METRICS AND MODELS TO SUPPORT THE DEVELOPMENT OF HYBRID INFORMATION SYSTEMS

TREVOR THOMAS MOORES

Submitted for the degree of
Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

August 1993

THE UNIVERSITY OF ASTON IN BIRMINGHAM

Metrics and Models to Support the Development of Hybrid Information Systems

Trevor Thomas MOORES

Submitted for the Degree of Doctor of Philosophy

1993

THESIS SUMMARY

The research described here concerns the development of metrics and models to support the development of hybrid (conventional/knowledge based) integrated systems. The thesis argues from the point that, although it is well known that estimating the cost, duration and quality of information systems is a difficult task, it is far from clear what sorts of tools and techniques would adequately support a project manager in the estimation of these properties. A literature review shows that metrics (measurements) and estimating tools have been developed for conventional systems since the 1960s while there has been very little research on metrics for knowledge based systems (KBSs). Furthermore, although there are a number of theoretical problems with many of the "classic" metrics developed for conventional systems, it also appears that the tools which such metrics can be used to develop are not widely used by project managers. A survey was carried out of large UK companies which confirmed this continuing state of affairs. Before any useful tools could be developed, therefore, it was important to find out why project managers were not using these tools already.

By characterising those companies that use software cost estimating (SCE) tools against those which *could* but *do not*, it was possible to recognise the involvement of the client/customer in the process of estimation. Pursuing this point, a model of the early estimating and planning stages (the EEPS model) was developed to test exactly where estimating takes place. The EEPS model suggests that estimating could take place either before a fully-developed plan has been produced, or while this plan is being produced. If it were the former, then SCE tools would be particularly useful since there is very little other data available from which to produce an estimate. A second survey, however, indicated that project managers see estimating as being essentially the latter at which point project management tools are available to support the process. It would seem, therefore, that SCE tools are not being used because project management tools are being used instead. The issue here is not with the method of developing an estimating model or tool, but in the way in which "an estimate" is intimately tied to an understanding of what tasks are being planned. Current SCE tools are perceived by project managers as targetting the wrong point of estimation. A model (called TABATHA) is then presented which describes how an estimating tool based on an analysis of tasks would thus fit into the planning stage.

The issue of whether metrics can be usefully developed for hybrid systems (which also contain KBS components) is tested by extending a number of "classic" program size and structure metrics to a KBS language, Prolog. Measurements of lines of code, Halstead's operators/operands, McCabe's cyclomatic complexity, Henry & Kafura's data flow fan-in/out and post-release reported errors were taken for a set of 80 commercially-developed LPA Prolog programs. By re-defining the metric counts for Prolog it was found that estimates of program size and error-proneness comparable to the best conventional studies are possible. This suggests that metrics can be usefully applied to KBS languages, such as Prolog and thus, the development of metrics and models to support the development of hybrid information systems is both feasible and useful.

KEY WORDS: Software cost estimation, Software development, Knowledge based systems, Project management.

To the cat I never had.

Acknowledgements

Acknowledgements can never do justice to the number of people I owe thanks during my time at Aston University. I would like, therefore, to name a few people who are owed special thanks and hope that those missing from this list will forgive their omission. I would like to thank in particular:

Roger Barrett
Toby Barrett
Alistair Cochran
John Edwards
Chris Harris-Jones
Beryl Hodder
Nigel Hough
John Kidd
Alan Logan
Colin Thompson
Mike & Jayne Wood

The research described in this thesis was conducted as part of the IED4/1/1426 project developing a meta-method for the integration of conventional and knowledge based systems. The industrial partners are *BIS Applied Systems Limited* (Birmingham office) and *Expert Systems Limited* (based in Oxford). The research is supported by the *Science and Engineering Research Council* and the *Department of Trade and Industry*. The material within this thesis has previously appeared in the following publications:

EDWARDS J S and MOORES T T (1992a) "Metrics and project management models in the development of hybrid information systems." In *Proceedings, International Conference on Economics/Management and Information Technology*, Tokyo, 31 August-4 September, 1992.

EDWARDS J S and MOORES T T (1992b) "Metrics and models to help information systems project managers." In *Proceedings, OR Society Annual Conference*, Birmingham, 8-10 September, 1992.

EDWARDS J S and MOORES T T (1994) "A conflict between the use of estimating and planning tools in the management of information systems." To appear in *The European Journal of Information Systems*, **3**.

MOORES T T (1992) "On the use of Software Cost Estimating tools." *Doctoral Working Paper No.6 (NS)*, Aston Business School, Aston University, April 1992.

MOORES T T and EDWARDS J S (1992a) "Could large UK corporations and computing companies use Software Cost Estimating tools? A survey." *The European Journal of Information Systems*, **1**(5), 311-319.

MOORES T T and EDWARDS J S (1992b) "Why would anybody use an estimating tool? - A comparison of users and non-users." In *Proceedings, European Software Cost Modelling Meeting*, Munich, 27-29 May, 1992.

List of Contents

List of Figures

List of Tables

Chapter 9

1. Introduction

> *"If a man will begin with certainties, he shall end in doubts; but if he will be content to begin with doubts, he shall end in certainties."* Francis Bacon ('The Advancement of Learning').

Summary: *The IED programme is outlined as well as the main body of the work for Project IED4/1/1426, from which this research is taken. The main aim of the IED4 project is to support management in the integration of conventional and knowledge based methods. The research aims, goals and structure of this thesis is then set out.*

Developing software systems is a complex and difficult task. A software system is defined here as any system which makes use of an electronic platform and is driven by a piece of software that carries out some function: normally, that the system provides some required response or output given some expected input. This definition does not necessitate that the system provides any useful or beneficial function, since what counts as 'useful' or 'beneficial' can be a matter of serious debate. For instance, would the guidance system in a missile be described as "beneficial" for the recipient? When the system is small, developed by one person and is for personal use, these issues of 'use' and 'benefit' can be answered by the same person. For larger systems, however, there is the problem of communicating between the client and the developers. If there is any output from this discussion it is usually a document which outlines the rôle (and thus the functionality, use and benefits) of the required system.

The problem of communication is not the only issue, however, since as the clients' requirements are transformed into a (hopefully operational) system, it may sometimes be difficult to believe that the process by which it is brought about is driven by anything other than magic. This feeling can be reinforced by the fact that - for most of its life - a software system is no more than an intangible object within the circuits of a CPU fronted by the Cyclops-like eye of a monitor. But if software development is driven by magic then there need be no good understanding of how the actions of the development team result in the final product. Like any other type of magician, a good project manager would be someone that had an intuitive understanding of the relevant incantations and an innate skill to direct the

project team along the right track. In this respect, there is no place for project management techniques or methods because magic - by its very nature - breaks all the natural laws and so is beyond any serious study.

The goal of software engineering, on the other hand, is to destroy this description of software development and aims to transform the state-of-the-art from that of magic to that of science. The definition provided by Boehm (1976) is that software engineering is:

> "The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them (p1226)."

Boehm goes on to say that, ideally, such a development environment would make use of techniques such as formal, machine-readable specifications, top-down design, structured programming and automated aids for software testing. These techniques would allow the previous incantations to be replaced by a set of well-understood, repeatable and refinable processes by which systems emerge in a good, orderly fashion. With such a well-formed process in place it would then be possible to understand the effects of certain actions and allow a meaningful set of best practices to be established. What is important to Boehm is that software should be cost-effective and reliable enough "to deserve our trust (*ibid*, p1226)", and as software systems increasingly pervade daily life the issue of "trusting" software becomes ever more important.

Borning (1987) questions whether we can ever have sufficient faith in systems which (for instance) are meant to be safety-critical. Borning gives two examples of the failure of US missile attack warning systems: firstly, in October 1960 a "massive missile attack" with a certainty of 99.9% turned out to be the rising Moon; secondly, in June 1980 an 'attack' with a random pattern turned out to be a chip failure. Only the lack of visual verification in 1960 and the illogical pattern of the attack in 1980 meant that neither alert resulted in a mistaken retaliation of missiles. The first example was a lack of precision in defining the environment in which the system was meant to operate, while the second example was clearly a hardware problem. In a third example, however, Borning notes that even when the software is dealing with more everyday problems, poorly defined software can prove fatal when the underlying model is inaccurate. In 1983 severe flooding in the lower Colorado river claimed a number of lives. The problem was that there was a "monumental mistake" in computer projections of snow melt-off flow, and so, too much water had been dammed upstream before the spring thaw. With each of these examples Borning's contention is that:

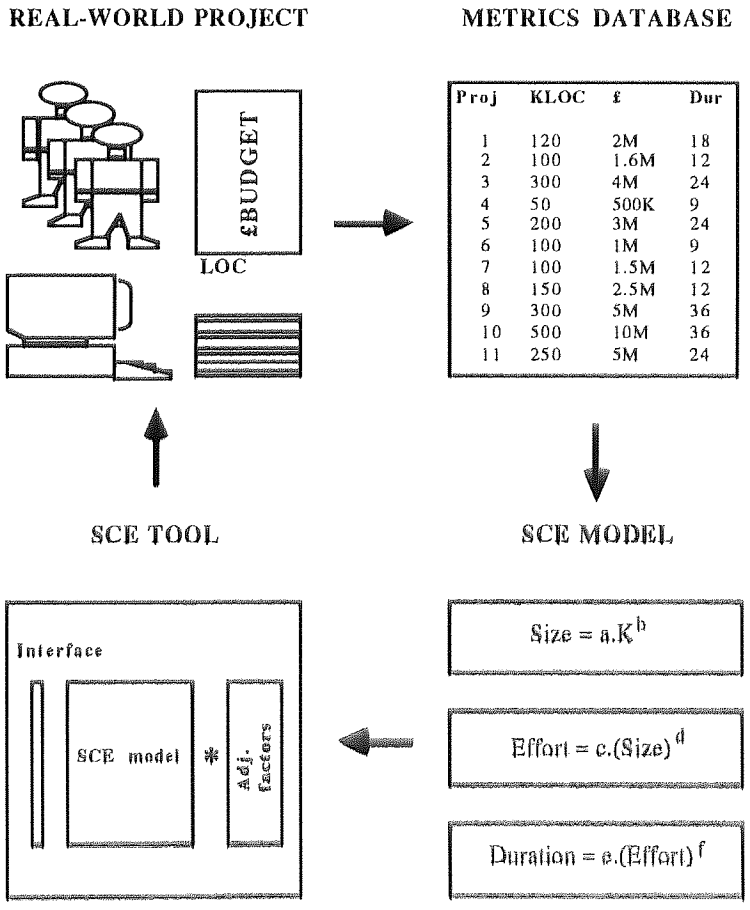
> "... the sheer complexity of the system is itself a basic cause of problems. Anyone who has worked on a large computer system knows how difficult it

> is to manage the development process; usually, there is *nobody* who understands the entire system completely (p120, original italics)."

Software engineering would not deny that trapping the semantics of software development into a scientific framework will be a difficult and complex problem in itself. The point is, however, that it cannot remain a black art.

Perhaps, then, a more fitting analogy for the project manager is the medical doctor: employing a set of well-known techniques to reveal the structure of the patient and identify the symptoms of common diseases. In the project manager's case, the patient is the software development project while the common diseases to be identified are malfunctioning tasks or software. It may still be a matter of debate whether the most effective medicine is to stop the disease before it starts (preventative) or treating the disease as it appears (reactive). Get it wrong either way and the patient/project could still end up dead. The fact that software development still has the air of "witch doctoring" even given the number of management "treatments" which are now available is a major part of this thesis.

Figure 1.1 : The metrics→models→tools approach



The focus of this research, therefore, is the development of metrics, models and tools to support the development of hybrid information systems. The definition of a metric used throughout this thesis is as follows:

> *A 'metric' is a measurement of some aspect of the development process or products which gives some indication of other (more important) properties.*

It will be assumed throughout that there is an intimate connection between the properties of a project under study, the metric data which can be extracted from the project, the models which can be deduced from this database of information, and the final development of project management tools which can then be used to estimate the same properties for subsequent projects. This metric→model→tool approach is represented in Figure 1.1. The specific need here is for a set of metrics which can form the basis of system size and structure models and which can also be applied to both conventional and knowledge based systems. The focus on size and structure is made on the grounds that:

- 'Size' is the key input to models of effort and duration. Early estimates of effort allows the cost and feasibility of the project to be assessed.
- 'Structure' is the means by which the error-proneness of software can be deduced. The "quality" of a piece of software is often boiled down to a judgement of its likelihood to fail.

Metrics for size and structure, therefore, are the basis of models for cost and quality. Such models have the general form:

$$U = a.K^b * \text{Adjustment\_factors}$$

where 'U' is some unknown property, 'K' is some known or at least measurable property, while the parameters *a* and *b* are typically found by regression. Adjustment factors capture internal differences between similar projects which make one project different from another. Such differences might include the experience of the development team, the need for 100% reliability, etc. By representing the model in such a way that its use is simplified, e.g., by building an electronic version with interface and help facilities, then the model becomes a tool which is potentially usable by a third party, such as project managers on other projects. Because the metrics→model→tool approach begins and ends with the real-world problem of software development, the tool becomes useful only if it can accurately predict the relationship between U and K for subsequent projects.

The models of particular interest here are those for estimating the cost, duration and quality of software systems. Models of cost and duration are often related in the sense that models estimating person-months effort is a surrogate for cost, while an estimate for effort is the key input to models of project duration, thus:

Effort	=	$a.(Size)^b$ * Adjustment_factors	(in person-months)
Cost	=	Effort * £person-month^{-1}	(in £s)
Duration	=	$c.(Effort)^d$	(in calendar-months)

As can be seen, system size is a key input to the effort model. The availability of this first measurement, will be seen to be an important problem for software cost estimating (SCE) models.

Equally problematic is the concept of 'quality', defined by Boehm *et al* (1976) as a hierarchy of desirable attributes, including reliability, portability, efficiency and testability. Despite the breadth of these features, the vagueness of these notions has resulted in quality becoming synonymous with reliability and the ideal of zero defects. The concept of reliability was quickly defined as the time between system failures, or mean-time-to-failure (MTTF)(e.g., Musa, 1979). The longer the MTTF the better the reliability of the system and, hence, the better the quality of the system as a whole. The issue of reliability continues to be the most active area of research in quality (e.g., Lipow, 1982; Musa *et al*, 1987; Brocklehurst & Littlewood, 1992).

Issues of quality for KBSs have also looked at reducing the likelihood of defects. This could mean either testing the faults in the reasoning of a KBS (Miller, 1990; Preece, 1990), or by applying a rigorous method from specification to development (Plant, 1991). Although 'quality' clearly involves more than MTTF (Card, 1990), the likelihood of a "bug" or error being discovered within the system can be taken to be the most significant attribute. A system which contains a number of bugs would substantially reduce its ability to attain any of the other quality attributes. For this reason, metrics for 'quality' will be taken to be measurements that estimate the likelihood of a program having a defect or "bug".

What remains unclear, however, is whether there are any metrics which can be used when a project has <u>both</u> conventional and knowledge based components. A project manager of such a hybrid system would thus have to face the additional problem of integrating these often specialist tasks into a coherent development project that can still be understood and controlled. But which tools would be most useful, and what set of techniques can be developed which can deal with both conventional and knowledge based components? These are the questions that need to be answered. The rest of this chapter will set out:

- the background to the research (carried out as part of Project IED4/1/1426);
- the main questions to be addressed by this research;
- the structure of this thesis.

1.1 Background to the research

The research contained within this thesis is based on work carried out as part of Project IED4/1/1426. The project is part of the Joint Framework for Information Technology (JFIT) programme funded by the Department of Trade and Industry (DTI) and Science and Engineering Research Council (SERC), and administered by the DTI's Information Engineering Directorate (IED) and SERC's Information Technology Directorate (ITD). This section will outline the aims of the JFIT programme and Project IED4/1/1426 in particular.

1.1.1 *The JFIT programme*

The JFIT programme was designed to follow the example of ALVEY and ESPRIT by continuing to foster links between industry and academia. In particular, JFIT set out to advance work in the software engineering, knowledge based systems and human computer interaction fields. Morgan *et al* (1988, p2-6), define JFIT as aiming to:

- increase the applicability of existing and anticipated research results;
- maintain and increase the UK's strength in strategically important areas of information technology (IT);
- ease and accelerate the process of technology transfer between project partners and from developers to users.

The JFIT programme co-ordinates a number of projects within the fields of microelectronic devices, systems architectures, systems engineering, and control and instrumentation.

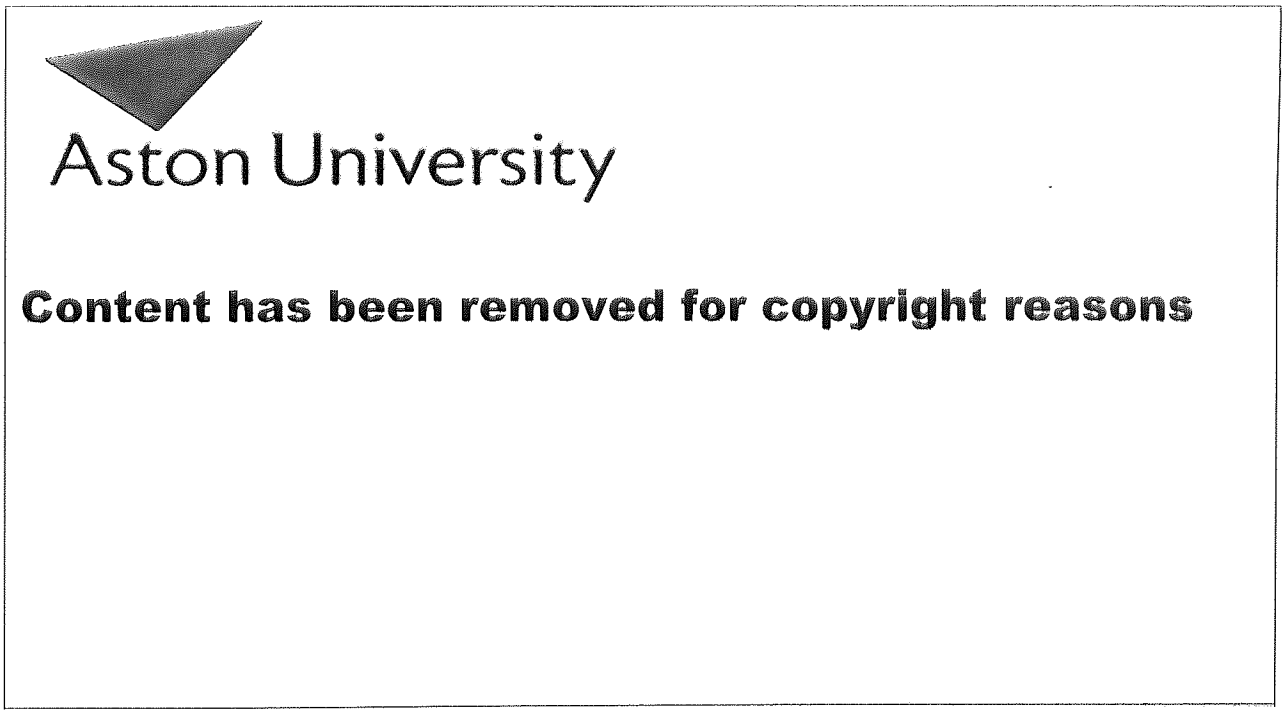
1.1.2 *The IED4/1/1426 project*

The IED4/1/1426 project falls within the systems engineering field and is supervised by the DTI's Information Engineering Directorate. The project is a collaboration between BIS Information Systems Ltd (Birmingham office), Expert Systems Ltd (based in Oxford) and The University of Aston. The project set out to develop tools and techniques to support and control:

- the integration of conventional (DP) and knowledge based system (KBS) methodologies;
- the configuration of a such a hybrid method to a particular project;
- the estimation of the cost, duration and quality of a hybrid system.

The key areas being addressed are software development methodologies and project management. The research into methods integration and configuration was carried out by BIS Information Systems Ltd and Expert Systems Ltd, while the problem of estimation is addressed by this thesis. To understand fully the problems being addressed by this research, therefore, it is necessary to describe the other half of the project. The rest of this section will describe the need for methods integration, the development of the RUSSET integration tool, and consequently, the need for metrics.

Figure 1.2 : The RUSSET structural model (cf. Harris-Jones *et al* 1993, p29)



1.1.3 *Methods integration and RUSSET*

The key to methods integration is to describe methods in such a way that components from different methods can be compared, and where identical components can be identified and deleted from the final hybrid method. "Components" here are the tasks, techniques and products described by the method. A tool - called RUSSET -

has been developed to carry out the integration of methods (Harris-Jones *et al*, 1992, 1993). RUSSET contains three models: a structural model; a perspectives model, and; a methods process model (MPM). These models are used to describe methods in general and properties of methods components in particular.

