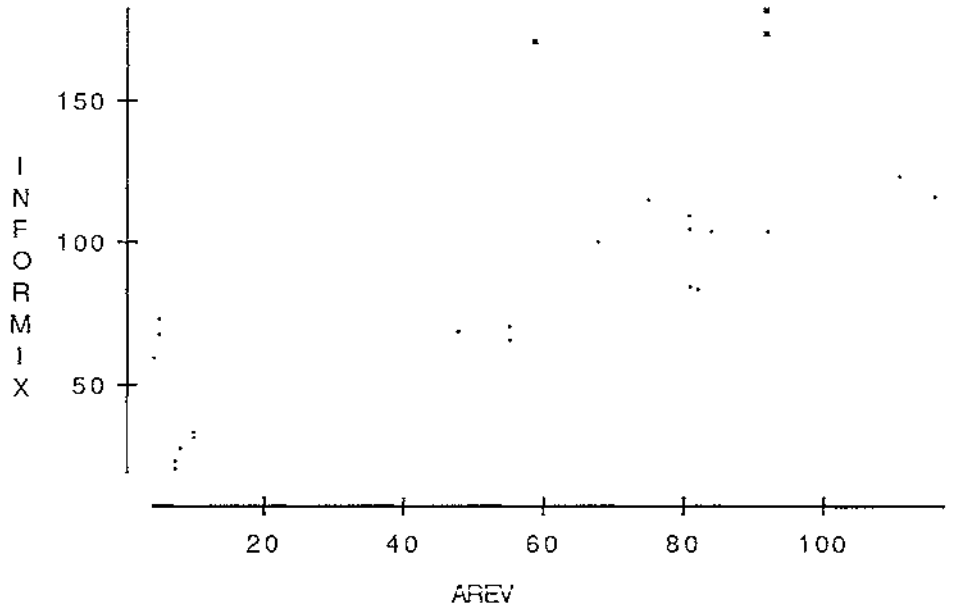


Fig. D.1 Plot of Informix (LOC) vs AREV (LOC) - System 1

Dependent variable is: INFORMIX
R² = 58.0% R²(adjusted) = 56.1%
s = 30.51 with 24 - 2 = 22 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	28267.4	1	28267	30.4
Residual	20474.0	22	930.635	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	37.4602	11.11	3.37
AREV	0.916383	0.1663	5.51

Table D.1 Regression analysis of plot in Fig. D.1.

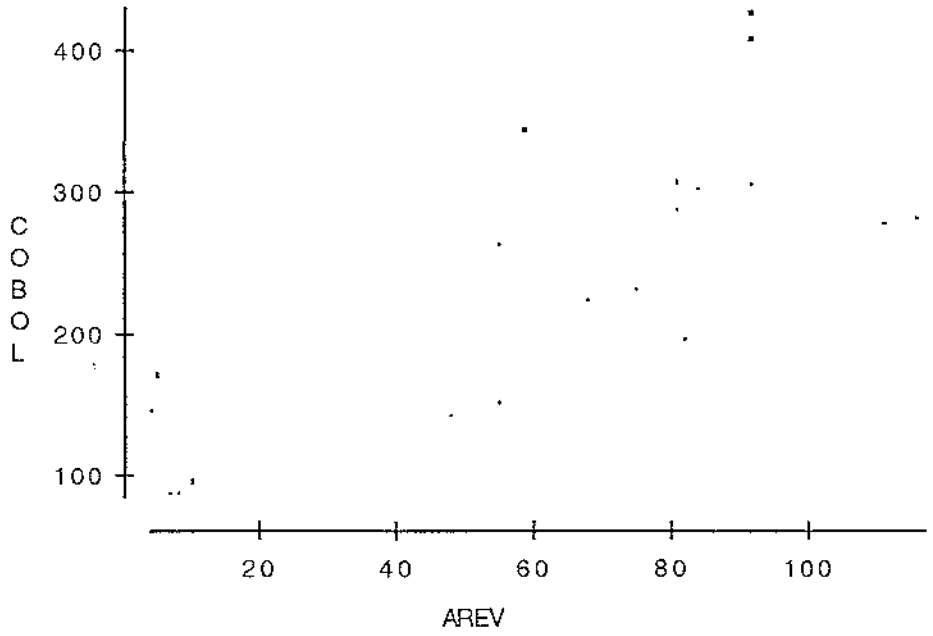


Fig. D.2 Plot of Cobol (LOC) vs AREV (LOC) - System 1

Dependent variable is: COBOL
R² = 65.5% R²(adjusted) = 63.9%
s = 61.67 with 24 - 2 = 22 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	158512	1	2e+5	41.7
Residual	83662.7	22	3802.85	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	105.175	22.46	4.68
AREV	2.17003	0.3361	6.46

Table D.2 Regression analysis of plot in Fig. D.2.

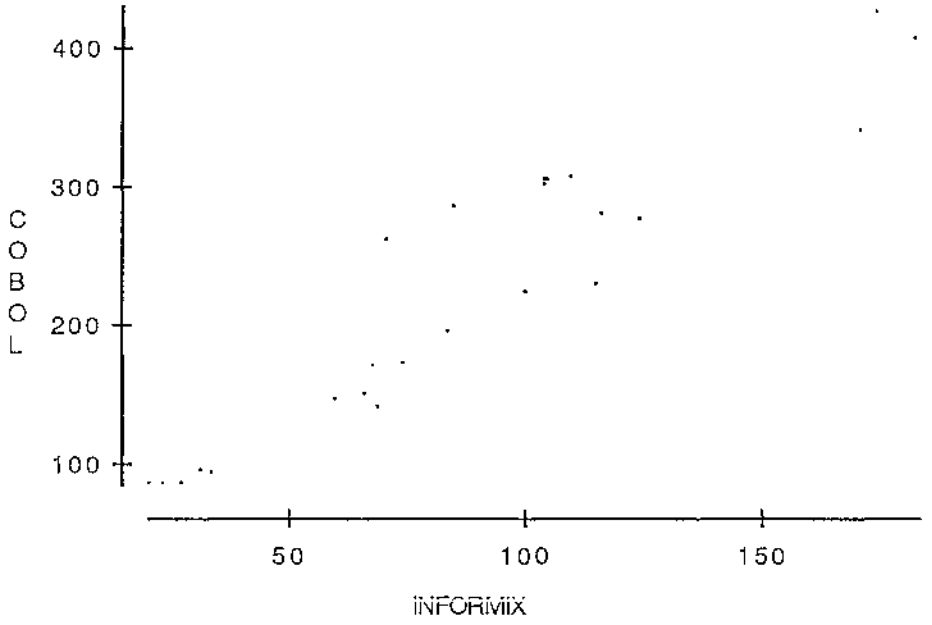


Fig. D.3 Plot of Cobol (LOC) vs Informix (LOC) - System 1.