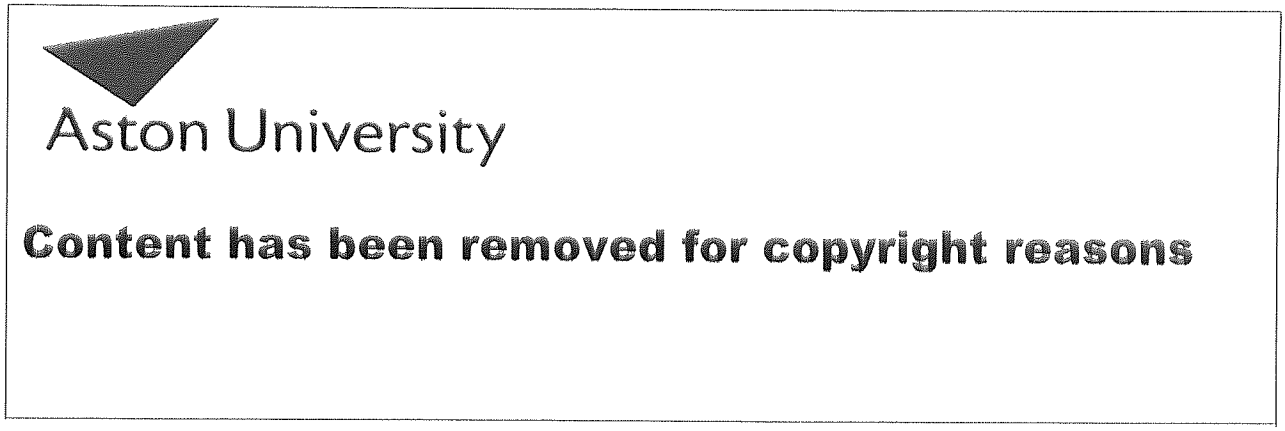
The structural model abstracts descriptions of method components into three levels (see Figure 1.2): from the description by the methods handbook (level 1); to an intermediate frame-based representation which has a common structure across all methods (level 2); to a model of all tasks, techniques and products a method must have in order to be a complete method. In this sense, level 3 must contain a certain amount of knowledge about what a method is and is used to identify gaps and overlaps between methods. The expertise at this level was supplied by an expert in methods and methods integration.

The perspectives model defines the orientation of a method in terms of the class of objects which are used to describe the properties of a system. The method perspective defines which set of (correctly oriented) techniques are able to represent the system. The concept of a perspective follows that also used by Olle *et al* (1988) where conventional (DP) methods are classified as having one of three orientations:

- data perspective (the system is described in terms of transformations of data and represented using techniques such as data normalisation and entity-relationship diagrams);
- process perspective (the system is described in terms of discrete inter-related processes and represented using techniques such as functional decomposition and process dependencies);
- behaviour perspective (the system is described in terms of responses to certain events and represented using techniques such as tactics and strategies).

Knowledge based systems (KBSs), on the other hand, are seen as being strong in the areas of problem-solving behaviour and domain knowledge. The behaviour perspective was extended to take account of the first of these areas. A fourth (knowledge) perspective was also needed in order to deal with methods defining a system in terms of domain knowledge objects. The knowledge perspective defines a static but semantically rich class of (knowledge based) objects which are represented using techniques such as object hierarchies and semantic nets. This places the knowledge perspective on an opposite pole to the process perspective (see Figure 1.3).

Figure 1.3 : The four systems perspectives
(cf. Harris-Jones *et al* 1993, p30)



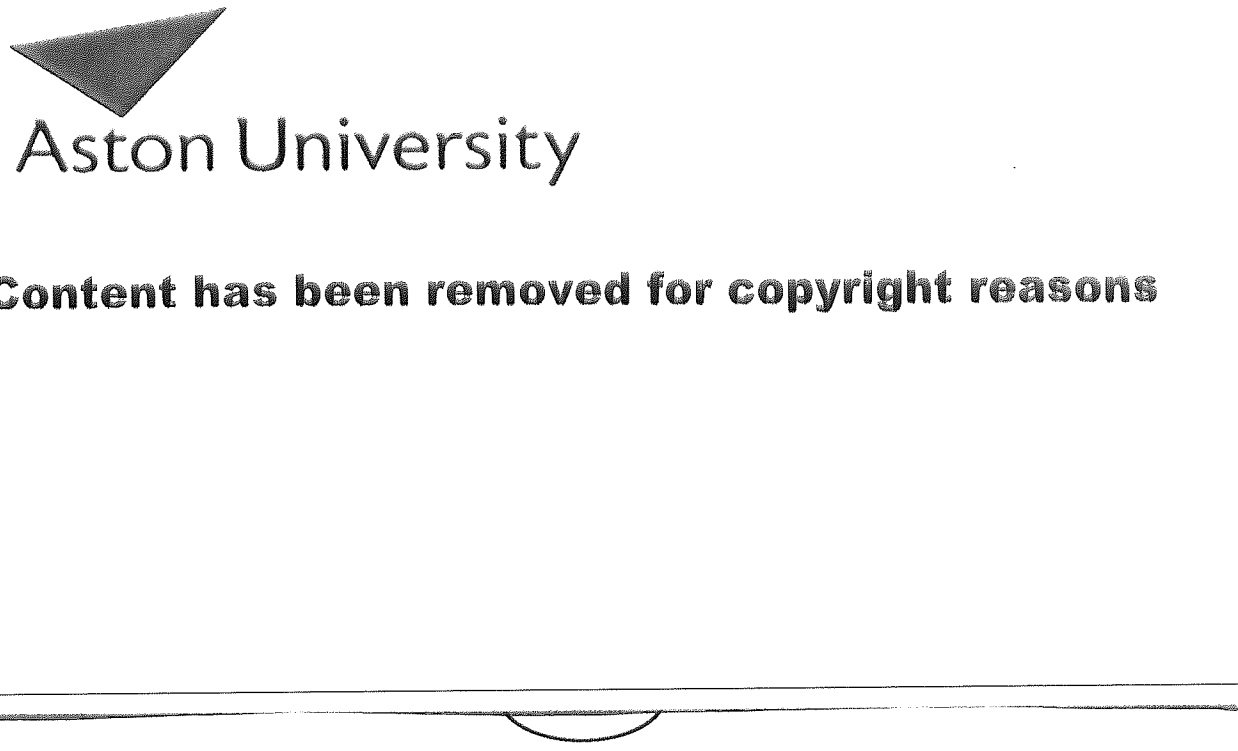
The perspective model is used to determine whether two or more methods components which match functionally (they perform the same task) are identical or complimentary. Two components are identical only if they match functionally and have the same perspective. If the perspectives are different, then the matching components provide different representations of the system and so one cannot be substituted for the other.

A methods process model (MPM) is a description of how the methods components in the level 3 structural model are configured into a life-cycle. That is, an ordering of the tasks is imposed usually by logical relation where X must be done before Y can begin. Harris-Jones *et al* (1993) point out (p33) that very few methods give detailed descriptions of the MPM and so it is not always clear how the method can be configured. The MPM used by RUSSET (see Figure 1.4) is similar to Boehm's (1988a) spiral model except the "assess" quadrant is replaced by "decision" on the grounds that:

> "At this point in a project a decision has to be made about the next step. Although an assessment of the risk is a very significant element in the decision making process, there are other elements including, for example, internal political issues. These can often kill a project far more efficiently than risk factors (Harris-Jones *et al*, 1993, p34)."

The MPM is used to configure the method components using knowledge of the way in which a development can be organised (Waterfall, spiral, etc.), and management rules of thumb such as the larger the project the more checkpoints required (while the smaller the project the fewer reports and checkpoints required).

Figure 1.4 : RUSSET's methods process model (MPM)
(cf. Harris-Jones *et al* 1993, p33)



In summary, therefore, the structural model, perspectives model and MPM are used to describe conventional and KBS methodologies in such a way that different methods can be efficiently integrated into a coherent whole. This will enable organisations to take their existing conventional method and integrate it with a KBS method. The integration process takes into account information about the project on which the method is to be used, tailoring the contents of the method accordingly. In this way, not only is a fully integrated method generated, but the output from RUSSET also provides a sound basis for detailed project planning.

Even with a well-constructed method in place, however, whether the proposed system is developed at all could well depend on the project manager's ability to accurately estimate, plan and control the project as a whole. If mistakes are made over the cost and duration of the project then at best there will be a strain on limited personnel and hardware resources, and at worst, the project could be cancelled due to costs outweighing any remaining benefits. It is at this point that the problems of project management are put into sharper focus.

1.2 Main research questions

There are a number of questions which must be answered before a clear definition of the most useful metrics and models can be established. The questions are related to those a project manager might ask if presented with a set of metrics and estimating tools for use on a project. These questions are initially of a theoretical nature and relate to the use of metrics and estimating tools on any project (whether conventional or knowledge based) and ask:

- What problems do project managers face?
- What are "metrics" and "estimating tools"?
- How are they developed?
- How are they used?

Only by investigating these theoretical questions can the feasibility of using metrics and estimating tools be established. The questions then become more empirical and ask whether the theory relates to the way project managers actually work and generate estimates. The empirical questions ask:

- Are estimating tools being used in industry?
- If not, why not?
- Does this affect the sorts of metrics and models which a project manager would actually find useful?

It is entirely possible that even though metrics have a sound theoretical rôle to play they have been superseded, replaced, or ignored by current practice. The result of these empirical questions would modify the set of tools and techniques which might be usable (they have a rôle to play), into a set of tools and techniques which are useful (they are the best or only way of solving some project management problem). Having established that metrics and estimating tools are both usable and useful, the question then becomes whether the technology can be used on projects which have both conventional and knowledge based components. The final question therefore becomes:

- What would a hybrid metric/model/estimating tool look like?

These are the questions which are addressed by this thesis.

1.3 Thesis structure

The structure of this thesis is shaped by the need to answer the theoretical and empirical questions outlined above before addressing the feasibility of applying metrics and developing size and structure models for hybrid systems. After outlining the problems being addressed in Chapter 1, the rest of this thesis can be divided into three parts:

1.3.1 *Theoretical issues*

Chapter 2: This chapter outlines the problems faced by project managers in the development of software systems. Recent results continue to suggest that there is a lack of control even though a number of management techniques have been developed since the 1950s. Various methods of deriving an estimate are described, before explaining the use of life-cycle models to understand the process of software development. It will be argued that although these planning techniques are well established, control requires accurate numbers to be available before the plan can help in project control. This establishes the importance of metrics in project management.

Chapter 3: A description of the nature of metrics as a measurement of the software products or the development process itself is given. Particular attention is paid to the "classic" metrics that measure size (Halstead's Software Science and Function point analysis), and structure (McCabe's cyclomatic complexity and Henry and Kafura's data fan-in/out). It will be argued that although size and structure metrics may form the basis of any metrics programme, it is not at all clear that well-defined counting strategies for such metrics have been established. As such, the evidence for the usefulness of metrics for conventional systems' development remains in doubt.

Chapter 4: This chapter describes the few metrics which have been developed specifically for knowledge based systems (KBSs). They are typified in the same way that metrics for conventional systems are plausible, mainly directed towards measurements of program code, while more useful estimating models suffer from a lack of validation. It will be argued that although there is meant to be a theoretical difference between conventional and knowledge based systems, these KBS metrics show that measurements of KBSs do not differ substantially from measurements of conventional systems. This holds out the possibility that conventional metrics could be extended to KBSs.

Chapter 5: This chapter describes the process by which raw (project) data can be converted into a software cost estimating (SCE) tool that estimates properties such as cost, effort (or manpower) and project duration. Such tools are developed by collecting historical data on the resources expended by a completed project and developing a regression model across the dataset. It will be argued that although many of the well-known SCE models have been developed into commercial tools, there is evidence to suggest that the models use poor statistical experimentation, they can be overly complex, and are generally inaccurate. This would seem to cast doubt on the suggestion that SCE tools can be useful to project managers.

1.3.2 *Empirical issues*

Chapter 6: This chapter presents the results of a survey of large UK corporations and computing companies. The survey aimed to discover whether: a) estimation was seen as a problem; b) estimating tools were in use; and, c) companies had the sort of development environment within which an estimating tool could be developed or calibrated. It will be seen that while most respondents see estimation as a problem and could develop/calibrate an estimating tool, less than a third actually do.

Chapter 7: Given that it is now known that project managers could but do not seem to be using Software Cost Estimating tools, the next question to be answered is "Why not?" A follow-up to the original survey is described in which a conflict between the use of estimating and planning tools is identified and resolved. It will be argued that by redefining SCE tools as those which can generate estimates before a plan (at a bidding/tendering stage) and supports the creation of plan-based (task-based) estimates, estimating tools do indeed have a necessary place in the armoury of current project managers.

1.3.3 *Hybrid metrics and tools*

Chapter 8: Building on the empirical results of the previous chapters, a definition and specification of a task-based estimating tool is presented. The tool is here called TABATHA. Given that the research is based within IED4/1/1426 it is assumed that the nature and number of tasks within a project are to be provided by the RUSSET tool. The general task-based model is presented along with the seven measurements required to instantiate and validate TABATHA. It will be argued that such a tool can deal with re-estimating as user requirements change because the effect of project changes occurs at the task level, exactly the point at which TABATHA generates its estimates.

Chapter 9: In order for TABATHA to be useful for hybrid projects, it is necessary to assume that the languages used to develop knowledge based components would also be susceptible to the same form of (metric) analysis as conventional languages. Prolog is taken to be a mainstream KBS development language and so a description is given here of an analysis of 80 commercially-developed Prolog programs. It will be argued that by refining the definition of three "classic" metrics (software science, cyclomatic complexity and data fan-in/out), excellent models of both Prolog structure and size can be developed. This is the first step to developing the more sophisticated models required by TABATHA to estimate the size and cost of Prolog components.

Chapter 10: The contributions to knowledge derived from this research are set out as well as the limitations of the research. The research questions posed in Chapter 1 are answered and areas of further work identified. The final remarks revolve around the potential of TABATHA (in Chapter 8) and the high-level sizing tool (in Chapter 9) to stand as useful models for use in hybrid systems development. The conclusion is that they can.

2. Problems of Project Management

> *"Software systems are similar to biological systems...They arise out of a painful birth, require intensive care in the beginning, slowly mature, reach their state of maximum benefit after several years, become increasingly inflexible and difficult to support, die, and leave behind a wealth of experience from which grows a new generation of systems."* H.M.Sneed (1989, p21)

Summary: *This chapter outlines the problems faced by project managers in the development of software systems. Recent results continue to suggest that there is a lack of control even though a number of management techniques have been developed since the 1950s. Various methods of deriving an estimate are described, before explaining the use of life-cycle models to understand the process of software development. It will be argued that although these planning techniques are well established, control requires accurate numbers to be available before the plan can help in project control. This establishes the importance of metrics in project management.*

Project management is defined here as the art (or science) by which techniques of control are exercised by an individual or group of individuals (management) over a set of tasks that has the properties of:

- at least one active or consumable resource;
- a start point;
- an end point;
- a goal which needs to be achieved.

The rôle of project management, therefore, is to organise the resources between the start and end points in such a way that the specified goal is achieved at an optimum cost. In software development terms, the resources are personnel and computing facilities and the specified goal is the delivery of a software system with its associated documentation. This definition assumes that some identifiable method of development is in place, otherwise there would be no recognisable process to be "managed". This point accords well with recent theories that propose a systematic approach to the improvement of software development processes (e.g., Humphrey, 1989). The definition presented here also suggests that there are preconditions to effective project management, notably that:

- the start point, end point and project goal are specified clearly at the outset;
- the resources allocated are sufficient;
- none of the above are significantly changed during the life-time of the project.

If these preconditions are violated then the project manager could be faced with few "winning moves" (Boehm & Ross, 1989). Creating the right conditions is the key to good management and in software development this begins at the outset when the client demands answers to questions such as "How much will it cost?" and "How long will it take?", while the project manager will also have to decide on the level of support required and answer questions such as "How many bugs is it likely to have?" and so "How much testing/fixing should be done?" These problems are compounded by the fact that even if all the resources are available, the chances are that the system specification will change as the project examines and produces the stated functionality. Indeed, it has been suggested that requirements specifications should be fluid and open to change (Spence & Carey, 1991).

Throughout the life of the project there will be a constant tension between demands for higher quality, more functionality, reduced development time and lower cost. This tension is sometimes referred to as the 'devil's square', where it is possible to satisfy two of the demands but at the expense of the other two. For instance, increasing quality and functionality will impact on the cost and time to deliver and this cannot necessarily be solved by adding more people (Brooks, 1975). With these problems at hand, it is essential that management understands the current state and progress of the project. The rest of this chapter will investigate the degree to which project managers can be said to be "winning", and discusses:

- the software "crisis";
- techniques for project management;
- methods of estimation;
- models of software development.

2.1 The software "crisis"

Since the late-1960s the software development industry has seen itself facing a crisis. The crisis began when the relative cost of hardware decreased to such an extent that the major expense in developing a system was the labour-costs involved in designing, writing and testing the software. The crisis for the development manager is the apparent inability to plan for and allocate these expensive resources so that projects are completed on-time and within budget. For instance, in March 1993 the national UK newspapers reported that the TAURUS system - which was meant to control share settlement procedures in the City of

London - had been abandoned after 10 years of development and over £275 million of investment. It was reported as needing a further 2-3 years before it became usable and eventually led to the resignation of the chief executive of the Stock Exchange. Surveys of senior management continue to find that less than half are satisfied with their software development departments (e.g., Gunton, 1989).

Some of these problems may be explained by the problems developers perceive with top management. For instance, AEI (1989) conducted a survey of UK, French, German, Italian and Swiss companies. The 635 replies (24.4% response rate) were mainly from the Retail/Finance/Manufacturing sectors with 159 replies received from UK companies. Most interestingly, to the statement "Top management will not devote sufficient time to make Information Technology a success," 37% of UK respondents agreed while 37% disagreed (p23). Agreement to this statement was found to be 33% in Italy, 21% in France and 21% in West Germany. The polarity in UK companies may explain the result that to the question "Top management are satisfied with the present contribution from IT investment," only 27% of UK respondents agreed (p26). This proportion is much lower than those responding from France (53% agree), West Germany (52% agree) or Italy (58% agree). Of the companies which rated their use of IT as "very successful", only 11% agreed with the first statement while 63% agreed with the second. The conclusion seems to be that developing information systems is not purely a technical problem, but requires continued and committed support from top management.

Table 2.1 : Total IT expenditure for the UK

Year	Expenditure (£Bn)
1986*	13.6
1987*	16.0
1988*	18.0
1989	-----
1990**	25.7
1991**	27.1

* Computer Weekly (1988)
** OTR-Pedder/CUYB (1992)

This is particularly important considering the cost of software development. Boehm & Papaccio (1988) expected annual US software costs to rise by 12% each year from around

$70 billion in 1985. This rate has been exceeded in the UK with total IT expenditure rising from £13.6Bn in 1986 to £27.1Bn in 1991 (see Table 2.1). The rise from 1990 to 1991 is only 5.5%, however, and coincides with the onset of the recession in the UK.

If budgets become tighter then one might expect that the need for project control becomes even more important. However, while management tools - such as Gantt charts - have been developed to support project planning since the 1950s, the ability of any given manager to control a project seems to remain a serious problem. Surveys of European companies show that effort and schedule overruns are in the order of 30-50% (e.g., Jenkins *et al*, 1984; Phan *et al*, 1988). If the lower end of this scale (30%) is taken to be the typical project over-spend throughout the UK software development community, then the UK spent over £6.3Bn more in 1991 than was originally planned. Such levels are not confined to the UK since within a sample of 600 US firms, 35% admitted to currently having at least one runaway project (Boehm, 1991). Clearly, there is a problem with deducing the cost and duration of software development projects.

2.2 Techniques for project management

Managing a project clearly demands control, and control relies on planning and monitoring. In order to plan, the manager must analyse the nature of the project. A number of graphical techniques have been developed which represent the tasks scheduled, ongoing, completed or even abandoned. The most well-known of these are the work breakdown structure (WBS), Gantt chart and project network analysis (using PERT charts).

2.2.1 *Work breakdown structures (WBSs)*

Work breakdown structures (WBSs) are block diagrams which set out the major "chunks" of work to be carried out during a project (see Figure 2.1). Such diagrams are useful since a project manager is unlikely to be able to immediately specify all the tasks to be carried out, although it follows that there will probably be some management, hardware, software and quality assurance work. These blocks are then broken down into sub-tasks until the project manager is satisfied that all relevant tasks have been identified.

A WBS, then, is a top-down technique which makes clear how each piece of work fits within the project as a whole. It has also been suggested that the WBS can be used to accumulate costs-per-task which could then be used as a reference for later projects (Youll, 1990). What a WBS does not show, however, is the relative size or duration of each task. For this purpose Gantt and PERT charts were developed.