Dependent variable is: COBOL
R² = 87.7% R²(adjusted) = 87.1%
s = 36.87 with 24 - 2 = 22 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	212275	1	2e+5	156
Residual	29899.8	22	1359.08	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	41.2555	16.53	2.50
INFORMIX	2.08689	0.1670	12.5

Table D.3 Regression Analysis of plot in Fig. D.3.

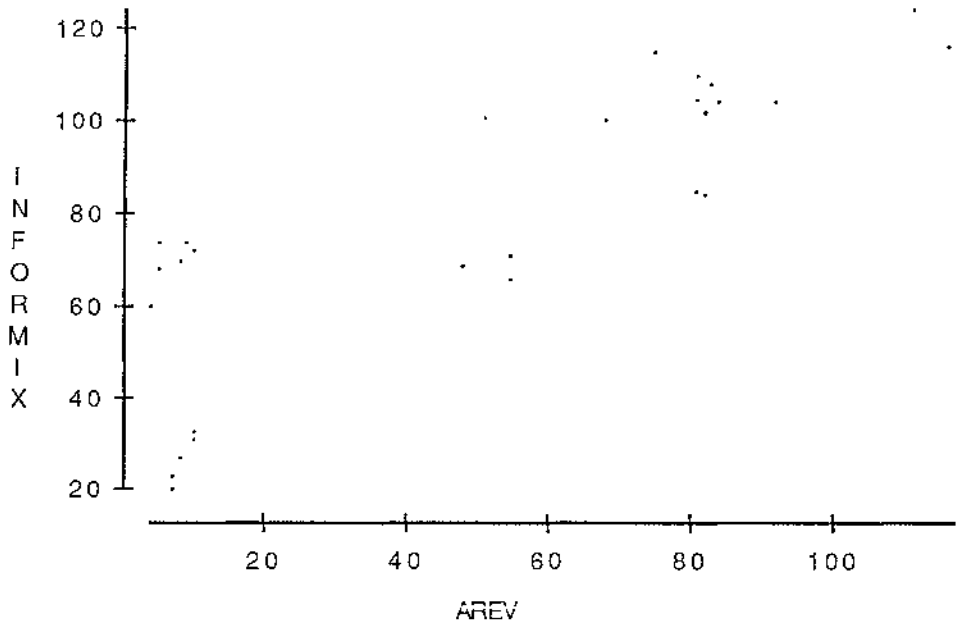


Fig. D.4 Plot of Informix (LOC) vs AREV (LOC) - System 2

Dependent variable is: INFORMIX
R² = 69.3% R²(adjusted) = 68.1%
s = 17.40 with 27 - 2 = 25 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	17087.3	1	17087	56.4
Residual	7570.96	25	302.838	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	45.3343	5.528	8.20
AREV	0.671668	0.0894	7.51

TABLE D.4 Regression analysis of plot in Fig. D.4

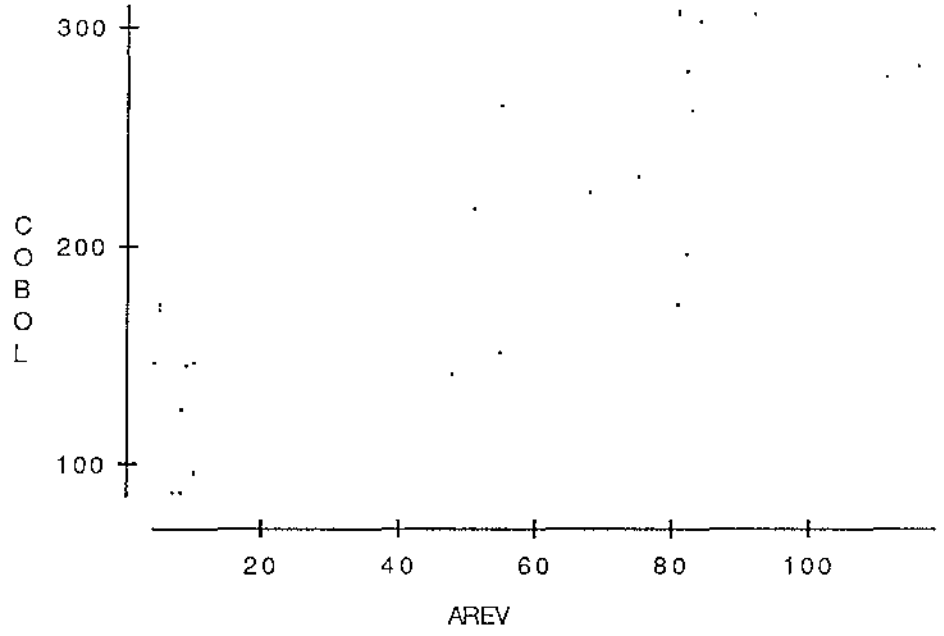


Fig. D.5 Plot of Cobol (LOC) vs AREV (LOC) - System 2.

Dependent variable is: COBOL
R² = 71.7% R²(adjusted) = 70.5%
s = 42.12 with 27 - 2 = 25 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	112112	1	1e+5	63.2
Residual	44356.4	25	1774.26	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	111.379	13.38	8.32
AREV	1.72045	0.2164	7.95

Table D.5 Regression analysis of plot in Fig. D.5.

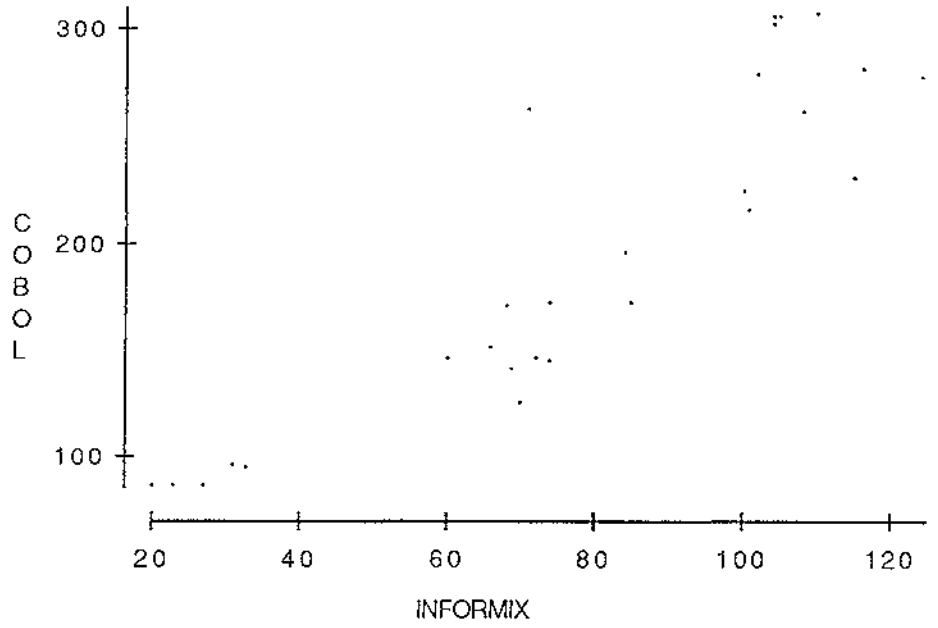


Fig. D.6 Plot of Cobol (LOC) vs Informix (LOC) - System 2.

Dependent variable is: COBOL
R¤ = 80.4% R¤(adjusted) = 79.6%
s = 35.03 with 27 - 2 = 25 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	125797	1	1e+5	103
Residual	30671.3	25	1226.85	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	18.9870	18.74	1.01
INFORMIX	2.25867	0.2231	10.1

Table D.6 Regression analysis of plot in Fig. D.6.

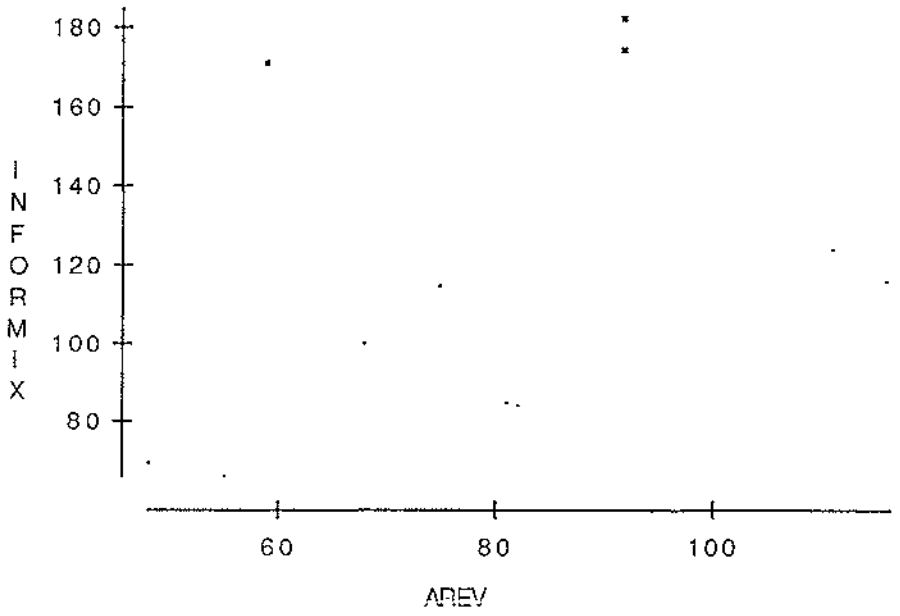


Fig. D.7 Plot of Informix (LOC) vs AREV (LOC)
Input/update modules - System 1

Dependent variable is: INFORMIX
R² = 13.8% R²(adjusted) = 4.3%
s = 41.22 with 11 - 2 = 9 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	2457.78	1	2458	1.45
Residual	15293.1	9	1699.24	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	59.7817	49.10	1.22
AREV	0.714904	0.5944	1.20

Table D.7 Regression analysis of plot in Fig. D.7.

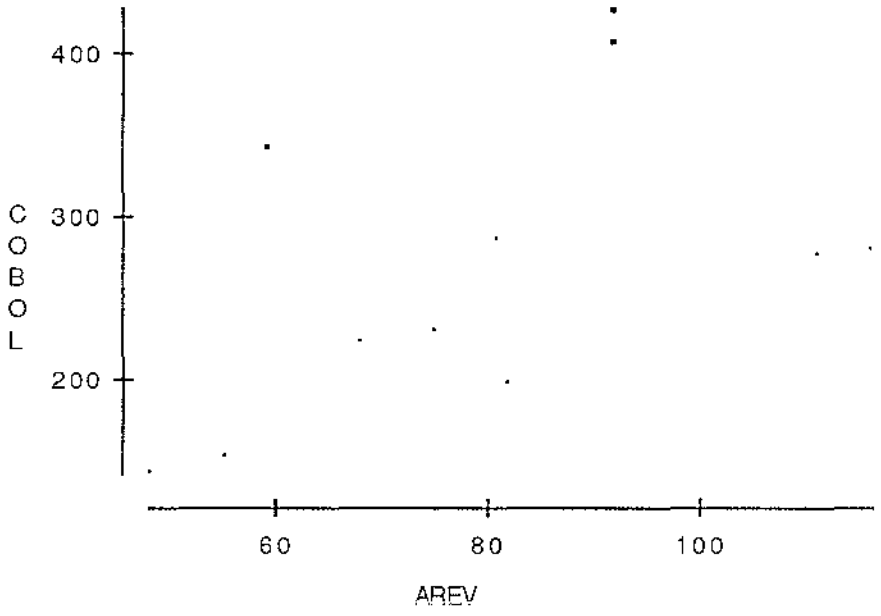


Fig. D.8 Plot of Cobol (LOC) vs AREV (LOC)
Input/update modules - System 1

Dependent variable is: COBOL
R² = 25.0% R²(adjusted) = 16.6%
s = 85.85 with 11 - 2 = 9 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	22075.9	1	22076	3.00
Residual	66326.8	9	7369.64	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	99.2432	102.3	0.971
AREV	2.14258	1.238	1.73

Table D.8 Regression analysis of plot in Fig. D.8.

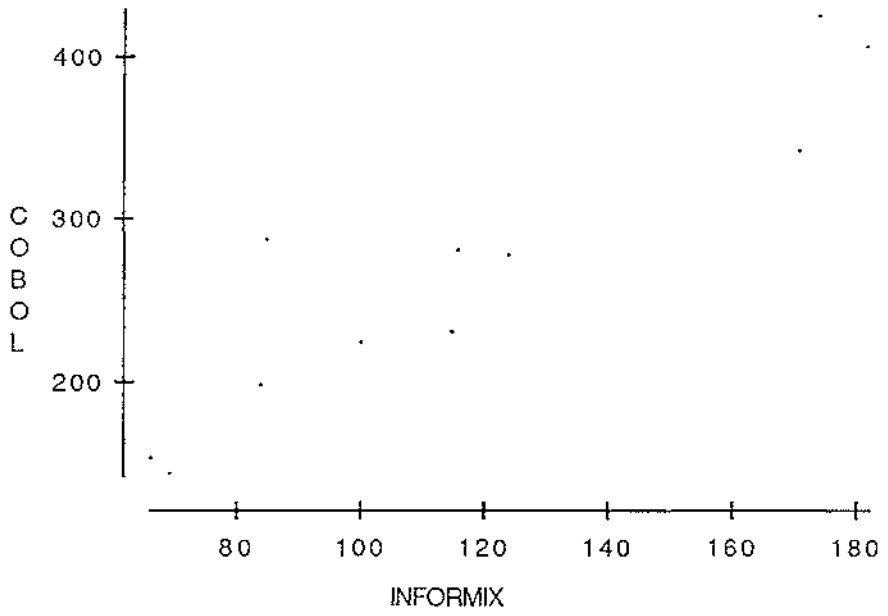


Fig. D.9 Plot of Cobol (LOC) vs Informix (LOC) Input/update modules - System 1

Dependent variable is: COBOL
R² = 85.8% R²(adjusted) = 84.2%
s = 37.35 with 11 - 2 = 9 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	75850.2	1	75850	54.4
Residual	12552.5	9	1394.73	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	28.7881	34.65	0.831
INFORMIX	2.06713	0.2803	7.37

Table D.9 Regression analysis of plot in Fig. D.9.