Figure 2.1 : Work breakdown structure for a simple project

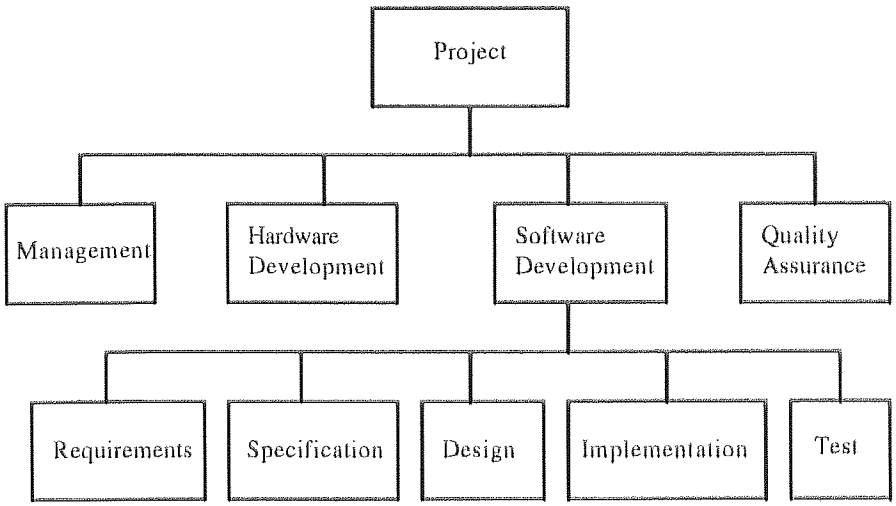
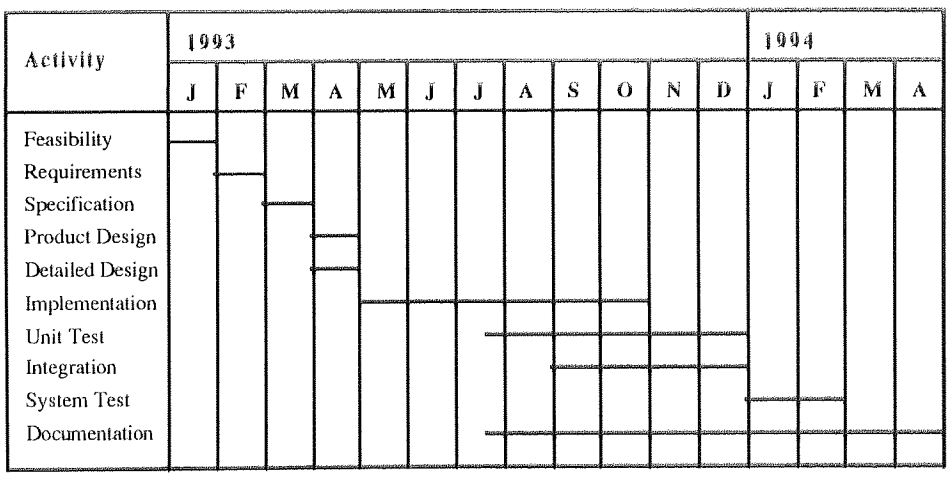


Figure 2.2 : A Gantt chart for a simple project



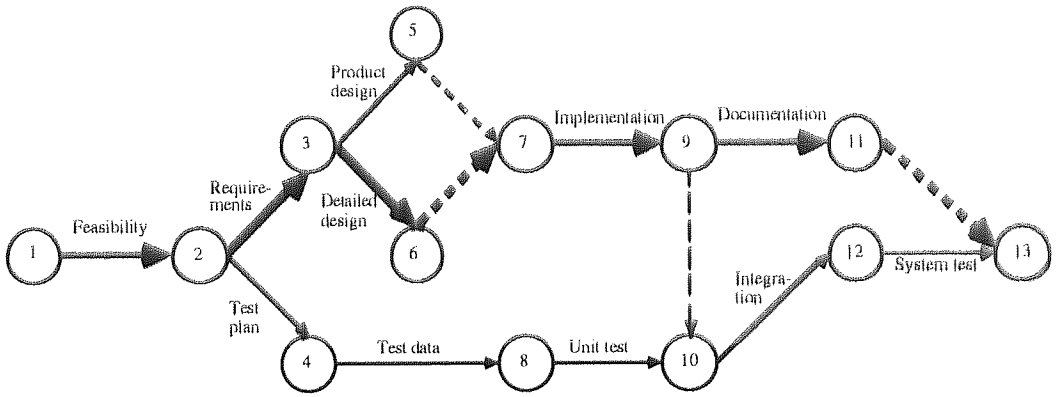
2.2.2 *Gantt charts*

Gantt charts represent a project as a calendar of activities where the start and end dates for each activity are represented as points joined by a line (see Figure 2.2). The Gantt chart is divided into columns representing time periods often headed by month initials (e.g., F=February). The duration of the task is therefore the length of the line from start point to end point (e.g., Documentation is shown as 9.5 months), and thus the progress of the project can be closely monitored. By assigning the number of staff required for each activity (in each row), the total number of staff in any particular month can then be calculated by summing the staff numbers down each

column. This would quickly show the busiest times of the project and when extra personnel may be needed.

However, Gantt charts can provide little assistance in planning or re-planning the actual ordering of tasks and are simply an effective means of displaying the final project plan. Re-arranging the overlap between activities would require the Gantt chart to be redrawn.

Figure 2.3 : Network for a simple project
(critical path shown with bold arrows)



2.2.3 *Project network analysis and PERT charts*

Project network analysis is a technique developed in the 1950s as a means of analysing the sequencing and scheduling constraints of competing project plans. A project is represented as a network of tasks denoted by arcs joining a start-node to an end-node (see Figure 2.3). The start-node denotes a task beginning, the arc represents the task ongoing, and the end-node denotes the end of one task and the beginning of another. Dashed lines are "dummy" dependencies used to make logical dependencies clearer.

Calculating the duration values for all the tasks allows the "critical" (or longest) path through the project to be identified. The critical path is the minimum duration of the project and is shown in Figure 2.3 in bold arrows. For a project to be completed earlier, one or more of the tasks on this critical path must be completed in a shorter duration. This may lead the project manager to buy tools to increase the productivity of certain tasks or assign more staff to its completion. This assumes that the expected durations are accurate.

PERT (Program Evaluation and Review Technique) charts were developed in the mid-1960s and are used to assess the probability and variance of completing tasks within stated durations. The expected duration, D, and variance, V, assigned to each task is assumed to follow a β-distribution (where considerable under-estimates are more likely than considerable over-estimates). By assigning estimates to each task for duration in terms of the most optimistic (OPT), most likely (LKY) and most pessimistic (PES), then the PERT calculations give:

$$D = \frac{OPT + 4 * LKY + PES}{6} \qquad \text{and} \qquad V = \left[\frac{PES - OPT}{6}\right]^2$$

If a task were assigned OPT=1, LKY=2 and PES=3, then D=2.0 and V=0.11. For instance, the project defined in Figure 2.3 could be calculated to have an expected duration of OPT=20 months, LKY=26.5 months and PES=33 months, which would give D=26.5 months and V=4.7 months. This means the project could take from 21.8 to 31.2 months. Those tasks with the highest variance (e.g., Documentation, Implementation, etc.) are clearly the tasks which put the project at risk from completing later rather than earlier. The project manager would then have to expend most of his or her efforts ensuring that these tasks are completed on time. For meaningful results to be deduced from PERT charts, however, it is assumed that accurate estimates have been assigned to each arc. If the values are wrong (and they are only estimates) then the critical path may only be identified as the project progresses and task deadlines begin to slip.

2.3 Techniques for estimation

The project management techniques described in the previous section are able to represent the relationship between tasks but give no indication of their relative size or the project as a whole. There are, however, a number of techniques which attempt to produce such estimates. Boehm (1981) provides a list of seven notable estimating techniques (pp329-343). They are:

1. *Algorithmic models* (producing linear or composite models of factors which influence cost). Such models have been developed since the mid-1960s (e.g., Farr & Zagorski, 1965). They are objective and thus repeatable, but rely on a database which can be quickly out-of-date as new methods and techniques are used. A detailed discussion of these models is given in Chapter 5.

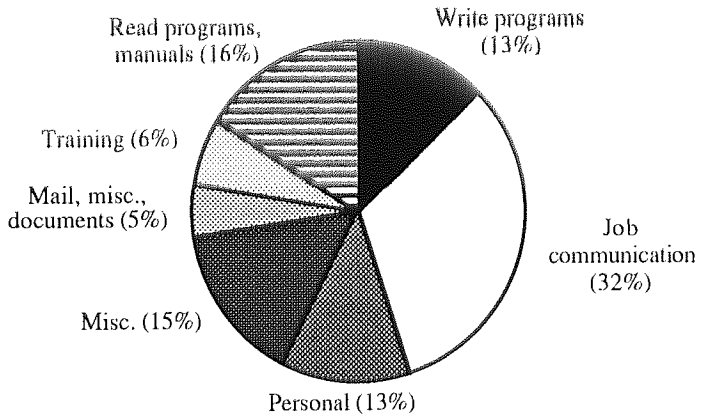
2. *Expert judgement* (using human experience). Means of structuring discussions between estimating experts, such as the Delphi technique, have been developed since the late-1940s. The Delphi technique co-ordinates the discussion between a number of project managers who are asked to estimate (and then explain the rationale of) costing a particular project. Although still seen as a useful technique (e.g., Goodman, 1992), the extent to which this technique would overcome the problems of estimating already experienced by project managers is, however, far from clear.
3. *Analogy with past projects* (based on comparison with past projects). Estimating-by-analogy is widely reported as being the most popular technique (e.g., Heemstra & Kusters, 1991). The question a manager still needs to answer is 'How far is the past project a good representation of the system about to be developed?' This may involve breaking the project down a number of levels before the analogy with a past project can be justified (Wolverton, 1974).
4. *Parkinsonian* (filling the space available). Boehm (1981) criticises this technique on the grounds that by attempting to estimate on the basis of maximum available resources, it is unlikely that the project manager is making the best use of these resources. Since the actual demands of the project are not being addressed there is no real estimating going on, and Boehm suggests that this technique can lead to disaster.
5. *Price-to-win* (deadlines or budget constraints defining the estimate). In this case, the financial rewards, rather than the demands of the project are being addressed. Boehm (1981) points out that as long as companies fail to distinguish between real and price-to-win estimates, this technique may win contracts but few friends.
6. *Top-down* (global properties producing an estimate which are then broken down into constituent parts). The advantage of this technique is that by taking the macro-view, competing under- and over-estimates at a micro-level are allowed to cancel out. The disadvantage is that components which have a significant impact on the project may be overlooked.
7. *Bottom-up* (summing the estimated costs of individual components to arrive at a total system estimate). The advantage of this technique is that the project manager now has detailed knowledge of individual components. The problem is that as the parts are summed to produce a final estimate, the project manager may ignore the cost of integrating and configuring these parts. Specifically, the time spent programming may, in fact, be a small proportion of a programmer's time (e.g., see Figure 2.4). Boehm (1981) suggests that if the estimate is produced by the person who will do the job this will give a certain motivation to its success.

Boehm's own conclusion is that none of the above seven techniques is significantly better than any other. He suggests (p342) that the most effective solution may be to combine a number of techniques, such as:

- top-down estimates using local experts or analogy;
- bottom-up estimation at the component level;
- comparison and iteration between the above two.

Such a combination of techniques is exactly the way companies now appear to be approaching the problem (e.g., Goodman, 1992). But at the heart of estimation is the premise that the development environment is stable enough to allow any technique to be applied consistently. Such stability requires that the process of developing software is understood, and it is at this point that life-cycle models become important.

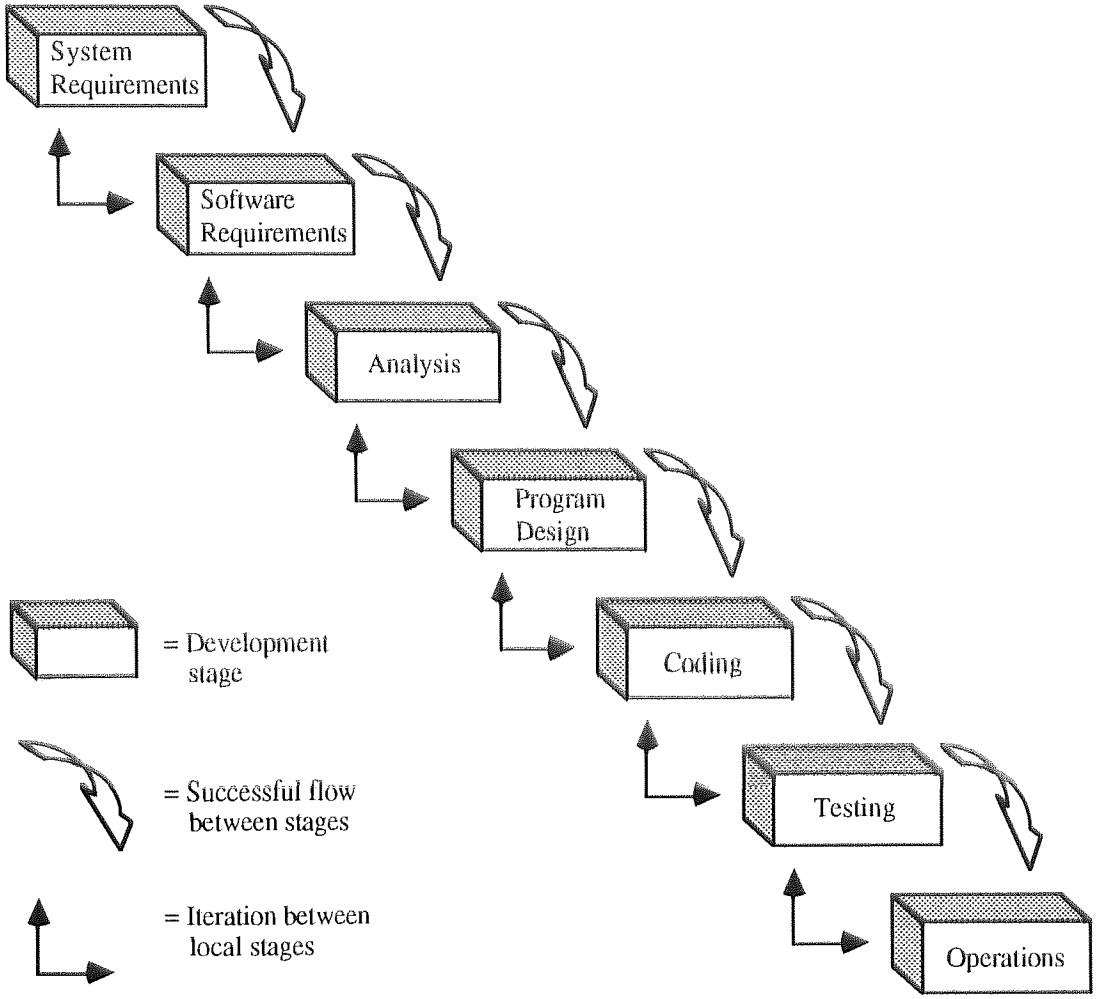
Figure 2.4 : Time spent by 70 Bell Lab programmers (after Boehm, 1981, p341, cf. Bairdain, 1964)



2.4 Models of software development

If software development is to be amenable to study then it is important that the process is structured by a method. A software development method can be defined as a prescription of those tasks (and techniques) which are required for the successful development of a software system. In the same way that a doctor who prescribes a course of drugs must also state how they are to be taken (once a day, twice a day, etc.), the documentation which purports to describe the method must also contain a Methods Process Model (MPM) that explains the order in which tasks are to be carried out. Unfortunately, no method covers every phase of development; few describe the project management tasks which are essential to the smooth running of any project; and, there are rarely any guidelines on how to relate the method to the project at hand (Harris-Jones *et al*, 1993).

Figure 2.5 : The 'classic' seven-stage Waterfall model
(after Royce, 1970, p2; Boehm, 1976, p1227)



It was in response to problems such as these that models were first developed in the early-1970s which sought a deeper understanding of what goes on during software development. To use the medical analogy again, these models attempted to provide the software developer with the anatomy and physiology of the development process. The classic model of the development process is the seven-stage Waterfall model (see Figure 2.5). Although the stages are sometimes renamed and others added (e.g., Feasibility before Systems requirements), the general philosophy of the model is that there are a number of logically ordered stages which have a number of tasks and produce uniquely identifiable products. These are:

1. *System requirements* (specifying the overall functionality and performance constraints of the system, delivering an Operational Requirements document).
2. *Software requirements* (defining the overall architecture of the system including control and data structures, delivering a Functional Requirements document).

3. *Analysis* (representing the relationships of components within the system and uncovering how the system needs to work, delivering an Analysis Report/Specification document).
4. *Program design* (detailing the algorithms, interfaces and structures of specific components, delivering a Program Design/Specification document).
5. *Coding* (implementing the program design, delivering Program Listings).
6. *Testing* (verifying the system components do what they are supposed to do, delivering Test Reports and perhaps a final Specification document).
7. *Operations* (a fully functioning system with associated Acceptance Test results and User Guides).

As each stage is completed the accepted products are signed-off by the client and so the project progresses to the next stage. It is generally accepted that the Waterfall cannot strictly follow the laws of gravity since some local iterations may be required, such that, if the testing stage uncovered some faults with the program design then the project would need to backtrack and refine the necessary design sections. In this way, major chunks of completed work are preserved. The model does not allow iterations from, say, testing back to requirements or analysis, since this would entail a more radical alteration to the current system and changes to products already signed-off often some time ago. In this case any control the Waterfall model has attempted to impose has been significantly broken.

The major weakness with the classic Waterfall model is that only when the coded system is subjected to testing at stage 5 can critical weaknesses in the requirements specification and design be uncovered. Royce (1970) addressed this specific issue by attempting to preserve the basic Waterfall model but adding five further steps (*ibid*, pp4-7):

1. *Produce a preliminary program design before analysis is complete.* This removes the possibility of a system failing storage, time and data handling constraints. These constraints are regardless of the final functionality specified by analysis and so, is independent of it.
2. *Carry out a 'ruthless enforcement' of all documentation requirements.* Royce calls this the 'first rule of managing software development (p5)'. Only then can there be a good understanding of how the system is meant to work.
3. *Ensure that, if the system is new, the critical functions delivered are actually the second version developed.* The first (pilot model) version involves carrying out the whole development cycle in miniature and so allows trouble spots in the design to be identified and some key functions to be tested early.
4. *Pay particular attention to the testing phase.* It is this stage which puts the project most at risk. Royce suggests that the majority of errors can be found by visual

inspection but he also advocates testing each logical path at least once before releasing the product to the customer. The previous three steps should have minimised the actual number of errors present.

5. *Commit the customer to accepting that the correct interpretation of the requirements are being implemented.* This 'formal and continuous' involvement should also be throughout the life cycle, from the earliest development stages until the system is finally accepted.

Royce's conclusion is that although these extra steps will add to the cost of developing software, they provide a significant improvement over the classic Waterfall and are necessary if software is to be developed successfully.

The Waterfall model itself has often been criticised on the basis that it does not realistically represent the software development process (e.g., Gladden, 1982). McCracken & Jackson (1982) even go so far as to condemn the concept of software life-cycles by saying that the very idea creates a rigidity in the development process. For them, systems in the real world are continuously changing as the customer's needs and requirements are explored and understood. McCracken & Jackson (1982) give the example:

> "... the conventional life cycle approach might be compared with a supermarket at which the customer is forced to provide a complete order to a stock clerk at the door to the store, with no opportunity to roam the aisles - comparing prices, remembering items not on the shopping list, or getting a headache and deciding to go out for dinner. Such restricted shopping is certainly possible and sometimes desirable - it's called mail order - but why should anyone wish to impose that restricted structure on <u>all</u> shopping (p31, original underlining)?"

An answer to their question would seem to be relatively straight-forward: If the customer wishes to take their time and is prepared to pay for changes and work which is made redundant, then there <u>would</u> be no problems. But this is rarely how large software development is carried out. A client 'roaming the aisles' is more like a project put out to tender where the client then selects from a number of supermarket/developers. Once a price is agreed, the client can quite rightly object if the final system turns out to be more expensive. Likewise, however, the developers may object if the client continuously changes their mind about what they meant when they said they wanted "such and such" a system.

To continue the supermarket analogy, if the customer and the stock clerk do not communicate well, then there is quite likely to be arguments and problems when the basket of groceries is delivered. But does this mean that life-cycles are the cause of this failure or that their continued use perpetuates the problem? Apparently so, since McCracken & Jackson (1982) continue that:

> "The life cycle concept rigidifies thinking, and thus serves poorly as possible the demand that systems be responsive to change (p31)."

McCracken & Jackson insist that the logic and language of the life cycle concept cannot incorporate prototyping or strategies of re-analysis/re-implementation in order to help the customer and analyst understand exactly what is required. Such techniques have been found to be useful in Decision Support Systems (and KBSs), but cannot be mapped onto the traditional life-cycle model. This is plainly false. Royce's (1970) refined model focused on this need and thus addresses the very problems set out by McCracken & Jackson. More recent models have also incorporated this point, in particular, Boehm's (1988) spiral model. The use of prototyping is often seen as an essential part of developing KBSs but it has also recently been seen as having a useful - if not essential - part to play in conventional/knowledge based systems' development (e.g., Macleish & Vennergrund, 1986; Bader, 1988; Partridge, 1990).

Installing a method of developing software which tackles exactly these issues, however, can turn out to be more difficult than may at first be thought. During the period 1989-1990 Avison *et al* (1992) were brought in by the telecommunications company BT Fulcrum (now Fulcrum Communications) with a view to interfacing a system (called SCOUR) which tested up to 180 000 telephone lines per night with a UNIX-based M6000 computer. This would allow the SCOUR test results to be more easily assimilated, analysed and made available. The solution involved introducing a number of methods and techniques into the organisation where no methodology had been used before. Avison *et al*'s view of a method is not that of a set of strict rules which must be followed at all times, but as a framework within which useful tools and techniques can be easily assimilated. A number of different techniques were therefore used as the situation demanded. However, the methodology-based approach itself received some hostile reaction, Avison *et al* (1992) saying:

> "... the approach was seen as 'academic', required too much effort 'up-front' and increased the danger of slippage in time-scales. From the developers' point-of-view, they had their own traditional ways of developing computer applications which often consisted of a specification in natural language which would then be transformed into code. They were resistant to the new techniques. Further they needed to be convinced that this approach was better than the approaches which were more familiar to them (p137)."

Why this hostility? Are McCracken & Jackson (1982) right after all? Avison *et al* deny that the problems lie in the fact that a number of techniques were being used rather than a single, whole method. In a real sense, Avison *et al* are following McCracken & Jackson's edict that any development methodology must be flexible and tuned to the actual demands of the situation. Perhaps, then, it is an inherent fact that any changes in the way software is

developed will always receive a "better the devil you know" response. If so, then only under the strongest pressure (or enlightened visionary) will a method, tool or technique be introduced into current software development departments or companies. This would be a bleak prospect indeed and demands that the techniques and tools that are used show themselves to lead to notable improvements, while those that do not require considerable investment are more likely to be taken up quickly. Such problems would also affect the type of software metrics and models which should be developed. It would seem foolish to develop further techniques if they are too costly or fail to lead to improvements. The question, then, is whether the software metrics and models which have been developed do indeed have these important strengths. This will be the subject of the next two chapters.

3. Software Development Metrics

"We don't measure software just to watch for trends; we look for ways to improve and get better... A medical doctor would never prescribe medicines or therapies to patients without carrying out a diagnosis first. Indeed, any doctor who behaved so foolishly could not stay licensed (Jones, 1991, p185)."

Summary: *A description of the nature of metrics as a measurement of the software products or the development process itself is given. Particular attention is paid to the "classic" metrics that measure size (Halstead's Software Science and Function point analysis), and structure (McCabe's cyclomatic complexity and Henry and Kafura's data fan-in/out). It will be argued that although size and structure metrics may form the basis of any metrics programme, it is not at all clear that well-defined counting strategies for such metrics have been established. As such, the evidence for the usefulness of metrics for conventional systems' development remains in doubt.*

In software development terms, a 'metric' can be defined simply as the measurement of some property of the software product or development process which is seen as being indicative of some other (harder to measure) property such as cost, quality and maintainability. Knowing the cost of a project helps with resource planning, while knowing the quality and maintainability of the products developed helps decisions about rework of existing program code and levels of support once the product has been released. Ideally, these metrics would have a meaningful relationship with the system size and structure from which a project manager can plan the manpower and resources required.

The temporal distance from (say) the earliest stages of the Waterfall model when the project manager needs to make the estimate, and the point at which the attributes are directly measurable can often cast doubt on the ability of any measurement program to provide useful estimates. For instance, while the literature on software metrics can be traced back to the early-1960s, it took over twenty years before industry seemed to investigate the use of metrics as a serious project management tool (e.g., Grady & Caswell, 1987). There are now a number of European-funded ESPRIT projects which have sought to develop and promote the use of software metrics in industry, such as REQUEST (P300), MUSE (P1257), MERMAID (P2046), COSMOS (P2686), PYRAMID (P5425) and AMI (P5494).

But do metrics really capture anything useful or important? The answer is sometimes given that a metric is a means of quantifying a management problem which has been framed as a question (Basili & Rombach, 1988). This Goals→Question→Metric (GQM) paradigm is the means by which the relevance of the metric result is meant to be interpreted. But interpreting the result of a metric can be a matter of debate and has led to the abandonment of at least one attempt to build an expert system to interpret metric data (Ramsey & Basili, 1989). In what sense, then, can an approach such as the GQM paradigm explain the meaningfulness of a metric?

If the goal were to increase productivity, then the question might be "What is our current productivity?" and the suggested metric might be to measure the number of lines of code produced each month. But how would a project manager validate the fact that the question correctly frames the desired goal and that the metric reasonably represents a measurement of the proposed question? If the goal were to increase product quality, what question would suffice to frame this goal? Questions about productivity may be relatively easy to frame, but "What is our current quality standard?" would simply beg the question "What is quality?" Does it have something to do with maintainability, adaptability, the code being error-free, or something else? If quality were equated to maintainability, what metric would reasonably capture this property? The number of bug fixes/changes? But what happens if the code is so complex that no-one dares implement any change? It appears that there as many questions about the efficacy of applying metrics as there are management questions to be answered by them. Although a "goal" may explain why the "metric" was devised, it still remains problematic whether the goal and metric are always related to the same thing (such as "quality" and "number of bug fixes"). This problem remains an open issue.

But software development is not only a complex and difficult task, it may also be difficult to accept that any measurements can be used to make the process any more understandable. Software development is an essentially intellectual, human activity and it is this very point which makes an analysis of the development process peculiarly difficult. Browne & Shaw (1981) point out that the biggest difference between models of the physical sciences and of software is that the former can be tested against reality, while for the latter the programmer is creating that very reality. In principle, therefore, the programmer can produce exactly the reality predicted by any model of software development that is proposed! This is amply demonstrated by Weinberg & Schulman (1974) who found that development teams could optimise whichever attribute of a program asked of them (e.g., shortest time to develop, best user interface, etc.).

If software development were truly as malleable as this then it would be beyond any scientific study or control. Indeed, there is at least one suggestion that there are some things

about software development which can never be measured in any useful sense (Shepperd, 1991, p78). Browne & Shaw (1981) are not so pessimistic but sound the cautionary note:

> "In establishing a hierarchy of models, we make the assumption that natural causality corresponds to our hierarchy of simplifications. Whether this assumption is entirely valid can be debated, but our limited intellectual capacity *forces* us to make it; without such an assumption, we could not cope with the complexity surrounding us (p22, original italics)."

By attempting to measure software development, then, we begin first with the assumption that there are invariant principles which exist within the process and which are therefore open to scientific study. The need to make this assumption is reinforced by DeMarco (1982) who states "you cannot control what you cannot measure", and this is the central theme of metrics: control through measurement. But what needs to be measured? It is assumed here that if cost and quality are the two key concerns of management, then metrics for size (to deduce cost) and structure (to deduce quality) are the two most important metrics. But how is the size or structure of a system or piece of code measured? The rest of this chapter will answer this question for conventional data-processing (DP) metrics, and, in particular, describe:

- metrics for measuring the size of software systems;
- metrics for measuring the structure of software programs;
- attempts to axiomatise a description of a "metric".

3.1 The size of a software program

The size of a software system provides vital project management data in terms of giving the project manager the ability to assess the demands of the project as a whole. The bigger the system, the more resources will be required in its development. If the system is too large for the developing department then questions of sub-contracting, modular development, or even cancellation can be more properly addressed. System size, therefore, is one of the most important attributes to be estimated early in the life of a project. A common measurement of system size is lines of code (LOC), which is sometimes taken as the baseline against which better measurements of size can be judged (e.g., Basili & Hutchens, 1983).

The problem with LOC is that it may be difficult to agree what a "line of code" is. For instance, Jones (1978, 1986) suggests there are eleven major counting methods, the first six are broad definitions of a LOC (at the program level) while the last five take into account the need to identify only those LOC which are the subject of work within a project (at the project level). The program level definitions given by Jones (1986, p15) are:

1. count only executable lines;
2. count executable lines and data definitions;
3. count executable lines, data definitions and comments;
4. count executable lines, data definitions, comments and Job Control Language (JCL);
5. count only physical lines (as they appear on an input screen);
6. count only physical lines ended with a logical delimiter (such as ";").

When these different counting procedures were applied to a single program written in Basic, Jones (1986) found that the LOC count ranged from 360 to 1500 (a difference of 4.2:1). Whichever of these program definitions is adopted, the project level definitions are then (*ibid*, p15):

7. count only new lines;
8. count new lines and changed lines;
9. count new lines, changed lines and re-used lines;
10. count all delivered lines and any temporary "scaffold" code constructed during the development process;
11. count all delivered lines, temporary code and any other support code.

Definitions 7-11 are important since they indicate which lines of code are to be identified with a project, and thus, are crucial for calculations such as productivity and cost-per-LOC. As can be seen, however, there are a number of competing alternatives which can breed confusion and cast doubt on the ability to compare results between companies or even departments. One could even query what is meant by "new" or "changed". If LOC is more problematic than might have been presumed, then, what other means of measuring system size are there? Although other "sizing" methods exist (e.g., DeMarco's "Bang" metric), the most well-known alternatives are Halstead's Software Science and the function point method of Albrecht & Gaffney and Symons.

3.1.1 *Software Science (Halstead, 1972, 1977)*

One of the first attempts to formalise a science of software and deduce equations for program size (in terms of its length) and development effort was put forward by Halstead (1972, 1977). The most important elements in this science is the nature of the program and the ability of the programmers. A "program" is defined in terms of operators and operands; the "programmers" are defined in terms of their memory capacity and ability to "chunk" information. By focusing on just these properties, Halstead suggests, we have an objective means of measuring a program. First, a family of measurements are defined (Halstead, 1977, p7):

η_1 = number of unique operators
N_1 = total number of operators
η_2 = number of unique operands
N_2 = total number of operands
η = $\eta_1 + \eta_2$ (vocabulary)
N = $N_1 + N_2$ (the sum of all program characters)

These can now be applied to a program. For instance, consider an implementation of Euclid's algorithm in Algol provided by Halstead (see Figure 3.1). A table of operator and operand counts is also given in Figure 3.1. Halstead denotes j as the rank of the most frequently occurring operator/operand, while $f_{1,j}$ denotes the frequency of the jth operator, and $f_{2,j}$ denotes the frequency of the jth operand. But what rules govern the count of operators and operands?

Figure 3.1 : Euclid's algorithm and associated Halstead-table (cf. Halstead, 1977, p7)

```
            IF ( A = 0 )
LAST:       BEGIN GCD : = B ; RETURN END ;
            IF ( B = 0 )
            BEGIN GCD : = A ; RETURN END ;
HERE:       G : = A / B ; R : = A - B x G ;
            IF ( R = 0 ) GO TO LAST ;
            A : = B ; B : = R ; GO TO HERE
```

Operator	j	$f_{1,j}$	Operand	j	$f_{2,j}$
;	1	9	B	1	6
: =	2	6	A	2	5
(..) or BEGIN..END	3	5	0	3	3
IF	4	3	R	4	3
=	5	3	G	5	2
/	6	1	GCD	6	2
-	7	1			
x	8	1			
GO TO HERE	9	1			
GO TO LAST	10	1			
	η_1=10	N_1=31		η_2=6	N_2=21

Defining a counting strategy is one of the major problems in Halstead's description of Software Science (e.g., Elshoff, 1978; Salt, 1982). Halstead himself suggests that recognising an operator and an operand is 'intuitively obvious'. For instance, with 'a bit of reflection' it will be seen that a pair of parentheses is one operator. Further, a BEGIN...END statement, performing an identical function, must also be regarded as the same operator. Labels, on the other hand, may or may not be counted, Halstead (1977) suggesting:

> "Since the labels HERE: and LAST: are neither variables nor constants, they are not operands. They must, therefore, be operators or parts of operators. The combination of the instruction GO TO HERE and the label HERE: determines program flow by determining a program counter or text pointer; consequently, the combination is classified as one operator. An unused label, on the other hand, is treated as if it were only a comment, hence not essential to, or part of, the program (pp7-8)."

Delimiters which determine flow - such as the semi-colon in Figure 3.1 - are counted as operators. Similarly, control structures such as IF, IF...THEN...ELSE and DO...WHILE are also considered operators. On classifying operands, Halstead says merely that the table makes this 'intuitively obvious and requires no further explanation (p8)'.

We must surmise, therefore, that an operator performs a function across at least one operand (otherwise it has no use and should be ignored). The scope of an operator can extend across more than one operand and even across other operators, e.g., the parentheses in *(A = B)* extend across the operands *A* and *B* and the operator =. What seems to be important is the function of the operator, not its particular name. Thus, parentheses and BEGIN...END statements are counted as the same, while comments and unused labels are ignored. For this definition to be accepted, however, it must also be presumed that a pointer is made different when it locates a different part of the program; otherwise, GO TO HERE and GO TO LAST would have to be considered the same operator. Halstead seems to embrace this distinction when he notes that:

> "... the ability to define labeled points, like the ability to define new functions, removes any limitation on the growth of η_1 that might otherwise be imposed by the instruction set of a machine, or the design of a language (p8)."

In other words, no matter how small the vocabulary of a language, a programmer can invent a new operator each time a labelled point is created. This is important,

because for larger programs, the (unrestrained) number of operands would begin to dominate vocabulary counts.

With a counting strategy in place (whatever it may be), Halstead goes on to define a number of relationships. The most important of these are summarised in Table 3.1. In most cases, Halstead justifies the validity of the relationship by way of high correlations between expected and actual program measures which ranged (where given) from 0.90 to 0.993. The highest correlations were those between program length, N, and the estimate, N*hat*. The significance of such empirical support is quite clear to Halstead (1977), who asks us to remember that:

> "... for every way in which an algorithm can be implemented in agreement with.. [the length equation].. there are an infinite number of ways in which an equivalent version could be written. This suggests that the human brain obeys a more rigid set of rules than it has been aware of, and that the parameters η_1, η_2, N_1 and N_2 may serve as useful elements in eliciting further relationships (p15)."

A Software Science, in other words, is possible and at its heart is the philosophy of "Man, the symbol manipulator" (Halstead, 1972). In fact, so compelling was this philosophy that Software Science survived nearly a decade before any serious doubts were raised. Early successes suggested good correlations for development time (Gordon & Halstead, 1976), the number of errors in a program (Funami & Halstead, 1976), and even attempted to explain the quality of a program (e.g., Elshoff, 1976; Gordon, 1979). By the late-1970s a large amount of empirical support had been gathered (e.g., see Fitzsimmons & Love, 1978).

But Halstead is clearly confusing high correlations with meaningful relationships (Hamer & Frewin, 1982). Other studies began to cast doubt on the apparent successes of Software Science, with high correlations not being seen between effort and time (e.g., Curtis *et al*, 1979) and where the program length equation was only questionably extendible to other (non-Algol) programming languages (e.g., Basili *et al*, 1983). Experimentation with the program length equation, however, is one area of Software Science which continued well into the 1980s (e.g., Johnston & Lister, 1981; Woodward, 1984; Felican & Zalateu, 1987). More recently, there has been continued interest in the comparison between Halstead's measurement of a program's vocabulary, $N.\log_2\eta$, and one based on Zipf's law (e.g., Prather, 1988; Chen, 1992) where a relationship is hypothesised between the length of a word (operator/operand) and its frequency within the text (program) as a whole.

Table 3.1 : Summary of Halstead's (1977) Software Science

Name	Equation	Meaning
Program length	$Nhat = \eta_1.\log_2\eta_1 + \eta_2.\log_2\eta_2$	The "length" of a program, N, is a function of the number of unique operator names (η_1) and the number of unique operand names (η_2) used.
Volume	$V = N.\log_2 \eta$	The "size" of a program, V, is a function of the minimum length (in bits) to represent each element ($\log_2\eta$) by the number of elements.
Potential volume	$V^* = (2+\eta_2^*).\log(2+\eta_2)$	The "minimum volume" of an algorithm, V*, independent of implementation language, requires only two operators (i.e., one function name and one assignment symbol) and no operand repetitions ($N_2^*=\eta_2^*$). The asterisk denotes a minimum value.
Program level	$Lhat = \frac{2}{\eta_1}.\frac{\eta_2}{N_2}$	The "efficiency of an implementation" can be defined as L=V*/V, and rewritten by assuming program level, L, is related to the proportion of unique operators (η_1^*/N_1) and the proportion of operand repetitions (η_2/N_2).
Intelligence content	$I = \frac{2}{\eta_1}.\frac{\eta}{N_2}.N.\log_2\eta$	"How much" is being measured in an algorithm is a function of *Lhat**V and a measure of the intelligence content, I.
Program-ming effort	$E = \frac{V}{L} = \frac{V^2}{V^*} = \frac{V^{*3}}{\lambda^2}$	"Programming effort", E, is the effort to make $N.\log_2\eta$ mental comparisons (V), and where effort is proportional to the difficulty of the task (1/L).
Program-ming time	$That = \frac{E}{S} = \frac{V}{SL} = \frac{V^2}{SV^*}$	"Programming time", T, is a function of the effort where there are 18 moments of psychological time (S) per second.
Language level	$\lambda = L.V^*$	"Language level", λ, defines a constant relationship between the program level and the potential volume of an implemented algorithm.
Delivered errors	$Bhat = \frac{V}{3000} = \frac{E^{2/3}}{3000}$	The number of "delivered errors", B, is a function of the number of mental discriminations between errors, given that high-speed memory handles five "chunks" to produce a result (giving V*=3000).
Modularity	$M = \frac{\eta_2^*}{6}$	The ideal "modularity", M, of a program is a function of the number of unique input/output parameters, given that only five "chunks" can be manipulated simultaneously to produce a result (giving a total of six "chunks").

For a project manager, however, knowing the size of system once it has been written is of little use. What is more important is to make early estimates of size and the required effort to build a system. Even here Halstead makes a number of dubious intellectual leaps which have been roundly (although sometimes unfairly) criticised (e.g., Lassez *et al*, 1981; Shen *et al*, 1983). Halstead (1977) defines programming effort as:

> "... the mental activity required to reduce a preconceived algorithm to an actual implementation in a language in which the implementor (writer) is fluent (p46)."

While this definition may be accepted, Halstead also requires that the program is "pure" (it has no superfluous logical statements or structures), and that the programmer works with a 'high degree of concentration'. By only going on to study algorithms published in academic journals it becomes unclear how Software Science would then relate to any everyday commercial programming project.

Notwithstanding this problem, Halstead goes on to say that if E is the total number of mental discriminations required to implement a program, then the time it would take to implement that program can be deduced by borrowing from psychology the notion of "psychological time". This concept (denoted as Stroud's number, S) is the moment of time required for the brain to perform 'the most elementary discrimination'. Halstead accepts Stroud's hypothesis that there are "five to twenty or a little less" of these moments per second to define the number of mental discriminations, S, as:

$$5 \leq S \leq 20$$

Halstead takes it that S=18, presumably because this fits the proposed model. There is no explanation as to why this should be so, but it is this number that allows the equation for E to be converted into an estimate of the time to program (see again Table 3.1). Empirical support for this equation was supplied in a series of experiments in which 12 algorithms from the *Communications of ACM* were implemented at random in PL/1, Fortran and APL by a single programmer fluent in all three languages. Using S=18, a correlation of r=0.92-0.94 was found between T and T*hat* (Halstead, 1977, p53). It can also be noted, however, that the actual value of S is effectively irrelevant here, since if correlation is being used as the measure of correctness, then S=9 would produce answers which were twice that of S=18 but the data points between T and T*hat* would have the same coefficient of correlation.

Although Halstead suggests these steps from E to T are straightforward, many controversial statements are made. Principally, Halstead imports theories on memory without taking into account which type of memory is involved in programming. Recent research sees memory as having three distinctly different stages, namely: sensory, short-term and long-term. Which stage does computer programming make use of? By defining effort in terms of the effort to deal with a preconceived algorithm, it must be assumed that Halstead is defining programming as a process of recall from long-term (i.e., memorised) memory. If so, then Coulter (1983) points out (p169) that the use of S does not apply since Stroud's number is specifically confined to sensory memory.

Halstead borrows other theories from psychology in an attempt to deduce equations for the number of delivered bugs, B, and the ideal modularity of a program, M, on the basis of the ability of a programmer to "chunk" about five units to produce a result. For Halstead, a "chunk" is a conceptually unique input/output parameter. But Coulter (1983) questions whether Halstead makes any meaningful use of the concept since it is not clear what such a "chunk" is in programming: a matrix, a vector, or something else? Coulter continues that a "chunk" can only be knowable on the basis of the individual programmer, and so is not a useful concept to use.

However, what seems equally clear is that understanding a program with many lines of code must involve some level of "chunking" (Brooks, 1977; Tracz, 1979); otherwise, programs would now be reduced to a token-by-token description with no basis for abstracting their higher-level meaning. The fact that it may be unclear what counts as a "chunk" for a programmer and how it could be accurately measured does not preclude its use (e.g., Vessey, 1987; Davis & LeBlanc, 1988). Even within psychology, clear definitions of what a "chunk" is can be hard to come by but consensus is being reached. A "chunk" has been described in terms of 'a familiar unit (Miller, 1956)', 'compound structures in memory (Simon, 1974)', or as 'familiar units of information based upon previous learning and experience (Eysenck, 1984, p88)'. But this problem should have been the beginning of further research, not the end of Software Science.

The contribution that Halstead does make to software metrics is that his was the first attempt, and thus, the clearest example of the problems that exist in attempting to define and then extend a model of the programmer's universe. He begins by defining a means of breaking a program down into elementary units which can then be studied. Dividing a program into operators and operands is the first step from the electronic form of the program to a model of what a program is which can then be

studied objectively. The problem is what to do next. For instance, Lister (1982) criticises the program length equation on the basis that for high-level languages - such as Pascal - the size of η_1 tends to tail-off relative to the size of η_2 and thus N*hat* begins to seriously underestimate N. For Pascal, the count of η_1 is deprived of labels used as control-transfer targets since it uses control structures instead. Only by taking the 'counter-intuitive' view that each control structure is a distinct operator does η_1 become large enough to satisfy the length equation. Lister (1982) states:

> "Such a counting scheme, however, is most unappealing. One of the beauties of Pascal is its economy of control structure: it seems counter-intuitive to wilfully disregard this economy in the counting scheme used (p68)."

However, it is not entirely clear whether the point Lister is making here is a criticism or not. If he is saying that a general definition of what counts as an "operator" or "operand" cannot be defined for all languages and must take into account properties of individual languages then this seems entirely reasonable. If he is saying that for Pascal the counting strategy must add one to η_1 for each control structure before N*hat* accurately estimates N, then this would seem to confirm that Software Science can be extended to Pascal. This is hardly a criticism. Since Software Science is a syntactic measure of programs, Lister cannot be suggesting that the counting strategy must make some statement about the (subjective) aesthetics of the programming language, beautiful or otherwise. The revised counting strategy that Lister has provided us with (e.g., Johnston & Lister, 1981) points to the problem of deciding what counts as an "operator" or an "operand", a problem which may not necessarily translate easily from one language to another. But this is far from being a devastating criticism.