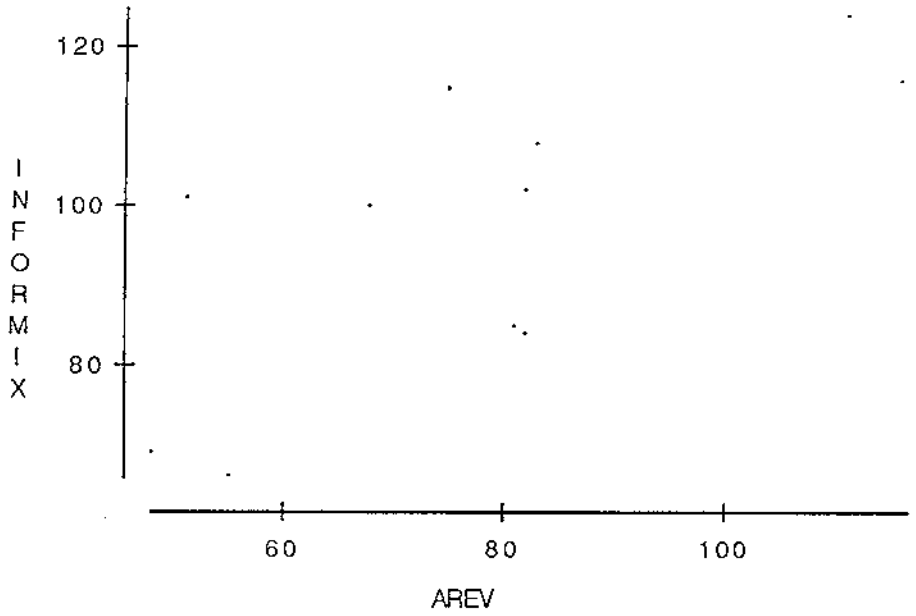


Fig. D.10 Plot of Informix (LOC) vs AREV (LOC)
Input/update modules - System 2

Dependent variable is: INFORMIX
R¤ = 49.6% R¤(adjusted) = 44.0%
s = 14.29 with 11 - 2 = 9 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	1805.00	1	1805	8.84
Residual	1837.18	9	204.132	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	50.0832	16.44	3.05
AREV	0.609254	0.2049	2.97

Table D.10 Regression analysis of plot in Fig. D.10.

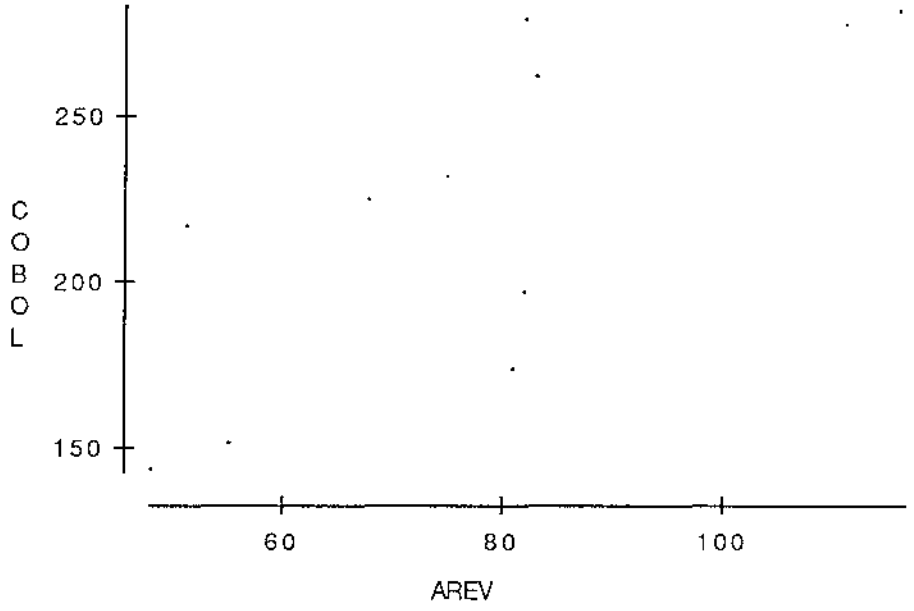


Fig. D.11 Plot of Cobol (LOC) vs AREV (LOC)
Input/update - System 2

Dependent variable is: COBOL
R¤ = 55.6% R¤(adjusted) = 50.7%
s = 35.74 with 11 - 2 = 9 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	14410.4	1	14410	11.3
Residual	11493.6	9	1277.07	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	88.6649	41.13	2.16
AREV	1.72146	0.5125	3.36

Table D.11 Regression analysis of plot in Fig. D.11.

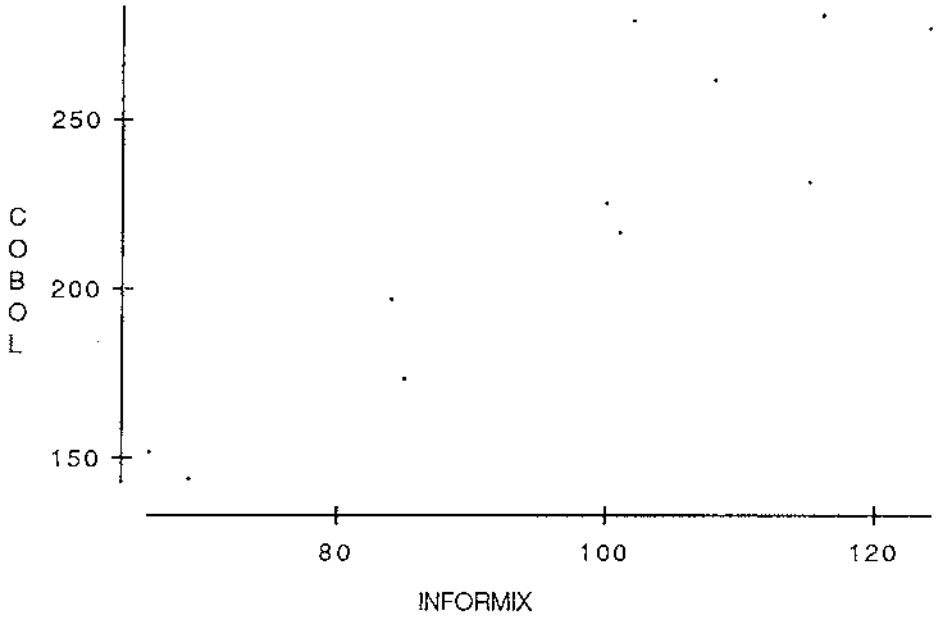


Fig. D.12 Plot of Cobol (LOC) vs Informix (LOC)
Input/update modules - System 2

Dependent variable is: COBOL
R¤ = 82.7% R¤(adjusted) = 80.8%
s = 22.30 with 11 - 2 = 9 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	21426.6	1	21427	43.1
Residual	4477.40	9	497.489	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	-13.9320	36.57	-0.381
INFORMIX	2.42547	0.3696	6.56

Table D.12 Regression analysis of plot in Fig. D.12.