The unfair nature of Lister's (1982) criticism is further demonstrated when he goes on to attack Halstead's suggestion that language level, λ, is a constant for a particular language. Given that Halstead defined language level, λ, as $\lambda = L.V^*$ and program level, L, as $L = V^*/V$ then it follows that:

$$\lambda = \frac{V^{*2}}{V}$$

Lister contends that since V^* is dependent on the number of parameters and V is dependent on the complexity of the algorithm, then for λ to be constant for a given language it must be true that:

algorithm complexity α (no. of parameters)^2

Lister (1982) finds this relationship to be 'unlikely (p69)'. In the year before this article appeared, however, a new metric had already been published which - although in opposition to Software Science - suggested exactly this form of relationship. Henry & Kafura's (1981) data-flow metric suggests that:

(design) complexity α (count of data fan-in/out)^2

An unlikely relationship which may in fact be true is exactly that which deserves our attention. The fact that Software Science may not have been fully worked out is indicative of a new immature science attempting to study a complex problem. It can hardly be seen as a mortal blow to the approach. Lister's own conclusion is that the assumptions, goals and areas of application of Software Science need to be made clear and a rigorous methodology put in place. Lister does concede that Software Science may develop in these ways, but it is interesting to note that few studies appeared in subsequent years which attempted to tackle these problems and it can be taken for granted that Software Science was effectively dismissed by the mid-1980s.

The conclusion to be made here, however, is that there is value in Software Science and it rests in the simplicity with which a program is described in terms of operators and operands. The failure of Software Science rests on Halstead's inability to extend beyond the program length equation to more useful models of programming effort and the number of errors within a program. If a count of operators and operands is taken to be a measure of program size (or length), then it can certainly compete with a count of LOC. The question is whether N or LOC is the more useful measure. This point is raised in later chapters.

3.1.2 *Function point analysis (Mk.I and Mk.II)*

The function point method was initially devised by Alan Albrecht in the late-1970s to correct the lines of code (LOC) paradox in productivity. This paradox suggested that if productivity was one of the targets of improvement for which management aimed, then all programs should be developed in low-level languages such as Assembler, since programmers typically produced more lines of Assembler per month than of higher-level languages such as Pascal. But this is nonsense since each line of Pascal is clearly seen to deliver more functionality than each line of Assembler. The solution for Albrecht & Gaffney (1983), therefore, was to rate the delivered functionality (or

'outward manifestations') of an application which is independent of the technology or language of its implementation.

The concept of a "function" is taken from Halstead's program length equation, N. They suggest that the factor $\eta_1.\log_2\eta_1$ can be omitted with only a small amount of error since the amount of data a program processes is a more determining factor than the number of operators. Further, η_2 (the number of unique operands) can be approximated by η_2^* (the minimum number of conceptually unique inputs and outputs), with the added advantage that η_2^* is available early at the design stage once analysis has detailed the external input and output requirements. Albrecht & Gaffney base their confidence on the fact that when the program length equation is reduced to $N\hat{}=\eta_2^*.\log_2\eta_2^*$ or $N\hat{}=\eta_2.\log_2\eta_2$, the correlation between N and $N\hat{}$ for 29 APL programs is still impressively high (0.918 using η_2^* and 0.988 using η_2).

In what also amounts to a partial answer to Coulter (1983), Albrecht & Gaffney describe η_2^* as the overall inputs and outputs to an application which include top-level inputs/outputs (such as screens and files). The function point measure of a program is thus calculated in three steps (see Figure 3.2):

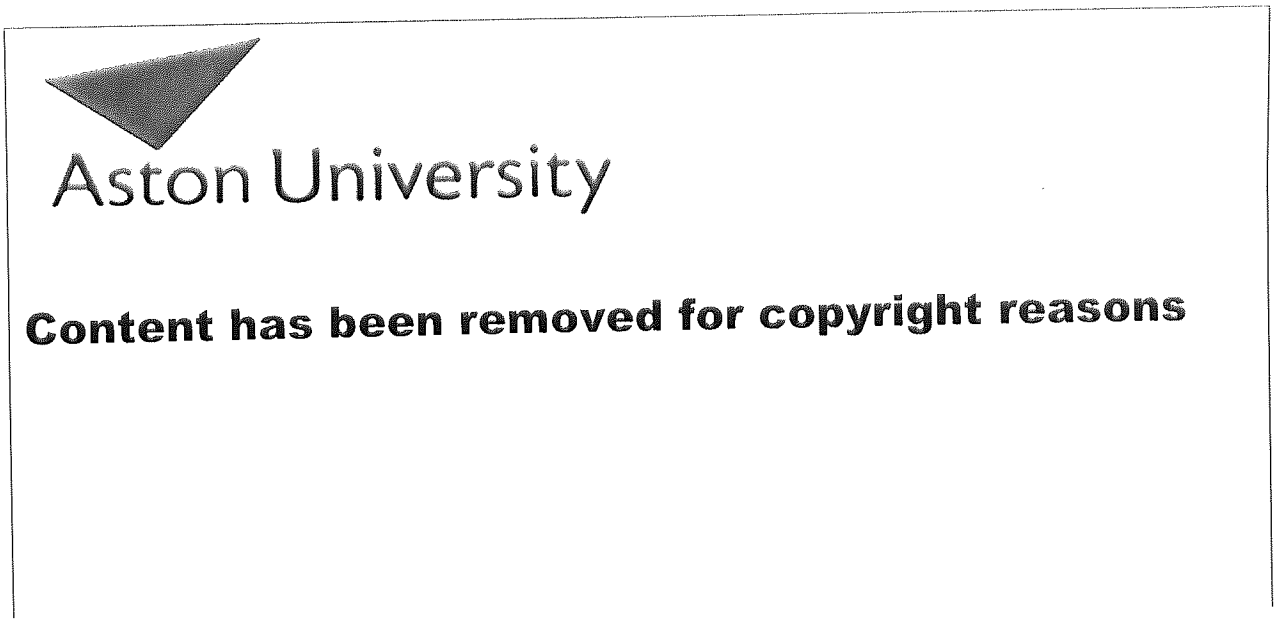
1. Evaluate the "information processing" power of the program by identifying and counting the different types of user functions used.
2. Develop an adjustment factor by rating the "processing complexity" or structure of the application.
3. Multiply the score in '1' by the structure score in '2' to give the total function point score.

The information processing size is found by first drawing a boundary around the application. This boundary identifies those components which are said to belong to the same application and defines what is meant by "internal" and "external". The five user functions defined by Albrecht & Gaffney (1983) are groups of user data or control information defined as types of (*ibid*, pp645-6):

- *external inputs* (those which cross the external application boundary and add or change data in a logical internal file, including input files entered by the user of another application);
- *external outputs* (those which leave the external boundary of the application, including reports and messages to other systems);

- *logical internal files* (those which - from the users' viewpoint - are generated, used or maintained by the application, although those which are not accessible by the user are not to be counted);
- *external interface files* (those which are passed or shared between applications as well as those which enter or leave the application, while outgoing external interface files also count one to the number of logical internal files);
- *external inquiries* (those which are input/output combinations in which a user or application generates immediate output, while inquiries differ from input types because inquiries direct a search for information and input types only change data).

Figure 3.2 : FPA Mk.I calculation worksheet
(cf. Albrecht & Gaffney, 1983, p647)



A user function is said to be unique if it has a unique format or processing logic and is rated as simple (few elements involved), complex (many elements involved), or average (neither simple nor complex). For instance, an external input is rated as simple if it has few data elements, while an external output is rated as complex if it has intricate data transformations and file references. Completing the appropriate calculations in Figure 3.2 then gives the (unadjusted) function count, FC. The processing complexity measure, PC, is a rating of the degree of influence of 14 project characteristics. When summed, PC produces a processing complexity adjustment, PCA, factor given by:

$$PCA = 0.65 + (0.01 * PC)$$

while the total function point measure (FP) is found by:

$$FP = FC * PCA$$

Thus, since PC can vary from 0.65 to 1.35, an adjustment of ±35% can be made to the nominal function point score. Albrecht & Gaffney call this measure a "formula" estimate, since it is based on counting features (rather than tasks) and can be used as a measure of (amongst other things) function points per man-month and man-hours per function point. The principal advantage of this approach is that it gives a measure of system size in terms which are more easily understood by the user than 'lines of code'. Since the five user types are independent of the implementation language, the function point method also promises to measure system size independent of the technology or language of its implementation.

Symons (1988, 1991) criticises what he termed the Mk.I approach on the basis that it will consistently underestimate systems which have high internal complexities. He notes:

- although straightforward, the Mk.I classification system aggregates differences too simplistically such that a component with 100 data elements can score, at maximum, only twice that of a 1 data element component;
- the choice of weight (points per component type) was derived by Albrecht in a "trial and debate" within IBM and as such, it is not clear why (say) interfaces should be weighted as more important than any other input or output;
- while the internal complexity of a system is related to the number of internal files referenced by an external input or output, in the Mk.I approach this can only contribute up to 5% of the processing complexity;

- the 14 complexity factors do not appear to be final and may require some 'reshuffling'.

Symons also takes issue with Albrecht & Gaffney's (1983) suggestion that productivity falls by a factor of roughly three as system size increases from 400 to 2000 FP's. Symons (1988) continues:

> "Most of the above criticisms of the FP method point toward the conclusion that the FP scale underweighs systems which are complex internally and have large numbers of data elements per component, relative to simpler and therefore "smaller" systems. If the criticisms are valid and significant, then the fall-off in productivity with system size may not be as serious as apparently observed. Clearly it is an important issue to resolve (p4)."

Symons' (1988) Mk.II solution begins with the following assumptions:

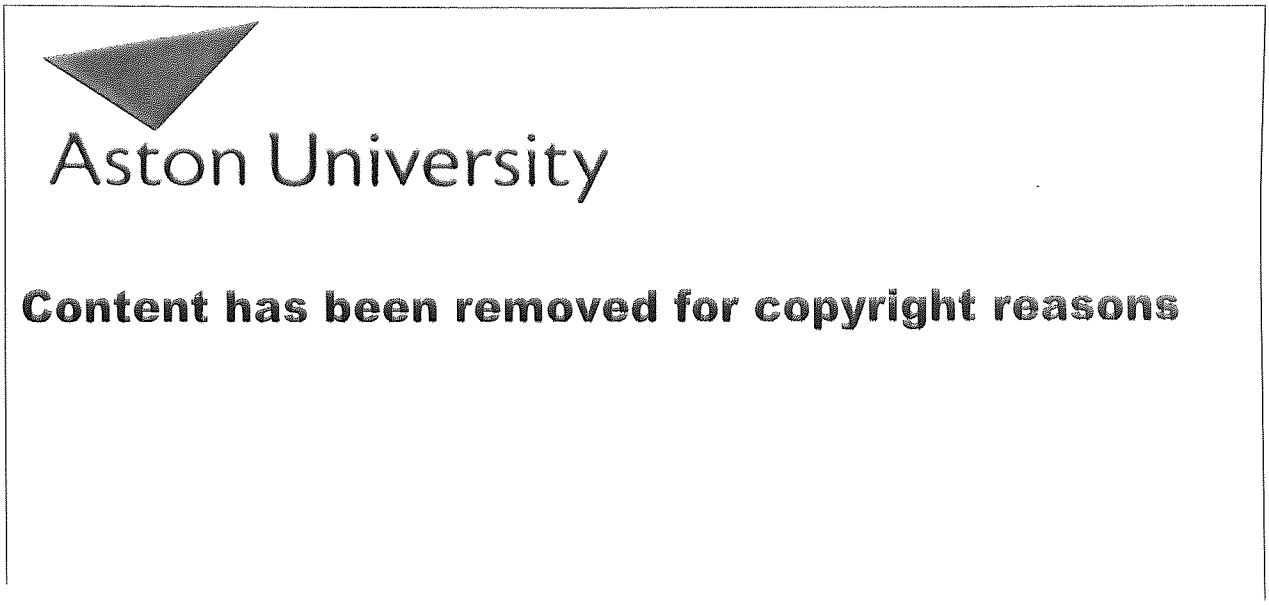
- a system consists of input/process/output combinations (transaction types);
- interfaces are either inputs or outputs (contributing to the overall complexity of a system only if they increase the size of the task);
- inquiries are just another input/process/output combination;
- at the transaction level, the term "logical file" should be considered to be dealing with entities (object, real or abstract) 'about which the system provides information'.

The Mk.II approach thus seeks to replace the data processing functionality of the Mk.I approach with a transactional interpretation of the value of functions delivered to the user. A system becomes a series of logical input-process-output combinations, and Symons (1988) continues:

> "The task then is to find properties of the input, process, and output components of each logical transaction-type which are easily identifiable at the stage of external design of the systems, are intelligible to the user, and can be calibrated so that the weights for each of the components are based on practical experience (p4)."

The logical transactions used by the Mk.II approach are available once an entity-relationship diagram has been produced for the proposed system (e.g., see Figure 3.3), and so, are available early in the development process. The process component is the most difficult component to apply a size parameter to and Symons relies on developing a notion of complexity. This 'complexity' is defined in terms of a count of the number of entity-types referenced by the transaction-type (where "referenced" means created, updated, read or deleted). Symons (1988) admits that the argument is

Figure 3.3 : Example entity model and associated FPA Mk.II breakdown (cf. Symons, 1988, p5)



'tenuous' and the measure 'crude', but suggests that empirical data shows the hypothesis is plausible. The size parameter for the input and output components are taken to be the number of data elements. Thus, the Mk.II formula for the information processing size of an application expressed as Unadjusted Function Points (UFPs) is given as (*ibid*, p5):

$$UFPs = N_I W_I + N_E W_E + N_O W_O$$

where,

N_I = number of input data element types (summed across all transaction-types)

W_I = weight of an input data element type

N_E = number of entity-type references (summed across all transaction-types)

W_E = weight of an entity-type reference

N_O = number of output data element types (summed across all transaction-types)

W_O = weight of an output data element type

The Mk.II function points scores were calibrated by collecting data from a consultancy study in which Clients "A" and "B" were each asked to nominate 6 systems of varying size and technology for assessment. Nine of the 12 systems were developed in time for further analysis to be made. The UFP formula was then deduced by calculating the average "man-hours per count" for input data elements, entity references and output data elements. When these averages were scaled down such that the average system size (in UFPs) were identical for the 8 systems studied under 500 UFPs, this gave a formula of (*ibid*, p7):

$$UFPs = 0.44N_I + 1.67N_E + 0.38N_O$$

Symons then added six further factors to the original 14 Mk.I degrees of influence (DI) components. The additional factors were meant to capture the need to:

- interface with other applications (including technical software, e.g., message switching);
- incorporate security features;
- provide third party direct access;
- meet documentation requirements;
- provide special training facilities (i.e., a training sub-system);
- define, select and install special hardware or software unique to the application (suggestion given by a project representative).

Like Albrecht, however, what counted as a technology complexity factor (TCF) component was a matter of debate. Each system was then scored using all 20 factors and project representatives were asked to estimate the effort devoted to each. In this way, Symons was able to deduce that the effort per DI component and found that the effort was not evenly distributed across the DI factors. In particular, component 19 (documentation) required twice as much effort as any other.

Finally, the Mk.II equation was calibrated against the Mk.I approach by plotting TCF (the 20 Mk.II factors) against TCF (actual). TCF (actual) was derived from the Mk.I processing complexity adjustment (PCA=0.65+[0.01*PC]) factor and given as (*ibid*, p7):

$$TCF_{actual} = 0.65 * \left[1 + \frac{Y}{X}\right]$$

where,

TCF_{actual} = PCA_{actual}
Y = man-hours devoted to TCFs
X = man-hours devoted to in UFPs

The suggestion here is that if Albrecht & Gaffney's representation of TCF is correct, then the relative effort devoted to TCFs:UFPs will be found to be 0.01:1. This would be found by plotting TCF against TCF_{actual} and finding the data points to be on a slope giving a weight of 0.01 per DI. Symons found, however, that 6 of the 8 systems calibrated to the Mk.II method lay on a slope of 0.005 per DI. This suggests that half the effort is used to achieve the 20 TCFs than the Mk.I approach would calculate. The six systems on this second line were all developed by Client "B" and Symons notes that this seemed to be because Client "B" had developed specialist tools to simplify development while some systems used fourth-generation languages. This questions Albrecht & Gaffney's central assertion that the function point method is technology-independent. Symons accepts, however, that the problems of calibration also affects the sensitivity of the Mk.II approach (*ibid*, p9).

What is more worrying is that by being dependent on entity-analysis, the Mk.II approach becomes equally dependent on the clarity of the rules by which entities are identified and counted. This is similar to the counting problems experienced by Software Science. Acknowledging the fact that some subjectivity is apparent when working with FPA, Symons (1988) suggests that 'for some time to come' FPA may have to be supervised by an experienced Function Point Analyst (*ibid*, pp9-10). If this is the case, then one could question whether FPA Mk.II has achieved the original FPA goal of producing a measure of system size which is more intuitive to users than a count of lines of code.

When Symons (1988) says that both Clients "A" and "B" preferred the Mk.II approach (p7), it is unclear whether these Clients are the project representatives who supplied Symons with the FPA data, or the users within the organisations who were

commissioning the systems. If it is the former then FPA Mk.II has failed to address the central theme that function points are meant to measure the user-functionality of the system; while doubts can be expressed about the latter since studies have shown that even FPA analysts can have trouble with the FPA approach.

Table 3.2 : Variations in sizing with function points or source lines of code (SLOC)(after Low & Jeffrey, 1990a, p70)

Program	Estimating method	No. of organi-sations	Develop-ment Language	No. of FPA analysts	Std.Dev / Mean (%)
1	FP Mk.I	7	-----	22	45.5
1	SLOC	1	Cobol	2	72.2
1	SLOC	2	PL/1	12	72.2
2	FP Mk.I	7	-----	7	33.8
2	SLOC	1	Cobol	2	68.8
2	SLOC	2	PL/1	2	80.8

Using specifications for two programs and twenty-two FPA Mk.I analysts in seven organisations, Low & Jeffrey (1990a) found a variance about the mean of more than ±30% between FPA analysts within an organisation, between organisations and between experienced and novice analysts (see Table 3.2). Low & Jeffrey do point out, however, that for the same two programs the FPA analysts were more in agreement when estimating size on the basis of function points (FPs) than on the basis of source lines of code (SLOC). As can be seen, the deviation of FPA scores varied from between 33.8% to 45.5%, while the deviation for SLOC varied from 68.8% to 80.8%. The actual FP or SLOC is not given. While later studies have reported variances between analysts at nearer ±10% (Kemerer, 1992), one could still query how feasible an estimation method is when even the initial counting procedure involves a degree of variance for the same system, especially when the estimate itself is likely to be at variance with the actual system size.

Heemstra & Kusters (1991) found that companies using the FPA method were actually less likely to avoid cost overruns than non-FPA companies, with more than 66% of FPA users having cost overruns of ≥10% on large or very large projects, compared to 50% of non-FPA users. Although the FPA users were much more likely to record data on past projects (96%) than non-FPA users (50%), this appears

to be of little help. The key problems experienced by the FPA analysts were found to be (*ibid*, p234):

- subjectivity in determining the input (28%);
- problems with sizing (26%);
- insufficient insight into the project to determine model parameters (20%).

If the FPA users really did keep records on past projects then presumably they would be able to at least partially overcome these subjectivity/sizing/insight problems by looking back over past data. It would appear not. Heemstra & Kusters also point out that for the most experienced FPA analysts the technology complexity factor (TCF) was virtually always set to 1.00 (i.e., no adjustment made to the UFP score). This leads to the suspicion that those companies using FPA are not using FPA in the way it was initially designed.

These studies suggest that the FPA approach is clearly a difficult method, although it continues to be widely used in both the U.S. and Europe (e.g., Jones, 1991; Betteridge, 1992), while tools are now being developed to assist FPA measurements (e.g., Matson & Mellicamp, 1993). However, the conclusion would seem to be that whilst the notion that the size of a system has something to do with the "amount" of functionality it aims to deliver remains appealing, the concept of a "function" - like that of a "chunk" - remains problematic and one is left with the doubt that FPA Mk.II is a method which is clear or accurate enough to provide project managers with useful information.

3.2 The structure of a software program

Software size alone will not allow a project manager to assess the difficulty and demands of a prospective development project. The missing attribute is that of structure. Structure normalises measurements of size (in terms of lines of code or any other measure), by uncovering the internal complexity of the system. As the FPA Mk.II attempts to make clear, a small system may require as much effort as a larger system simply because the size measure hides the internal complexity of a system. In other words, by generating a more complex arrangement of code the same functionality can be contained within a significantly smaller number of lines of code. Complexity is also seen as the best indicator of error-proneness, or the likelihood of a piece of software to have an operational fault (e.g., Schneidewind, 1979; Dunsmore & Gannon, 1980).

"Good structure" has often been described as the ability of a system to resist errors even when changes are made to some part of its sub-structure (Soong, 1977; Yau & Collofello, 1980). In this sense, the best structure is one with the greatest independence between modules or sub-parts. Using this definition, Ince (1990) traces the literature on structure back to a thesis on sociological order published in 1964. The problem of software complexity was recognised at about the same time with Underhill (1963) cautioning programmers to control the seemingly inherent growth in program complexity over time. For Underhill, "complexity" is the way programs increase their functionality and size over time but does not suggest how best to control this problem. Notwithstanding these exceptions, however, most of the research on complexity in software engineering began in the mid-1970s and focused on generating graphs which represent a change in the control structure or the direction of data within a program. The most well-known of these are McCabe's cyclomatic complexity and Henry & Kafura's data flow fan-in/out.