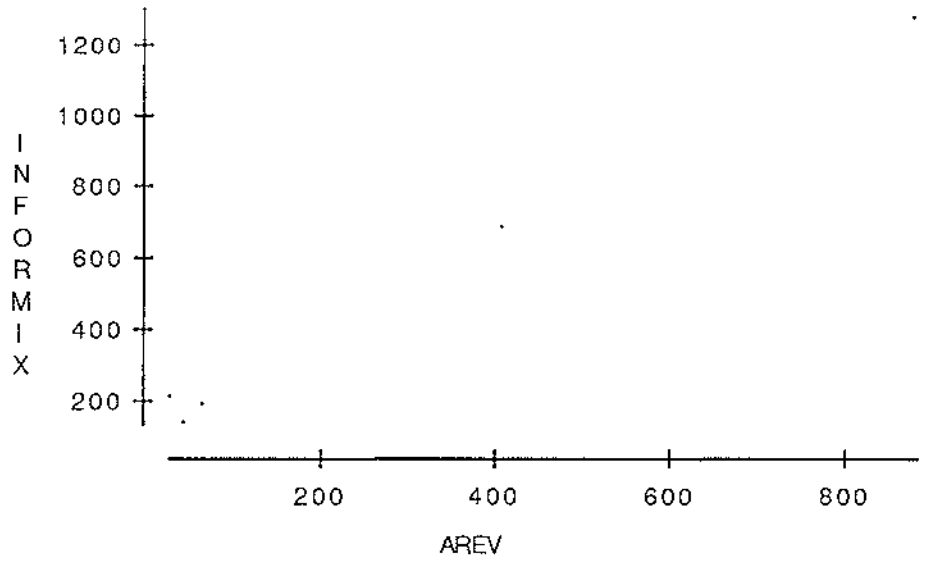


Fig. D.13 Plot of Informix (LOC) vs AREV (LOC)
Module types - System 2

Dependent variable is: INFORMIX
R¤ = 99.3% R¤(adjusted) = 99.1%
s = 46.83 with 5 - 2 = 3 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	962143	1	1e+6	439
Residual	6577.75	3	2192.58	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	126.575	27.62	4.58
AREV	1.33107	0.0635	20.9

Table D.13 Regression analysis of plot in Fig. D.13.

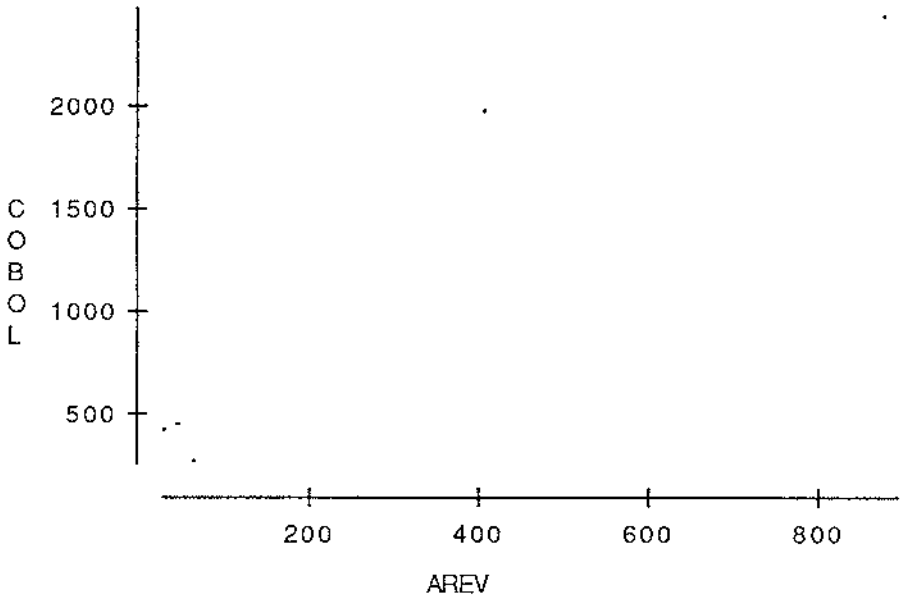


Fig. D.14 Plot of Cobol (LOC) vs AREV (LOC)
Module types - System 2

Dependent variable is: COBOL
R² = 89.7% R²(adjusted) = 86.3%
s = 378.1 with 5 - 2 = 3 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	3749038	1	4e+6	26.2
Residual	428944	3	142981	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	365.371	223.0	1.64
AREV	2.62749	0.5131	5.12

Table D.14 Regression analysis of plot in Fig. D.14.

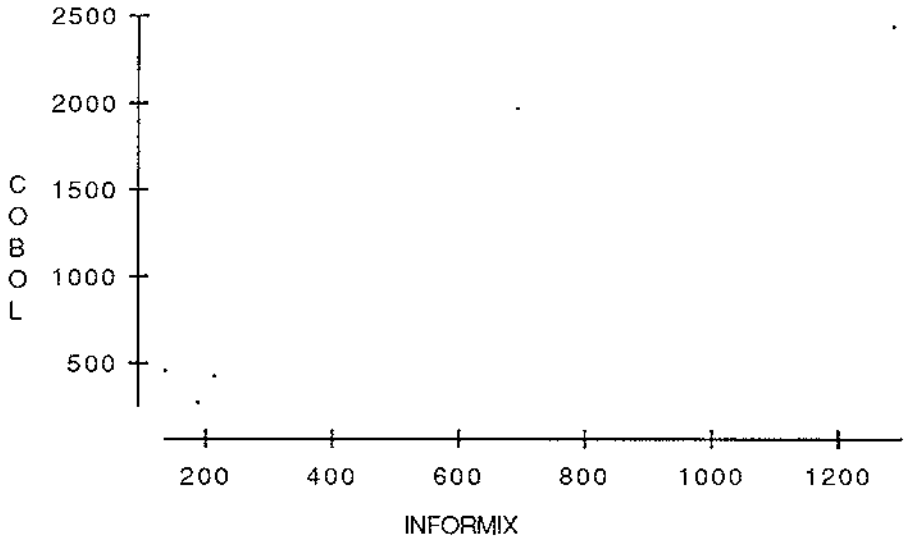


Fig. D.15 Plot of Cobol (LOC) vs Informix (LOC)
Module types - System 2

Dependent variable is: COBOL
R¤ = 91.4% R¤(adjusted) = 88.5%
s = 346.2 with 5 - 2 = 3 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	3818412	1	4e+6	31.9
Residual	359570	3	119857	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	109.769	235.3	0.466
INFORMIX	1.98537	0.3517	5.64

Table D.15 Regression analysis of plot in Fig. D.15.