3.2.1 *Cyclomatic complexity (McCabe, 1976)*

The idea of transforming a piece of code into a directed graph against which to measure "structure" or "complexity" begins with McCabe (1976) who offers an 'intuitive explanation' of how a program can be described in terms of a graph. McCabe's motivation is to show that while structured programs are believed to be easier to debug and maintain, it is not at all clear how complex a program can be before it becomes unstructured. By transforming a program into a graph, a measure of program complexity would then be a measure of the number of paths through this graph. Of course, a program with a backward loop has a theoretically infinite number of loops, so by 'path', McCabe (1976) means:

> "... basic paths - that when taken in combination will generate every possible path (p308)."

Paths are identified by defining a program as a strongly connected graph, G, which has *n* vertices, *e* edges and *p* connected components. The cyclomatic number, v(G), can then be calculated as (*ibid*, p308):

$$v(G) = e - n + 2p$$

where,

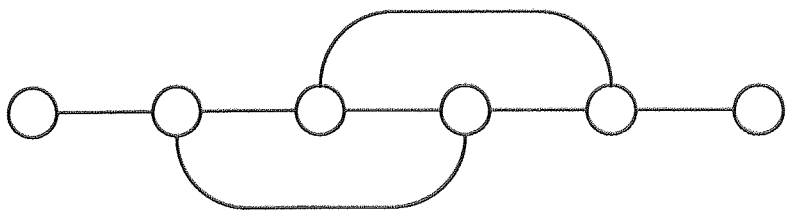
p = 1 (for one component)

The component count, *p*, signifies the number of separate routines within a program. In most studies, *p* is taken to be one. McCabe suggests this formula gives

a measure of the complexity of a program. By abstracting only from decision points, cyclomatic complexity is also independent of the number of functional statements within any given program. The first problem, however, is that McCabe (1976) confuses his terminology of vertices and edges somewhat by attempting to explain the application of this formula in the following way:

> "Given a program we will associate with it a directed graph that has unique entry and exit nodes. Each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. This graph is classically known as the program control graph... [where] it is assumed that each node can be reached by the entry node and each node can reach the exit node (p308)."

In other words, a graph is here reduced to a set of arcs (or edges, *e*) and nodes (or vertices, *n*). When the arcs have an orientation (each points in a particular direction) the graph is called a directed graph. A set of connected arcs which begin and end at a terminal node and where no arc is revisited is called a path. If the terminal nodes are identical, the path is called a circuit. All circuits can be described in terms of combinations of a number of paths. If there is at least one path that visits every node in the graph, the graph is said to be connected. If a node can reach every other node the graph is said to be strongly connected. On the basis of this representation, McCabe suggests that complexity can be defined explicitly as the presence of sub-graphs that branch into or out of a loop (or decision). This branching is isomorphic with the following undirected graph (*ibid*, p317):



This suggests a minimum complexity for nonstructured programs of v(G)=3. The difference for a structured program is that all structured programs can be reduced using the simple rule of removing nodes - one at a time - that have only one input and one output arc. Eventually the graph is reduced to a single node and this is a characteristic of all structured programs.

The equation for cyclomatic complexity can also be reduced to a count of the number of predicates (such as IFs, WHILEs, etc.), plus one. In practice, McCabe suggests that it was more convenient to count conditionals and so the refined measure of cyclomatic complexity becomes (*ibid*, p314):

v(G) = No. of conditionals + 1

Some care needs to be taken over the fact that compound conditionals are actually disguised IFs (e.g., IF "C1 AND C2" THEN.. becomes IF C1 and IF C2 THEN..); while CASE statements are actually multiple IFs, where a CASE statement with N conditions would have to be re-written as N-1 IF statements. The number of areas bounded by the arcs plus one is also given as a quick way of calculating v(G).

McCabe analysed the structure of several PDP-10 Fortran programs using a tool called FLOW (written in APL). Each program was broken down into blocks according to branching statements (IF, GOTO, etc.). From this analysis, McCabe suggests that not only could he begin to recognise individual styles of programming (*ibid*, p313), but it also allowed him to deduce a reasonable ('but not magical') upper limit of v(G)=10. Programmers at TRW were required to calculate v(G) as they worked and rethink any coding if v(G)≥10 (*ibid*, p314). When analysing a tape of 24 Fortran sub-routines for a real-time graphics system, McCabe found a number of programs ranging from v(G)=16 to v(G)=64. McCabe was told by the project members that these programs were notable because they were the 'most troublesome' (*ibid*, p314). There would seem, therefore, to be a relationship between v(G) and reliability. This result also aids testing, since, if a program, P, has a complexity of v but only ac paths have been tested, then one of the following must be true (*ibid*, p318):

- more testing needs to be done;
- the flow-graph can be reduced by v-ac paths;
- portions of the program can be reduced to in-line (i.e., structured) code.

While $ac<v$ then one of the above conditions are not satisfied and so more testing needs to be done. But, McCabe (1976) warns:

> "... this procedure (like any other testing method) will by no means guarantee or prove the software - all it can do is surface more bugs and improve the quality of the software (p318)."

Even though McCabe ends on a cautious note, the idea of transforming a program into a graph of pathways proved to be an appealing and influential concept. The first reactions, however, were to redefine the metric in order to accommodate problems in the way the graph represented certain program structures. New variations of the metric quickly emerged where more emphasis was placed on the bounds of ELSE

statements (Myers, 1977), arcs at a node forming "knots" (Woodward *et al*, 1979), the flow of both control and data (Oviedo, 1980; Stetter, 1984) and the nesting depth of statements (e.g., Harrison & Magel, 1981; Piowarski, 1982).

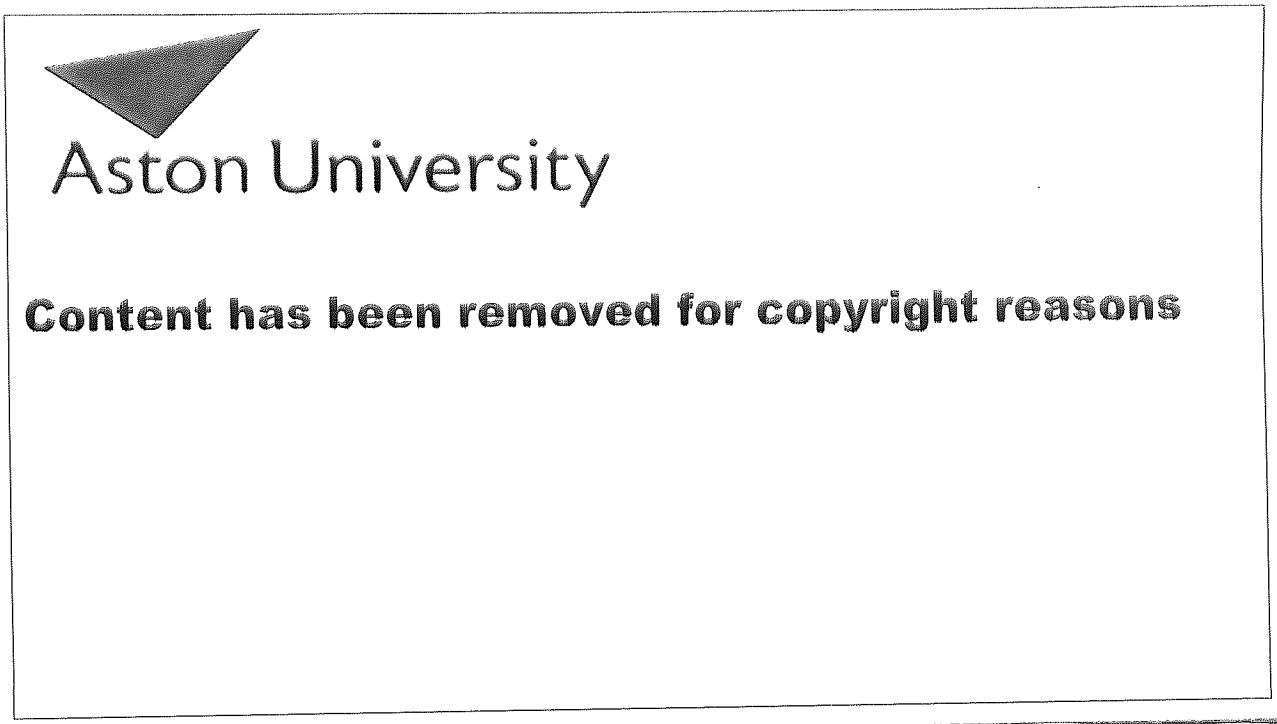
The most important use of these graph-theoretic measures of complexity has been to explain the number and distribution of errors (Kitchenham, 1981; Basili & Perricone, 1984). The complexity of a program has also been related to the difficulty of developing the program (Chen, 1978; Basili *et al*, 1983), the use or reachability of particular nodes (Iyengar *et al*, 1982), while later studies generated similar graphs on the basis of the system design or specification (e.g., Samson *et al*, 1987; McCabe & Butler, 1989).

Similar to Software Science, however, it is sometimes difficult to detect how the elements of the program are to be mapped onto the appropriate metric elements. Perhaps this is why McCabe sought to replace v(G)=e-n+2p with v(G)=No.of conditionals+1. By abstracting what a program is to a graph of nodes and arcs it becomes confusing to know whether we are measuring syntactical or psychological program complexity (Curtis *et al*, 1979). As such, it is no surprise to find empirical support is often contradictory where cyclomatic complexity is sometimes seen as corresponding well with notions of good programming style (e.g., Myers, 1977; Hansen, 1978), and sometimes as not doing so (e.g., Oulsnam, 1979; Baker & Zweben, 1980; Evangelist, 1982, 1983).

McCabe clearly wants to measure both syntactical and psychological complexity, but the bridge from its syntactical measurement to its psychological impact is both unsupported and uncertain. For instance, Prather (1984) notes that the two flow-graphs in Figure 3.4 are both assigned v(G)=4. And yet, if the complexity of testing these two procedures are at the heart of the v(G) metric, then it would seem that the graph in (b) must be seen as being more complex than (a), since all three IF...THEN loops can be tested independently in (a) while all three loops form sub-routines in (b) and cannot be tested independently. Furthermore, Halstead's model of effort, E (see again Table 3.1), has been found to correlate with construction time, the number of errors, and debugging time significantly better than cyclomatic complexity (Davis & LeBlanc, 1988). The lack of support for relating measures of complexity to what programmers see as "complex" and thus being error-prone is a typical criticism of structure metrics (e.g., Shepperd, 1988; Kitchenham *et al*, 1990). The conclusion would seem to be, therefore, that cyclomatic complexity remains an appealing but unproven structure metric.

Figure 3.4 : Two flow-graphs with the same cyclomatic complexity, v(G)=4 (cf. Prather, 1984, p341)

(a) (b)

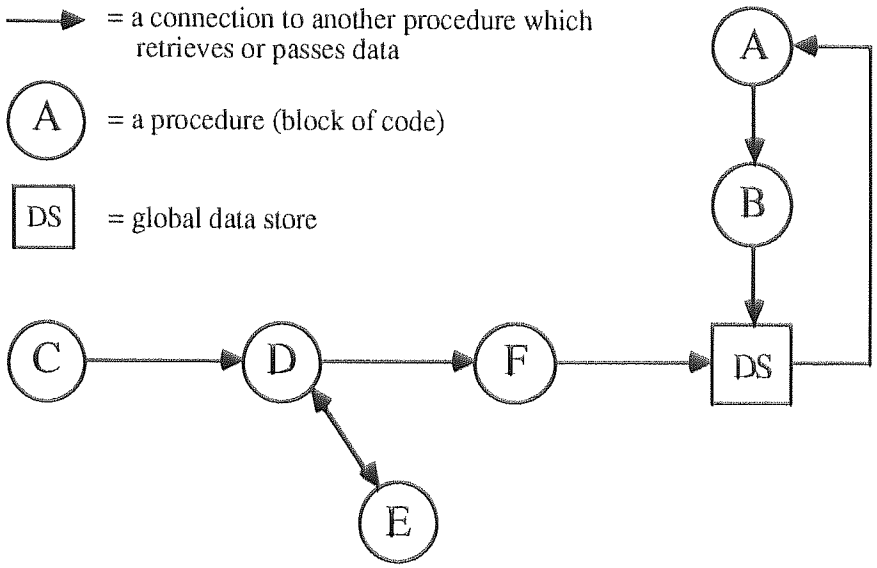


3.2.2 *Data flow fan-in/fan-out (Henry & Kafura, 1984)*

Rather than defining structure in terms of an analysis of tokens within the program (such as Halstead and McCabe), Henry & Kafura (1981, 1984) look to measure the environment created by the program itself. This "environment" is defined by the connections made within the system by information flowing from one point to another. There are two types of "flow". Global flow passes information from module X to module Y via a global data structure DS, such that, X deposits information in DS which Y retrieves. A local flow passes information from module X to module Y either directly (where X calls Y), or indirectly (where X calls Y and Y returns a value that X uses, or if module Z calls X and Y and a value is passed between X and Y). Henry & Kafura (1981) illustrate their terminology by way of an example (see Figure 3.5), and invest the diagram with the following behaviour:

> "Module *A* retrieves information from *DS* and then calls *B*; module *B* then updates *DS*. *C* calls *D* and module *D* calls *E* and *E* returns a value to *D* which *D* then utilizes and then passes to *F*. The function of *F* is to update *DS* (p511; original denotation)."

Figure 3.5 : Simple flow of data diagram (after Henry & Kafura, 1981, p512)



On this basis, Figure 3.5 contains eight information flows, namely: two global flows (B→A, F→A); four direct, local flows (A→B, C→D, D→E, D→F); and, two indirect, local flows (E→D, E→F). The structure of a procedure is then a measure of the fan-in and fan-out of data between individual procedures, where:

- *fan-in* is the number of local flows (direct and indirect) into X, plus the number of data structures that X retrieves information from;
- *fan-out* is the number of local flows (direct and indirect) from X, plus the number of data structures that X updates with information.

Henry & Kafura's definitions are often simplified to a count of the number of connections terminating at (fan-in) or emanating from (fan-out) a procedure. Using this representation of structure, complexity is based on a measure of the procedure's length and connections with its environment. Length is counted as lines of source code including embedded comments but not text preceding the procedure. Since Henry & Kafura suggest that embedded comments are rare, length is therefore roughly equivalent to a count of source statements. The procedure's connections to its environment is a count of the number of connections fanning into or out of the procedure. By assuming these two factors are independent, the complexity of a given procedure, C_p, is then given as (*ibid*, p513):

$$C_p = SLOC * (IFI * IFO)^2$$

where,

SLOC = source lines of code
IFI = information fan-in
IFO = information fan-out

The *IFI*IFO* term represents the input-output space, and the power of two represents the hypothesis put forward (by Brooks, 1975) that communication is non-linearly costly. Henry & Kafura suggest that since SLOC is a weak factor in the equation, potential flaws in a design can be evaluated early allowing for relatively inexpensive cycles of design-evaluate-redesign. Such design flaws are identified when a procedure has a high fan-in or fan-out. Although Henry & Kafura give few clues as to what is meant by "high", they suggest a high number of connections between a procedure and its environment would indicate that it is:

- overly complex (performing more than one function);
- a potential stress point (which would make debugging difficult, given its influence on other procedures);
- requires further decomposition into two or more smaller procedures.

The validity of these suggestions was tested by studying the development of the UNIX operating system. The aim of the study was to see how well procedure complexity correlated with the likelihood of a change being made. Without defining what is meant by "a change", Henry & Kafura studied 165 procedures of which 80 had changes, and which ranged in C_P from 4 to 27 432 000 and in LOC from 3 to 180. Given these very high upper limits Henry & Kafura ordered the data into powers of 10 (e.g., 10^0, 10^1, 10^2, etc.)(see Table 3.3), and found a ranked correlation between C_P and number of changes of r=0.94 (p=0.02). They also noted, however, that 53% of the procedures were less than 20 lines of code and contained only 28% of the errors. By investigating the power of the SLOC and IFI*IFO parts of the equation, Henry & Kafura found that while SLOC did not correlate well with number of changes (r=0.60, p=0.08), the IFI*IFO squared component did (r=0.98, p=0.03). The conclusion seems to be that SLOC is superfluous to the calculation of C_P. Indeed, IFO alone has sometimes been taken to be the best measure of complexity (Card & Agresti, 1988).

The most complex procedure in this study (called NAMEI) did not undergo any changes, however, and all Henry & Kafura tell us about this is that NAMEI has many connections and would be difficult to change. In other words, nobody dared to change this procedure!

Table 3.3 : Information flow complexity and number of changed procedures (after Henry & Kafura, 1981, p516)

Order (10^x, x=)	Number of procedures	Number of changed procedures
0	17	2
1	38	12
2	41	19
3	27	19
4	26	15
5	12	11
6	3	2
7	1	0

In a later article on the same UNIX study, Henry & Kafura (1984) suggests that NAMEI was clearly a design problem since it was relatively long (LOC=155), while fan-in and fan-out were high (IFI=13 and IFO=21). By identifying and separating out the different logical functions NAMEI dealt with, Henry & Kafura (1984) suggest that the combined complexity of this procedure could be reduced by 98% to C_P=20 776 (p571).

However, while other studies have continued to find good correlations between information flow and maintenance effort (Kafura & Reddy, 1987) and specified complexity and implemented complexity (Henry & Selig, 1990), the suggestion that limiting the size of a module will reduce the likelihood of change has not always been supported (e.g., Card *et al*, 1986). Like many of the metrics described in earlier sections, it is not always clear what is being measured by fan-in/out and how the metric is meant to capture complexity. Kitchenham *et al* (1990) suggest that code rarely fits neatly into Henry & Kafura's definitions of flow, while the notion of indirect local flow is imprecisely defined and remains ambiguous. Being a composite metric, programs returned values of zero if they failed to have at least one fan-in or fan-out, thus concealing the effects of the other components in the equation. Kitchenham *et al* (1990) conclude:

> "It would therefore seem preferable to use design metrics based on primitive counts rather than synthetics, unless it is very clear how the values obtained from the synthetics may be interpreted (p58)."

What is strange about the information flow metrics is that a metric which purports to be more sophisticated than the simple token-counting metrics of Halstead and McCabe should at first include and then dismiss program length from the equation. If "flow" is about communication, what contribution was this measurement of length supposed to add to the concept? The result of this confusion is that if the program has only one input and one output (i.e., it is a straightforward sequential piece of code), then the flow complexity is reduced to a measurement of its length. Adding further sequential code would, in this respect, add to its complexity. This result is in opposition to Henry & Kafura's definition of flow as a structural (not a token-counting) metric. This was perhaps their motivation in showing SLOC was a weak part of the equation.

But length is a necessary part of a program's character: the greater its length the greater the risks of errors (typographical mistakes), improper functionality (incorrect design) and inefficient coding (poor style). The amount of communication seems also to be important because it portrays the dependence of a procedure on the system as a whole. What is less clear is how well a metric which can return a zero value even if the procedure is hundreds of LOC long, or explodes into millions with small increases in data flow can capture this notion of complexity.

3.3 Axiomatising metrics

Most of the problems associated with the "classic" size and structure described in the previous section can be traced to three critical problems:

- the entities to be measured are not precisely defined, and thus, the counting strategy by which the metric is applied is also vague;
- while there is confusion over the counting strategy, it is difficult to understand what the metric purports to measure and how this relates to such things as "size", "structure" and "complexity";
- if the concepts are confused, then it is difficult to know whether the empirical support for and against the metric is actually measuring the same entities.

As Fenton (1991) points out, while the concept of a metric has a precise definition in mathematics, its use in software engineering is much looser, and thus, more problematic. Fenton describes the definition of a metric in mathematics as the function $m(x, y)$ by which the distance between two objects x and y is measured. Such metrics have three important properties (p59):

1. $m(x, x) = 0$ for all x;
2. $m(x, y) = m(y, x)$ for all x, y;
3. $m(x, z) \leq m(x, y) + m(y, z)$ for all x, y, z.

These three statements are axioms which define the mathematical notion of ('real') metrics. By the mid-1980s attempts were also being made to define a set of axioms which describe those properties a software metric must have in order to be a software metric.

An axiom can be defined as a statement which cannot be proved but leads to absurdity if denied. An axiom, therefore, is a core belief which must be taken as self-evident (such as Descartes' *cogito ergo sum*), and is the root from which other statements of truth can be logically derived. These axioms are derived by setting out what is knowable about a widely-held but troublesome concept and dismissing all those statements which could, conceivably, be false. The remaining statements form the axioms from which the troublesome concept is rebuilt and clarified. For instance, Descartes stripped away all suppositions about what ensures a person knows he exists and was left with only one axiom:"I think, therefore I am."

Given the problems discussed in §3.2 over what is meant by "complexity", it is perhaps not surprising to note that this (vague and troublesome) concept has been the prime target of the axiomatic approach. The first attempt was put forward by Prather (1984) who contended that a structured program is one which is built up 'inductively' from a number of simple statements (i.e., BEGIN..END, IF..THEN..ELSE and WHILE..DO constructs). On this basis, a proper measure of program complexity metric, m, would have to satisfy three inductive axioms (*ibid*, p342):

1. The complexity of the whole is greater than or equal to the sum of the complexity of the parts, given as

$$m(\text{begin } S_1; S_2; \ldots; S_n \text{ end}) \quad \geq \quad \Sigma\, m(S_i)$$

2. Twice the sum of the complexity of a section is greater than or equal to the complexity of any IF..THEN construct to that section, and is also greater than the sum of the complexity of the section parts, given as

$$2(m(S_1)+m(S_2)) \quad \geq \quad m(\text{if } P \text{ then } S_1 \text{ else } S_2) \quad > \quad m(S_1)+m(S_2)$$

3. Twice the complexity of a section is greater than or equal to the complexity of any WHILE..DO construct to that section, and greater than the sum of the sections parts, given as

$$2m(S) \quad \geq \quad m(\text{while } P \text{ do } S) \quad > \quad m(S)$$

The multiple of two in axioms 2 and 3 is based on the assumption that the complexity of the predicate, P, in the left-hand side of the inequality must also be calculated and added to the complexity of the section which contains it. Following axiom 1, however, the complexity of P cannot exceed the complexity of the section. In other words, an IF..THEN or WHILE..DO construct can contribute no more than half the complexity of a section.

Prather goes on to show that a new testing procedure, μ, can be defined which overcomes some of the inadequacies of the way McCabe's metric deals with nesting. The assumptions are made that any bottom-up testing methodology will have assigned units of complexity to small parts of the program, and so, parts in sequence order have a complexity which is additive, while parts which are nested are multiplicative. By further assuming that the complexity of a part of a flowgraph is equal to the maximum complexity of any one sub-flowgraph, Prather also goes on to compute a maximum complexity of a (GOTO) flowgraph of μ=100 and adds this notional maximum to the other accepted limits of v(G)=10 and $N.\log_2\eta$=50 (*ibid*, p347).

Fenton & Whitty (1986) criticise Prather's approach on the basis that he takes structuredness to correspond to a finite set of IF..THEN and WHILE..DO constructs. Fenton & Whitty see this as an unnecessary restriction and argue that:

> "... programs do not fall into just two categories, 'structured' or 'unstructured' (as assumed by many researchers in this area), but all have a quantifiable degree of structure characterised by the hierarchy of basic structures on which it is built (p330)."

The attack, here, is on Prather's second and third axioms which assume only IF..THEN and WHILE..DO constructs exist within a program. Instead, Fenton & Whitty propose that a program mapped onto a directed graph (called a digraph) is equivalent to a set of nodes, x_1 to x_n, structured either in sequence or nested. Thus, a digraph D_1 (say) is composed of and can be reduced to a sequence of non-sequential sub-flowgraphs F_1 to F_4. Fenton & Whitty's contention is that there is a finite set of digraphs and that the structure of all flowgraphs can be unambiguously reduced to a number of these basic components. A flow-graph which is non-sequential (i.e., it has some structure) and contains none of the basic sub-flowgraphs is said to be 'irreducible'. This produces four axioms (*ibid*, pp335-6):

1. The complexity metric, m, of a sequential flowgraph, F, is a function, g, of the complexity of its subcomponents, $F_1,...,F_n$, given as

$$m(\text{ seq}(F_1, ..., F_n)) = g_n(m(F_2), ..., m(F_n))$$

2. The complexity metric, m, of a non-sequential flowgraph, F, upon the nodes, $x_1,...x_n$, is a function, h_F, of the complexity of all irreducible subcomponents, $F_1,...,F_n$, given as

$$m(F(F_1 \text{ on } x_1, ..., F_n \text{ on } x_n)) = h_F(m(F), m(F_1), ..., m(F_n))$$

3. For a finite set, n, of components in sequence, g_n, the complexity metric, m, will generate a result within a range, given as

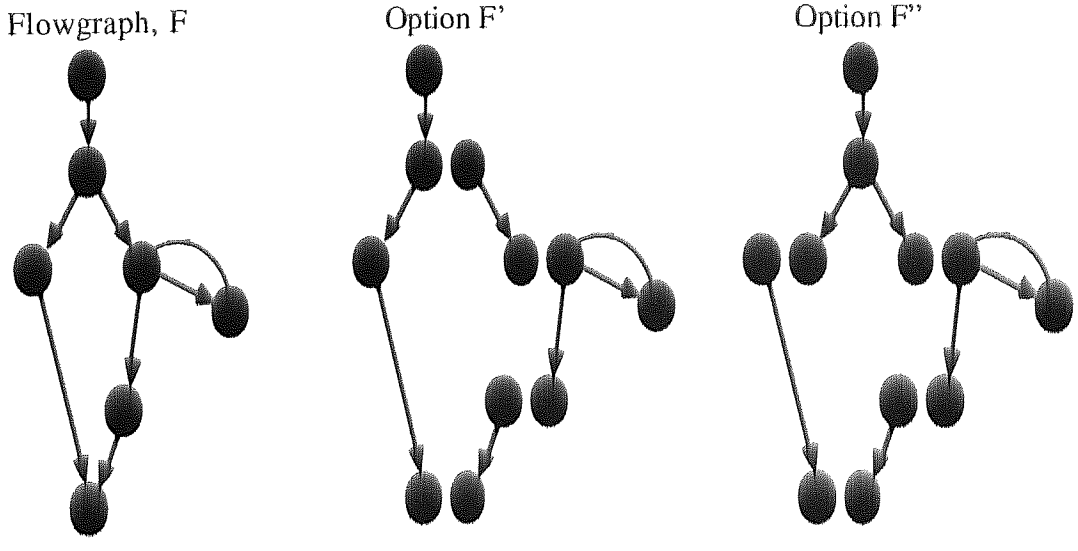
$$g_n^{\text{lower}} \leq g_n \leq g_n^{\text{upper}}$$

4. For each irreducible digraph, F, of nested components, h_F, the complexity metric, m, will generate a result within the range, given as

$$h_F^{\text{lower}} \leq h_F \leq h_F^{\text{upper}}$$

Axiom 1 effectively preserves Prather's first axiom - that a program's basic structure is a sequence of sub-components. The second axiom suggests the complexity of the whole is measured in the same way as an analysis of a finite set of sub-components, but do not follow Prather's suggestion in saying that the complexity of the whole is greater than the sum of the parts. Axioms 3 and 4 impose the limits of the metric which preserves the monotonic relationship between parts of a flowgraph and its whole.

Figure 3.6 : Options in the decomposition of a flowgraph, F



However, while Prather broadly accepts Fenton & Whitty's (1986) treatment of hierarchical metrics (Prather, 1987) and it seems reasonable to begin by accepting axioms 1 and 2, the relationship between a metric and its result (be it g_n or h_F) still does not seem to be fully represented. Notably, given that Fenton & Whitty's goal is to reduce a flowgraph to an unambiguous set of irreducible digraphs it is equally important to define a means of unambiguously defining how members of this set are identified and combined. A number of 'the most commonly occurring' irreducible digraphs are mentioned (*ibid*, p331), but make no mention of how these important 'building blocks of programs' are identified. For instance, another flowgraph, F, given by Fenton & Whitty (1986, p331) could be reduced to more than one combination of digraphs, F' to F^n (see Figure 3.6).

There are the same number of digraphs in each case so it cannot be suggested that the correct reduction has the fewest number of sub-components. Neither could it be argued that the forked digraph at the top of Option F" in Figure 3.6 is invalid, because Fenton & Whitty do not give a means of accepting or dismissing particular flowgraphs. The conclusion must be, therefore, that Fenton & Whitty's axioms are incomplete, and it is hard to see what needs to be added in order for the axiomatic set to be made complete.

A fuller description would seem to be given by Weyuker (1988), who set out to derive a means of comparing competing models of software complexity. If | P | represents the non-negative complexity measure of P (however it is measured), then Weyuker presents nine desirable properties of complexity measures which are based only on syntactic features of a program. They are, that a good complexity metric must:

1. *Discriminate* (assigns different numbers to programs which are not the same).
2. *Not be too insensitive* (does not assign the same number to programs which are different).
3. *Allow for equivalence* (assigns the same number to programs which are the same).
4. *Detect syntactic differences* (since measures of complexity are here based on structure, not function).
5. *Be monotonic* (since subprograms can be no more complex than the whole).
6. *Allow that the concatenation of two subprograms does not necessitate an increase in structure* (for instance, if the second subprogram does not interact with the first).
7. *Be sensitive to component ordering* (since a change in interaction between components should be reflected in the measure).
8. *Be insensitive to syntactic renaming* (since the ordering has not changed, although psychological complexity might be said to have changed).
9. *Allow for synergistic concatenations* (since there may be additional interactions between the components when they are concatenated).

The properties 6-9 are described by Weyuker as defining more subtle differences between the complexity measures. But she finds a number of well-known complexity metrics fail to fulfil the important properties 2 and 5. The metrics analysed were a count of program statements (described as 'probably the oldest and most intuitively obvious notion of complexity'), McCabe's cyclomatic complexity model, Halstead's programming effort model, and a model of data flow (from Oviedo, 1980). Applying her desirable properties of complexity measures to these four complexity metrics, Weyuker finds that none of the four structure metrics satisfy all nine of the desirable properties.

Cherniavsky & Smith (1991) show that even satisfying these properties does not ensure that the metric itself is a useful or meaningful measure. Shepperd (1991) points out that some of Weyuker's properties are, at best, only hypotheses of good structure and so it is no surprise that Weyuker did not find any structure metric which met her criteria. For instance, monotocity indicates measures which are at least higher than the nominal scale, but how do we define concatenation (property 9)? Of syntactically incorrect programs such concatenations are meaningless, while for design components it would be inappropriate.

Although a number of other attempts have been made in this field (e.g., Zuse & Bollmann, 1989; Ejiogu, 1991; Schneiderman, 1992), one is still left with the conclusion that no clear set of axioms have been established which can guarantee the efficacy of a metric. One reason for this failure could be the problem that the field of software metrics has not matured enough for it to be possible to deduce what the essential properties of a good metric are. This is because no metric has proven itself beyond doubt and established the beginning of a workable science of software measurement. In other words, until a metric which is both accurate and widely used has been found, it is not possible to deduce what it is about a (successful) metric that makes it work. In lieu of this framework the only alternative seems to be to take a naïve approach where proof of the usefulness of a metric is taken to be a reliable and high correlation between the actual and estimate values. Of course, high correlation does not prove a meaningful relationship but if no other framework is in place, what other justification can be given?

This thesis, therefore, will not take an axiomatic approach to generating hybrid metrics, but begin where most of the early studies began: observing what types of useful measurements can be taken and then testing whether these measurements are related to the important properties which need to be estimated. What remains uncertain is whether the measurements that can be taken of knowledge based systems are similar or radically different from the type of (DP) metrics described here. This will be the subject of the next chapter.

4. Metrics for Knowledge Based Systems

> *"The debate over the possibility of computer thought will never be won or lost; it will simply cease to be of interest, like the previous debate over man as a clockwork mechanism."* J. David Bolter (1984, p190).

Summary: *This chapter describes the few metrics which have been developed specifically for knowledge based systems (KBSs). They are typified in the same way that metrics for conventional systems are plausible, mainly directed towards measurements of program code, while more useful estimating models suffer from a lack of validation. It will be argued that although there is meant to be a theoretical difference between conventional and knowledge based systems, these KBS metrics show that measurements of KBSs do not differ substantially from measurements of conventional systems. This holds out the possibility that conventional metrics could be extended to KBSs.*

Knowledge based systems (KBSs) are part of the field of Artificial Intelligence (AI) and can be defined as systems which incorporate at least one heuristic to choose between two or more solutions to a problem. A KBS, therefore, makes decisions based on facts and a knowledge of the domain in which it is working. Thus, a robot which arc-welded a point in space regardless of whether the sheets of metal had made it to that point in the production line would not be called a KBS; while a robot which sought out the appropriate edges of the metal and decided if welding is likely to be successful would be called a KBS. The second robot would have to answer questions such as 'Are the sheets of metal in place?', 'Is the arc-welding torch on?', 'Has the metal been welded successfully?', etc. If the answer to any of these questions is "No", then the KBS robot would have to resolve the problem; the non-KBS robot would simply arc-weld free space. The key to the use of a KBS, then, is the presence of data which may be partial, uncertain, or even false. This places KBSs in sharp contrast to conventional DP systems, which typically require a precise definition of the data to be handled by the system.

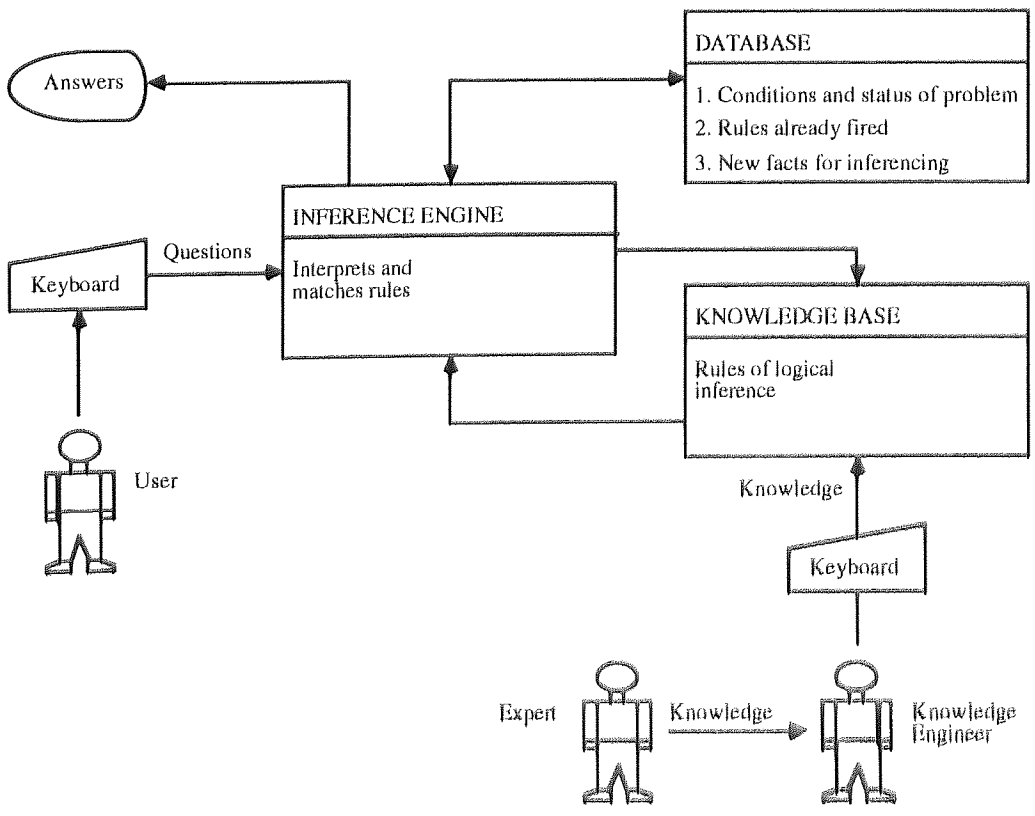
Many knowledge based systems are also called expert systems (ESs). For simplicity, an expert system is defined here as a sub-set of the field of knowledge based systems. The peculiarity of expert systems is that they are designed to take part in a process of

interrogation (of or by humans) whereby information is elicited. Where the human is being interrogated the expert system is often being used to teach the user about the subject domain in which it is expert. The organic chemist DENDRAL (Feigenbaum *et al*, 1971), the medical expert system MYCIN (Shortliffe, 1976) and the Molybdenum geologist PROSPECTOR (Duda *et al*, 1979) are classic examples within AI. The arc-welding robot would only be an expert system if it were possible to ask the robot how it went about welding. Knowledge based systems ordinarily have this interface, and so the terms "knowledge based system" and "expert system" become virtually interchangeable. In order to take a broad view of intelligent systems, the more general term "KBS" will be used throughout this thesis.

The question to be addressed here is whether the development of a KBS can be measured, managed and controlled in much the same way as conventional systems development. If not, then it would be difficult to define any metrics, models or tools which can support the development of hybrid systems since half the system would become opaque to the project manager. The rest of this chapter, therefore, will describe:

- how KBSs are developed;
- current KBS metrics;
- problems with current KBS metrics.

Figure 4.1 : Elements of a knowledge based system (after Anderson, 1989, p95)



4.1 Developing knowledge based systems

Knowledge based systems (KBSs) are typically instantiated as production systems. A production system is defined as a set of IF-THEN rules contained within production memory and a database of assertions (called memory elements) contained within working memory. The "knowledge" within a KBS is represented as production rules in the knowledge base (see Figure 4.1). A production rule is a set of (conjunctive and/or disjunctive) conditions, where the left-hand side of the rule (the IF-part) must be matched with elements in the working memory. The working memory is a database of assertions which effectively details what the KBS "knows" to be true at any one time.

With the input of data the production system checks to see whether the data matches any IF conditions. All productions which can be satisfied form the conflict set and effectively defines the range of deductions that can be made. The interpreter (or inference engine) must then perform 'conflict resolution', whereby one of the rules in the conflict set is chosen for execution (else the system halts). It is at this point that the experts' heuristics make the difference between traversing the entire search space and making the most plausible decision first time around. Once an appropriate rule has been chosen the right-hand side (the THEN-part) is executed. The right-hand side of a production rule contains one or more actions, such as, placing another fact in the working memory. Once executed, the cycle of match-resolve-execute is repeated until no further actions are required.

The heuristics coded within the system are most often derived by interviewing human experts in the relevant field and finding some way of making this knowledge available to the system (see again Figure 4.1). The use of human experts has led to KBSs being seen as systems which perform tasks that would require intelligence if performed by a human. This has been further refined to include only those domains where the body of knowledge is large (Barr & Feigenbaum, 1982), and the area of expertise narrow and specialised (Hayes-Roth *et al*, 1983; Waterman, 1986). It should be noted, however, that there is no reason why a KBS should be said to exhibit only "human" intelligence; the key here is the use of knowledge, not of species.

The greatest problem with developing a KBS is coding the knowledge of the expert, which the experts themselves may find difficult to articulate. Thus, prototyping is characteristically used in the building of such systems. This technique has four principal stages:

1. define the broad requirements of the system;
2. rapidly build a small, rough system addressing some key issue in '1';

3. allow the expert and end-user to evaluate the prototype in '2';
4. carry out any changes and return to step '2' to implement further enhancements.

These prototypes are further seen as progressing through five stages of evolution (e.g., Waterman, 1986, p140):

1. *Demonstration* (a high-profile part of the problem is coded, demonstrating the viability of the KBS implementation).
2. *Research* (the depth of the systems' domain knowledge continues to increase, giving a credible performance, although not fully tested).
3. *Field testing* (the system is tested within the user environment).
4. *Production* (quality, reliability and speed requirements are satisfied).
5. *Commercial* (the system is in regular commercial use).

Unfortunately, in the mid-1980s Waterman noted that only a few expert systems had reached the fourth stage of evolution. Those that had also tended to grow explosively in size and often required re-implementation into a more efficient language to enhance speed. This can be seen in the development of the knowledge base of the VAX expert system R1/XCON, which grew over 500% from 772 rules in 1980 to 4 000 rules in 1986 (McDermott, 1980; van de Brug *et al*, 1986). This raises critical questions about the ability to test or maintain knowledge bases, an area which remains an active area of research (e.g., Hayes-Roth, 1989; Preece, 1990; Soorgard, 1991).

There is now more evidence that KBSs are being actively used in a commercial setting (e.g., D'Agapayeff, 1987; Andrews, 1989). A telephone survey of 199 companies conducted on behalf of the UK government's Department of Trade and Industry, DTI (1992) reported more than 1500 KBSs in existence, with 'more than a suspicion (p12)' that many more existed. It was found that over 55% of these systems addressed the complex tasks of diagnosis, advice and assessment (p14). Even given these more encouraging signs it must still be acknowledged that the development of KBSs remains a risky venture. What makes KBS development particularly risky is the lack of a comprehensive methodology. Without a well-understood method it would be difficult to control the project since it would not be clear where the project should go next. Prototyping itself is an inadequate methodology since, as Edwards (1991) notes:

> "... there is no final "stopping criterion", the process [of prototyping] can never be said to be finished. Some regard this as a virtue in terms of flexibility, others as a disaster in terms of an open-ended commitment to the use of resources (p103)!"

Beyond simple code hacking (which has no manageable project structure), those KBS methods which do exist range from re-interpreting the Waterfall model (Macleish & Vennergrund, 1986; Weitzel & Kerschberg, 1989), through cyclical-prototyping models (Ince & Hekmatpour, 1988; Konig & Behrendt, 1989), to the multi-dimensional knowledge acquisition and implementation phases of KADS (e.g., Hayward *et al*, 1988; Breuker & Wielinga, 1989). Since the late-1980s, however, there has been a growing realisation that AI techniques have a role to play in conventional Software Engineering and KBS development can profit from the interaction (Partridge & Wilks, 1987; Bader, 1988; Partridge, 1990). More importantly, the difference between conventional and KBS as software systems seems to be closing.

For instance, Baumert *et al* (1988) note that since at least 50% of expert system building is still traditional data-gathering and coding then some of the well-understood characteristics of the Waterfall model could be used to define an expert system methodology. Such a methodology, they suggest, would have to satisfy the following 16 requirements (*ibid*, p333):

1. Include a section defining terms;
2. Establish guidelines for evaluating the applicability of an ES solution to the problem;
3. Define the rôle of the project members;
4. Make the expert an active member of the development team;
5. Involve the users;
6. Assign a knowledge engineer whose responsibility it is to deliver a system which is satisfactory to both expert and user;
7. Use prototyping to develop the requirements definition and expert system software in parallel;
8. Allow for frequent prototype demonstrations;
9. Allow for rapid, but controlled, system changes;
10. Agree a date when requirements are finalised (and prototyping comes to an end);
11. Provide for an objective means of reporting project progress;
12. Establish a set of baseline documents (updated with each prototype and formally delivered at the end of the project);
13. Provide guidelines for budgets, schedules and contingencies;
14. Establish a set of standards and procedures which include knowledge acquisition and promotes modularisation and a structured approach;
15. Allow for an active rôle by the group controlling system quality (e.g., Quality Assurance);
16. Establish open lines of communication throughout.