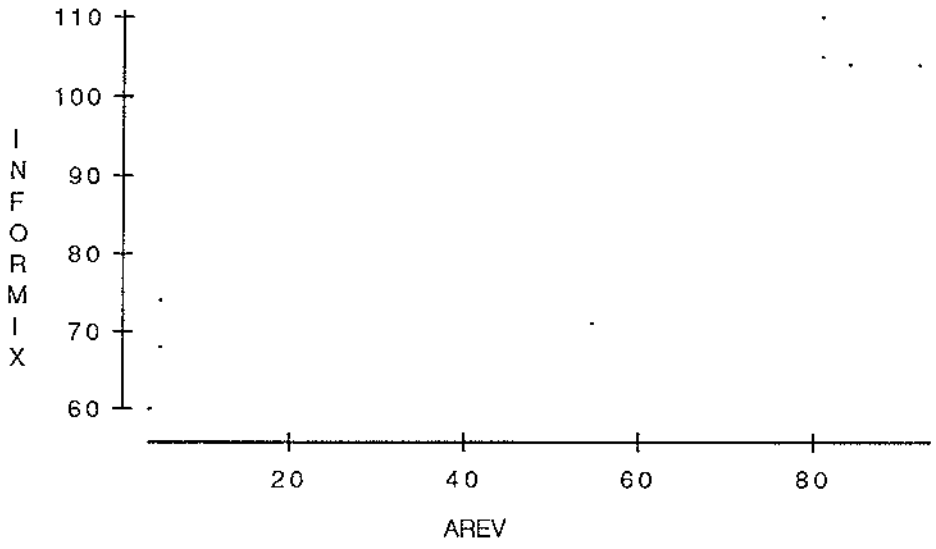


Fig. D.16 Plot of Informix (LOC) vs AREV (LOC)
Reports modules - System 2

Dependent variable is: INFORMIX
R² = 82.1% R²(adjusted) = 79.1%
s = 9.369 with 8 - 2 = 6 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	2419.29	1	2419	27.6
Residual	526.708	6	87.7847	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	63.1701	5.619	11.2
AREV	0.468401	0.0892	5.25

Table D.16 Regression analysis of Plot in Fig. D.16.

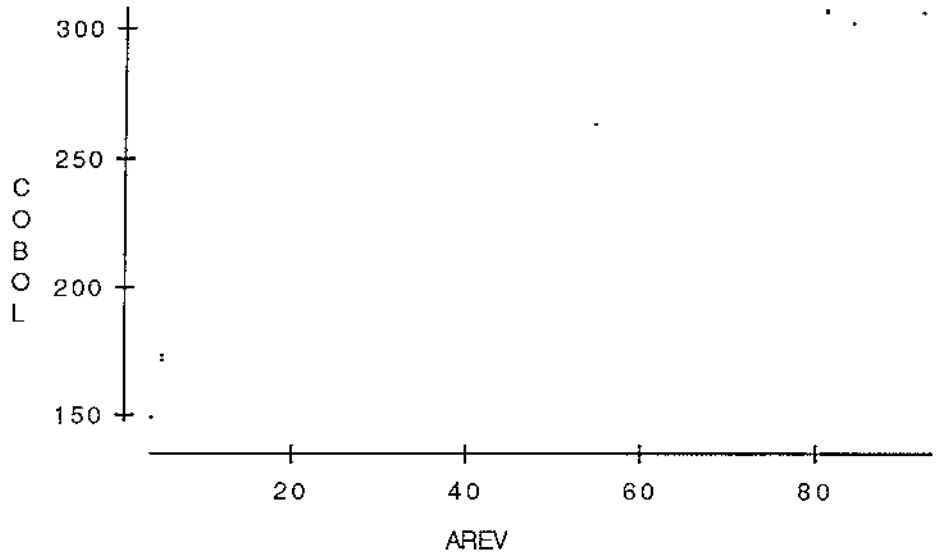


Fig. D.17 Plot of Cobol (LOC) vs AREV (LOC)
Reports modules - System 2

Dependent variable is: COBOL
R¤ = 97.9% R¤(adjusted) = 97.6%
s = 10.99 with 8 - 2 = 6 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	34370.4	1	34370	285
Residual	724.470	6	120.745	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	158.056	6.591	24.0
AREV	1.76549	0.1046	16.9

Table D.17 Regression analysis of plot in Fig. D.17.

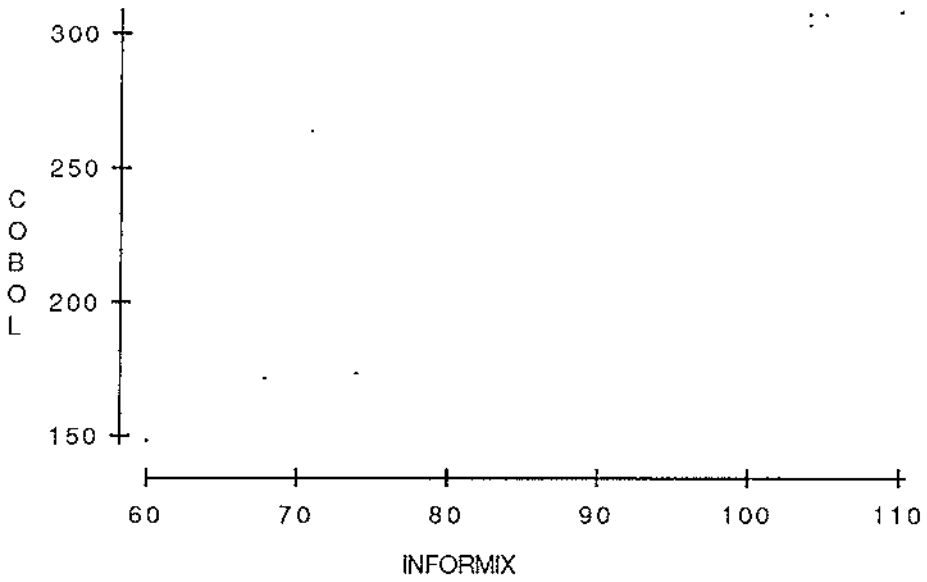


Fig. D.18 Plot of Cobol (LOC) vs Informix (LOC) Reports modules - System 2

Dependent variable is: COBOL
R¤ = 82.4% R¤(adjusted) = 79.5%
s = 32.09 with 8 - 2 = 6 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	28918.2	1	28918	28.1
Residual	6176.71	6	1029.45	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	-24.7014	52.66	-0.469
INFORMIX	3.13306	0.5911	5.30

Table D.18 Regression analysis of plot in Fig. D.18.

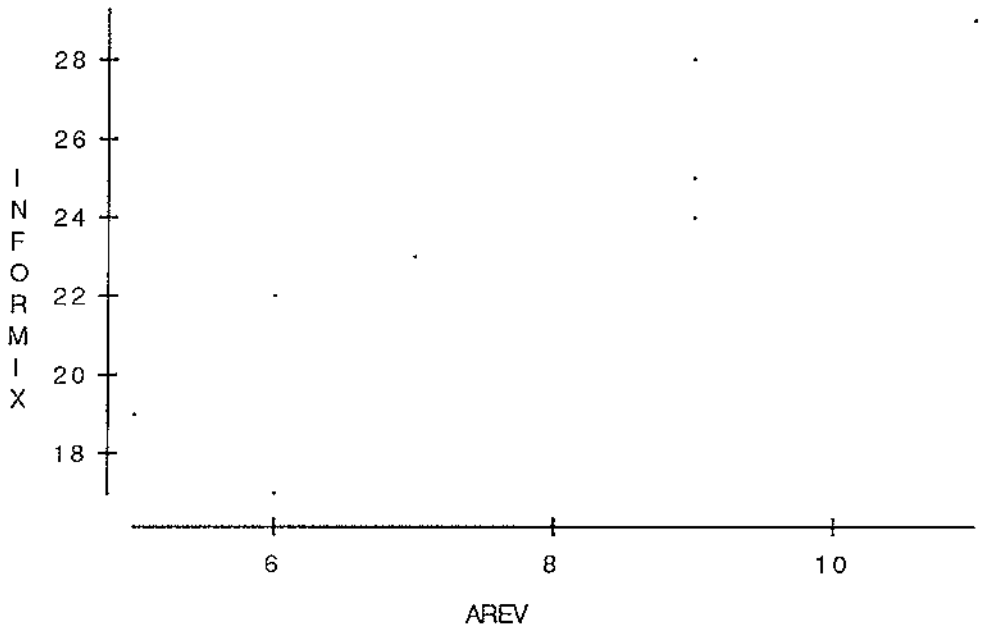


Fig. D.19 Plot of Informix (LOC) vs AREV (LOC)
Forms modules - System 2

Dependent variable is: INFORMIX
R² = 80.0% R²(adjusted) = 76.7%
s = 1.981 with 8 - 2 = 6 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	94.3242	1	94.32	24.0
Residual	23.5508	6	3.92514	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	9.51695	2.912	3.27
AREV	1.78814	0.3648	4.90

Table D.19 Regression analysis of plot in Fig. D.19.

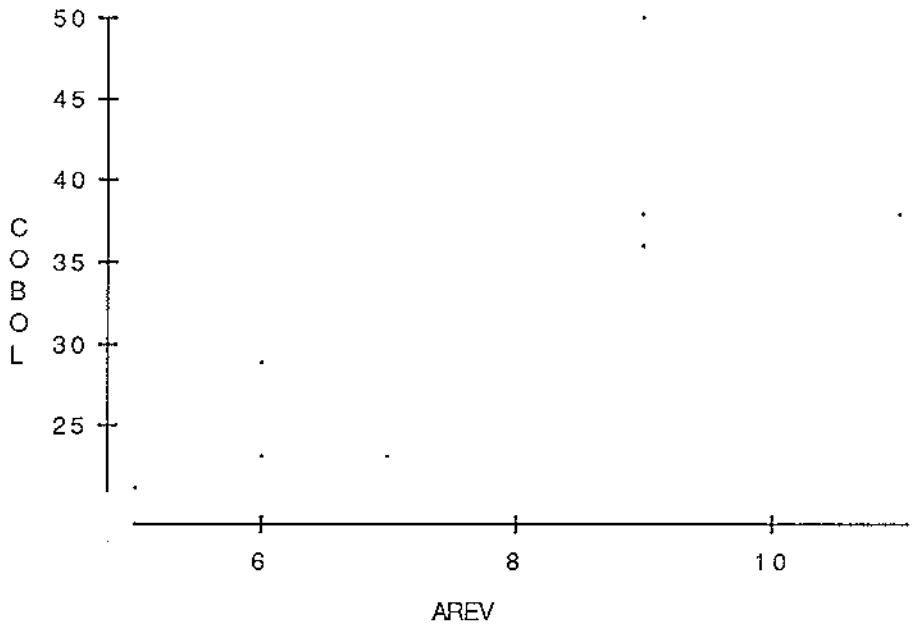


Fig. D.20 Plot of Cobol (LOC) vs AREV (LOC)
Forms modules - System 2

Dependent variable is: COBOL
R¤ = 61.0% R¤(adjusted) = 54.5%
s = 6.764 with 8 - 2 = 6 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	429.025	1	429	9.38
Residual	274.475	6	45.7458	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	2.69492	9.943	0.271
AREV	3.81356	1.245	3.06

Table D.20 Regression analysis of plot in Fig. D.20.

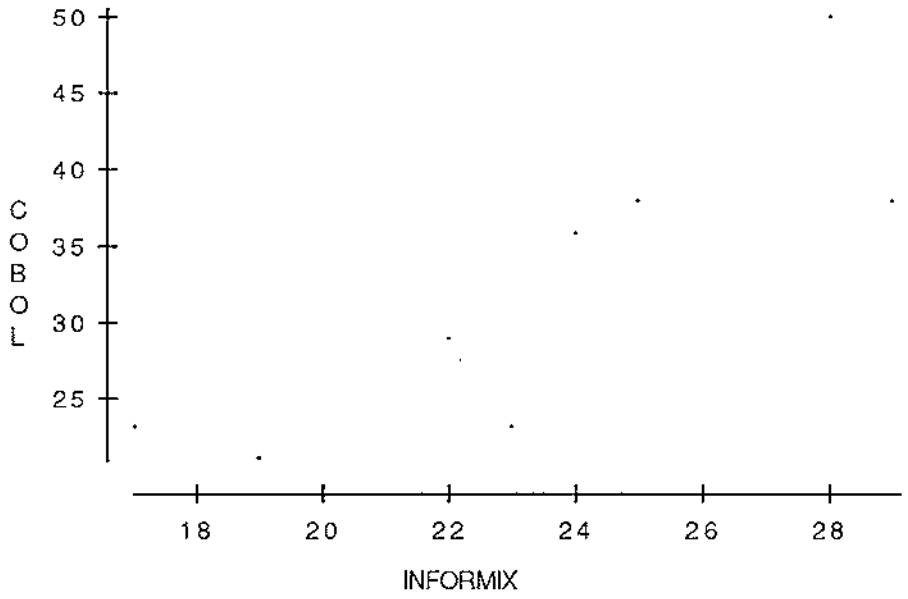


Fig. D.21 Plot of Cobol (LOC) vs Informix (LOC) Forms modules - System 2

Dependent variable is: COBOL
R² = 70.8% R²(adjusted) = 65.9%
s = 5.854 with 8 - 2 = 6 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	497.858	1	498	14.5
Residual	205.642	6	34.2736	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	-15.7890	12.77	-1.24
INFORMIX	2.05514	0.5392	3.81

Table D.21 Regression analysis of plot in Fig. D.21.

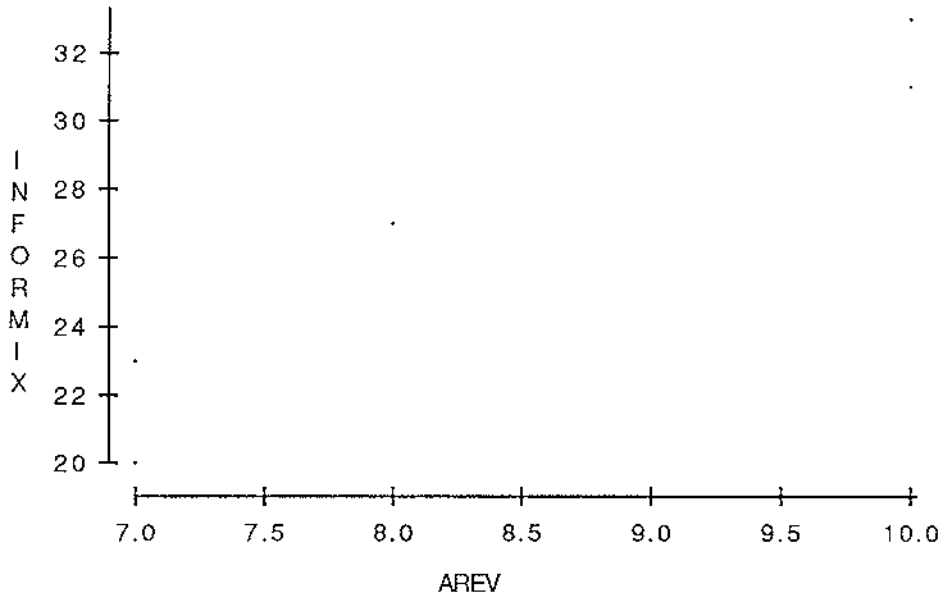


Fig. D.22 Plot of Informix (LOC) vs AREV (LOC)
Menus - System 2

Dependent variable is: INFORMIX
R² = 91.8% R²(adjusted) = 89.0%
s = 1.792 with 5 - 2 = 3 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	107.170	1	107	33.4
Residual	9.63043	3	3.21014	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	-1.86957	5.026	-0.372
AREV	3.41304	0.5907	5.78

Table D.22 Regression analysis of plot in Fig. D.22.

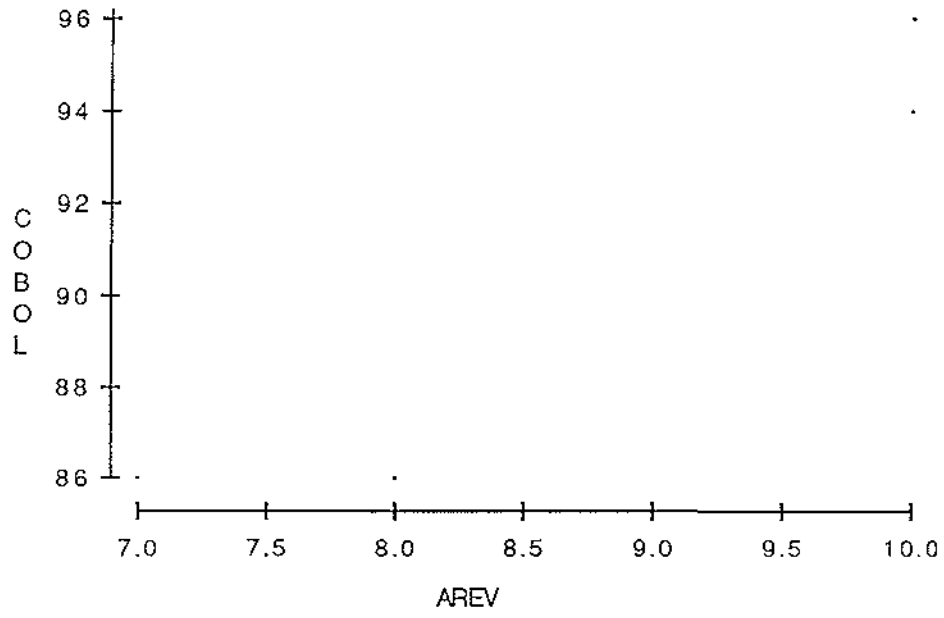


Fig. D.23 Plot of Cobol (LOC) vs AREV (LOC)
Menus - System 2

Dependent variable is: COBOL
R² = 90.9% R²(adjusted) = 87.8%
s = 1.736 with 5 - 2 = 3 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	90.1565	1	90.16	29.9
Residual	9.04348	3	3.01449	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	63.3043	4.871	13.0
AREV	3.13043	0.5724	5.47

Table D.23 Regression analysis of plot in Fig. D.23.

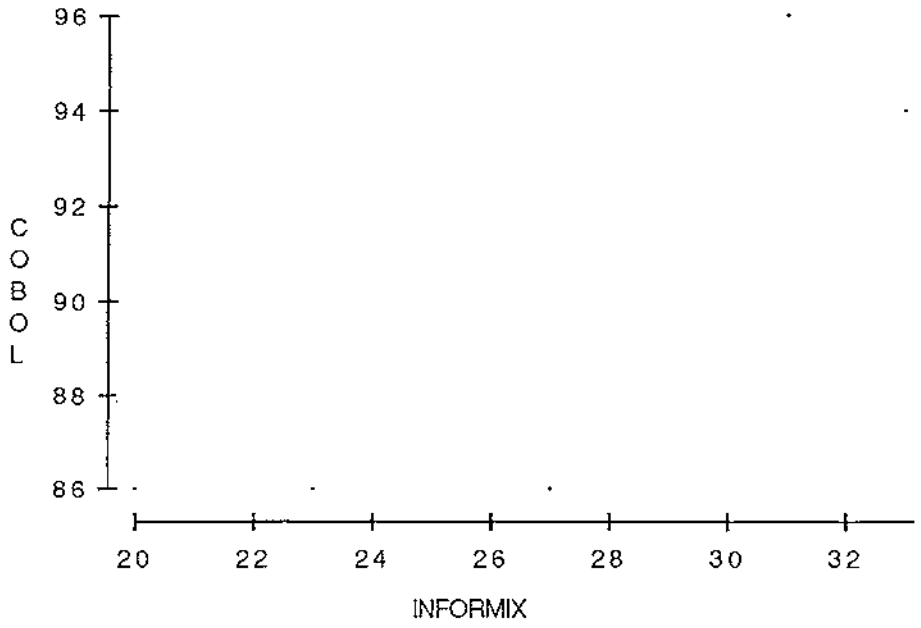


Fig. D.24 Plot of Cobol (LOC) vs Informix (LOC)
Menus - System 2

Dependent variable is: COBOL
R¤ = 72.4% R¤(adjusted) = 63.2%
s = 3.020 with 5 - 2 = 3 degrees of freedom

Source	Sum of Squares	df	Mean Square	F-ratio
Regression	71.8370	1	71.84	7.88
Residual	27.3630	3	9.12100	

Variable	Coefficient	s.e. of Coeff	t-ratio
Constant	68.5822	7.610	9.01
INFORMIX	0.784247	0.2794	2.81

Table D.24 Regression analysis of plot in Fig. D.24.