What is noticeable about these points is that although they use key words such as "expert", "knowledge", "prototyping", etc., only three of them are intrinsically related to expert system development (requirements 2, 4, and 6). Requirement 2 (evaluating the applicability of ESs to the problem) is a more specific case of a feasibility study where the "KBS-ness" of the problem must be evaluated. Similarly, requirements 4 and 6 (making the expert part of the development team and using a knowledge engineer to design the system), are a recognition of the extra "expert" dimension of the system. The other requirements, however, can be applied equally well to models of conventional systems development (see again §2.4). This includes the need to structure the use of knowledge acquisition techniques (requirement 14), since this can be equally useful to both conventional and KBS development. Indeed, the use of a first-pass prototype to capture poorly understood requirements was a key strategy advocated by Royce (1970) for conventional systems.

A generalisation of these points within Baumert *et al*'s guidelines would then suggest that: a methodology (for any type of system) must foster confidence and co-operation amongst the group by providing guidelines (for development), standards (for knowledge acquisition and prototyping) and procedures (for reporting and monitoring). It would then only be in the particular techniques used which would make KBS projects differ from conventional projects. This result suggests that although KBSs typically attempt to automate uncertain and complex problems, parallels can be drawn between the development of KBSs and conventional DP systems. Given the similarity in project structure, there seems to be no reason why KBSs cannot be measured in the same way as conventional systems. The problem of knowing what to measure and how meaningful the metric can be remains equally problematic between conventional and KBS projects. The question becomes: what is there to measure?

4.2 A review of KBS metrics

There seems to be little work being done on KBS metrics although it seems unlikely that this lack of research is because KBSs do not suffer from the same problems of estimation as conventional systems. It seems more likely that KBSs have yet to attain a commercial footing where they are seen as the right solution to business problems and therefore demand the same level of management control. Defining exactly what sort of problem is amenable to a KBS solution is itself a non-trivial problem (Basden, 1984; Prerau, 1985). As such, the literature which details measurements of and models for KBSs remains relatively scarce. Given the peculiar KBS characteristics of production rules, IF-THEN matching and prototyping, it is no surprise to find that those KBS metrics which do exist have focused on these properties. Furthermore, following conventional history, KBS metrics began by

looking at code before asking more general questions about their use to manage KBS development. Those studies which do exist are described below.

4.2.1 *Code metrics*

One of the first measurement studies of an AI program was carried out at Carnegie-Mellon University by Gupta & Forgy (1983, 1989). Six large systems in active use or under continuing development were studied (including a version of the R1/XCON expert system). These systems ranged in size from 103 to 1932 production rules. Four of the systems were written in OPS5 and two written in SOAR. Gupta & Forgy's prime objective was to test if the matching of rules using the Rete algorithm would benefit from parallelism, i.e., where a processor is assigned to each node in a Rete-network. The Rete algorithm was developed by Forgy (1982) to address the problem that 90% of run-time is taken up with production rule matching. The Rete-network links the left-hand side of all production rules and stores the result of a match as a state within the network. Data then moves the working memory from one state to another.

The study carried out by Gupta & Forgy made a static analysis of the code by taking measurements of a number of textual properties. Counts were made of such things as:

- condition elements per production;
- actions per production;
- negative condition elements per production;
- attributes per condition element;
- tests per two-input node;
- variables bound and referenced;
- variables bound but not referenced;
- variable occurrences in left-hand side;
- variables per condition element;
- condition element classes;
- action types (e.g., make, remove, modify).

The results found, firstly, that the profiles of the OPS5 programs were more like each other than the SOAR programs, while the two SOAR programs were more like each other and less like OPS5 programs. This accords with Lister's (1982) criticism of Software Science that syntactic counts are likely to differ between languages. Secondly, measurements of the Rete algorithm network found that, on average, 63%

of the activity is due to constant-test nodes (those which test whether production rules with constant values are satisfied). Since these are low-cost actions that make very little impact on the working memory elements, Gupta & Forgy conclude:

> "First, we should not expect smaller production systems (in terms of number of productions) to run faster than larger ones. Second, it appears that allocating one processor to each production is not a good idea. Finally, there is no reason to expect larger production systems will necessarily exhibit more speedup from parallelism (1989, p92)."

In other words, by classifying the components of a program, Gupta & Forgy (1983, 1989) found that the interaction between production rules was too simple to warrant parallelism.

What is missing from this study, however, is an examination of the way in which the systems were developed. Were they all developed by the same person? A team? Did project members change over time? Was any method used to control the development of the system? Answers to these questions are needed before any of Gupta & Forgy's contentions can be properly understood. For instance, the difference in the profiles of OPS5 and SOAR programs could be explained by programming style rather than any intrinsic difference between the languages. Furthermore, given the use of systems under continuing development as well as those in active use, it would be difficult to confirm whether the study compared like with like. A first prototype under development is likely to be quite different in nature and structure to a third or fourth generation prototype in active use. Finally, it is questionable whether the sample is large enough to support any of Gupta & Forgy's conclusions.

What is particularly interesting about Gupta & Forgy's study, however, is the way in which they began their analysis by classifying the elements that go to make up a KBS program: in this case, conditions, actions, attributes, variables and tests. This can be related to Halstead's search for a means of breaking down a program into its constituent parts, although Halstead did not seek to use this information to determine the dynamic behaviour of a system. The Rete-algorithm itself can be seen as one approach to the representation of the internal complexity of a KBS program, since it represents explicitly the relationship between facts and their results. Such a representation would seem to be similar to the data-flow graphs of Henry & Kafura (1981, 1984). Apart from a revised version of the 1983 report appearing as the 1989 article, however, there appears to have been no follow-up to this work.

Later attempts to define the characteristics of a KBS program include Markusz & Kaposi (1985) who sought to deal with the psychological complexity (rather than the computational complexity) of an implemented algorithm. By looking for 'objective and quantifiable indicators of the complexity of the design of programs (*ibid*, p487)', predictions of cost and useful system life were being sought. Markusz & Kaposi studied DEC10 Prolog programs and defined complexity in terms of:

- data relating to its environment;
- number of sub-tasks;
- relationships between sub-tasks;
- data flow through the structure.

Markusz & Kaposi use the term "partition" to describe the relationship between Prolog sub-tasks. To briefly describe the syntax of Prolog programs, production rules are expressed using first-order (IF-THEN) predicate calculus and are called clauses. The IF part is called the clause head, while the THEN part is called the clause body. The head and body are separated using a ":-" operator. Data is usually represented using strings beginning with an upper-case letter, e.g., Input, Output, etc. (The Prolog language is described in detail in §9.1.) The local complexity, λ (denotation changed), between clauses and the global complexity, g, of the program are defined by Markusz & Kaposi (1985) as (p487):

$$\lambda = P_1+P_2+P_3+P_4$$

where,

P_1 = number of new data entities in the clause body
P_2 = number of sub-clauses defining the clause body
P_3 = sum of relations between P_2 and P_1 (add 2 for each recursive call, otherwise P_3=0)
P_4 = number of new data entities in P_2

and (*ibid*, p488):

$$g = \sum_{i=1}^{n} \lambda_i$$

An example of calculating λ for a program is given in Figure 4.2.

Figure 4.2 : Example of a program breakdown
(cf. Markusz & Kaposi, 1985, p488)

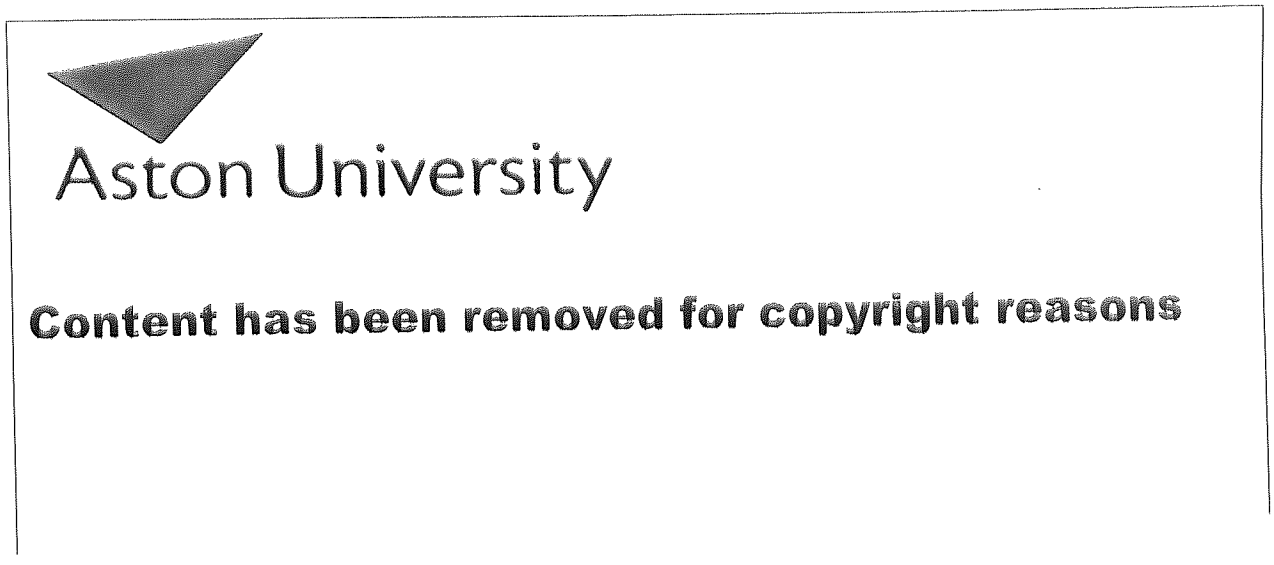
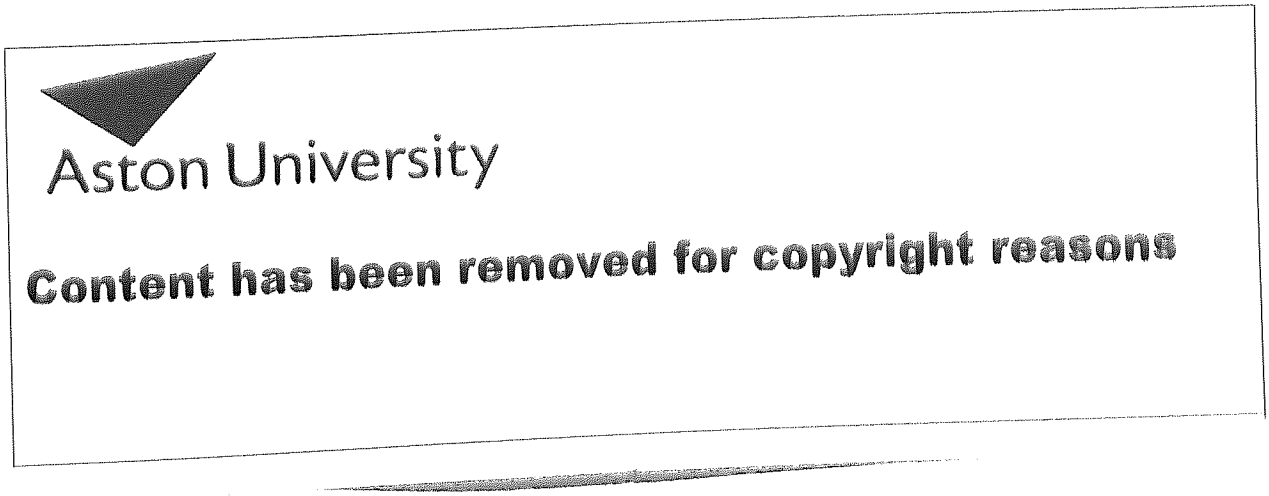


Table 4.1 : Complexity bounds by range
(after Markusz & Kaposi, 1985, p488)



They suggest that λ 'reflects the intuitive feeling of experienced designers about the relative complexity of design tasks (p488)', while *g* is a measure of program size and is hoped to be 'quite different' from λ. By summing - rather than multiplying - local complexity Markusz & Kaposi's definitions can be likened to Chapin's (1979) Q-metric and DeMarco's (1982) "bang" metric, where a count of local data becomes a measure of size.

Markusz & Kaposi focus their attention on λ, and suggest that there is likely to be an upper limit to this measure. They began by suggesting four bounds for λ which nominally follow the 'magic number 7'. They then used this table of bounds (see Table 4.1) to investigate the effect of top-down (TD) and bottom-up (BU) design practices on λ. Three versions of a design methodology were developed (called PRIMLOG, NEW PRIMLOG II and NEW PRIMLOG III, respectively). By studying three (small) CAD programs, Markusz & Kaposi found that:

- top-down design under PRIMLOG produced lower local complexity although if the scores become too low the hierarchy deepens and more logical errors are found;
- mixed TD-BU and only controlling local complexity, average local complexity went up, errors were fewer but harder to find;
- mixed TD-BU and no control, produced a number of much more sophisticated partitions (requiring V to be redefined from V (17-34) to V^{III} (>100));
- most program partitions (from 46%-67%) were rated in the trivial or simple categories, while the program developed using PRIMLOG had no partitions in the highest categories (V and above), the highest proportion of partitions in the lowest categories (S and below);
- using PRIMLOG was quicker (73 man-days) than only controlling local complexity (80 man-days), although no figures are given for the uncontrolled design.

On the basis of these results, Markusz & Kaposi suggested that for experienced programmers an upper limit or bound of λ=7-10 seemed to be appropriate. Although very loose as a bound, the upper bound of this figure is remarkably close to McCabe's (1976) upper limit of v(G)=10, and may suggest that the structure of Prolog programs measured using λ is comparable to conventional programs measured using v(G). Reconstructing programs which exceeded this domain, they suggest, led to a more elegant design. They give the example of a Prolog partition called TASK which had λ=55. When TASK was subdivided into 8 sub-partitions

($\lambda_{average}$=7.6), a partition called VALUE was identified which was subsequently used in a number of other parts of the system. Such a reconstruction is similar to Henry & Kafura's (1981, 1984) study of the UNIX kernel.

The problem with Markusz & Kaposi's formulation, however, is also similar to the criticisms levelled at complexity metrics for conventional metrics: has the measurement between metric and complexity been established? Can the sum of local complexity equal anything other than "total complexity" rather than "system size"? The reconstruction of TASK may suggest it can, but Markusz & Kaposi do not admit how many other partitions with λ>10 could not be usefully reconstructed, or, of equal importance, how many partitions with λ<10 continued to cause the developers problems. Markusz & Kaposi do admit, however, that run-time suffered in the reconstructed partitions but they blamed this on *g* rather than λ. Since *g* is a direct summation of λ it is difficult to see how the blame can be placed on one rather than the other.

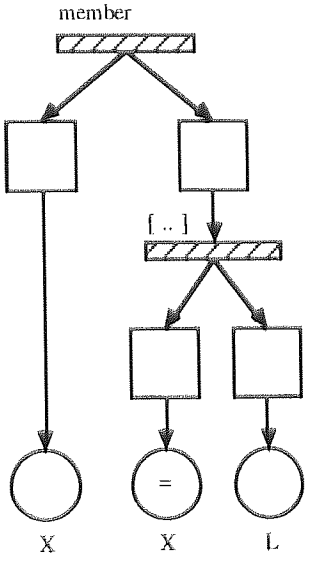
Further attempts to define the structure of PROLOG code were put forward by Myers & Kaposi (1991), who suggested that Prolog is useful not only as a language in its own right but as a general syntax for specifications. This also follows the suggestion of Ince (1984). The measurements they propose are based on the 'text of the code (p414)', or, as Gupta & Forgy (1983, 1989) put it, as measurements of the syntactical (surface) properties of the program. This places Myers & Kaposi well within the standard conventional approach of measuring what is easiest to measure: static properties of the program code. Such an approach is sensible given that until it is known whether any measurement is applicable, it would be difficult to know which set of measurements are "best".

Myers & Kaposi aim to represent a Prolog program in terms of uniquely identifying the location of data, linking the predicates which define the location of data and the nature of the data itself. A "predicate" (or functor) is how Prolog assigns a name to a clause and is represented by Myers & Kaposi as a structure bar at the root of a digraph (e.g., see Figure 4.3). The location of data within the clause is represented by a box while directed arcs model "contains" relationships. Nodes represent data, where an empty node is a variable, a node with a value is a constant and a node with an "=" models two variables with the same name (see Figure 4.3a). Clause bodies are also called data "forests" and are represented with data "growing" from the functor root (see Figure 4.3c).

Figure 4.3 : Representations of a Prolog clause (after Myers & Kaposi, 1991, pp419-420)

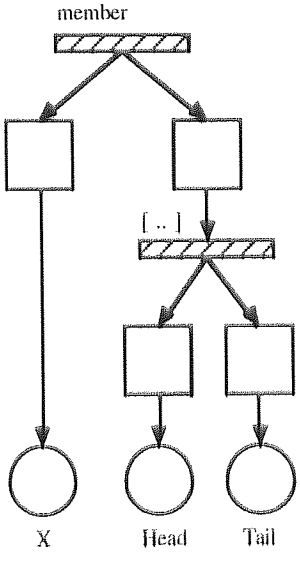
(a) Digraph of a fact

member(X, [X | L]).



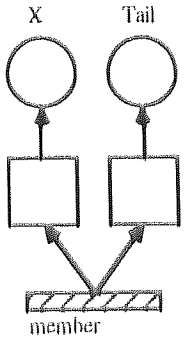
(b) Digraph of a clause head

member(X, [Head | Tail]):-

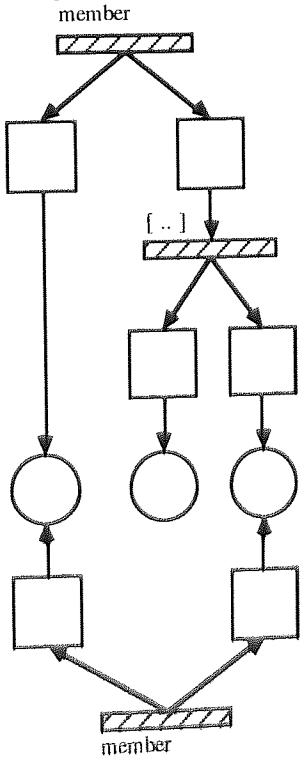


(c) Digraph of a clause body

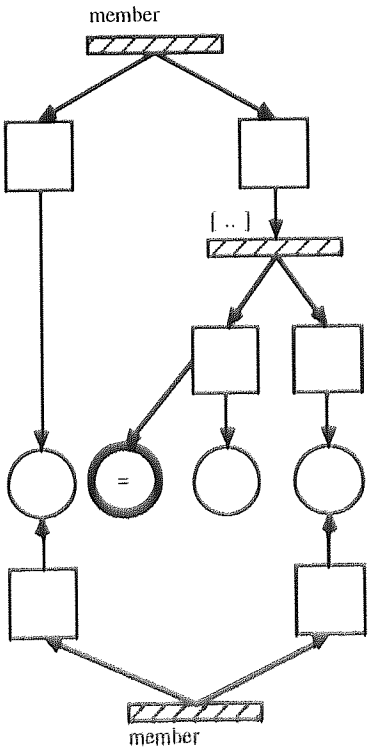
member(X, Tail).



(d) Simplified digraph of (b) and (c) showing recursive data relationships



(e) Procedural digraph of the two **member** clauses in (a) and (d)



NB: The node values are not shown in (d) or (e) in order to make the representation clearer.

Recursion - whereby a procedure calls itself - is represented by including a relation between the original clausal predicate and the lowest-level data structures (shown twice in Figure 4.3d as the fusion of the nodes **X** and **Tail**). This would seem to adequately capture the complexity of data referenced more than once by the same predicate. Using this form of representation, a Prolog procedure is the linking of two or more data relations (see Figure 4.3e), while a Prolog program becomes:

> "... a hierarchy of Prolog procedure models linked through source locations. The links are established by procedure and relation names acting as pointers in the locations, derived from the initial code (p419)."

The value of a model representing the nature of a program is that the model can be used to define the space within which measurements are taken. While Gupta & Forgy (1983, 1989) used the Rete-network to denote elements within a KBS program, Myers & Kaposi (1985) take similar measurements from their model, including (p428):

- number of root nodes, A;
- number of location nodes, L;
- total number of 'content nodes' (structure bars and atomic data types), T;
- total number of leaves, Y;
- number of constants, C;
- number of distinct variables, V;
- total number of variables, X;
- number of variable repetitions, I;
- number of structure nodes (functors), S;
- depth of nesting, D;
- out-degree of the i^{th} structure node (number of arcs from a structure bar), E_i.

These counts allow Myers & Kaposi to define the inter-relationships between the data parameters and their location as a representation of nodes and boxes. Ejiogu (1985, 1991) proposed a similar tree-like model for the entirety of conventional systems development. The software problem Δ is defined as an inverted tree where each node of the tree represents the decomposition of the problem into complexes (units of function/thoughts), $\Delta_1,...\Delta_i$. The notion that Δ can be refined into a tree is defined as Theorem 1 and relies on the use of structured programming. Ejiogu (1985, 1991) then goes on to suggest measuring the height of the tree and its monadicity (number of leaves). Such proposals are clearly similar to those put forward by Myers & Kaposi (1991).

However, while Myers & Kaposi (1991) suggest that their representation allows for measures which resemble conventional structure metrics (*ibid*, p432), their representation misses a crucial point. Principally, if their representation is to truly resemble other conventional structure metrics, why have they gone to the trouble of devising an essentially awkward model of Prolog? Why not simply define counting strategies for measuring the Prolog code directly? For instance, the counts of V and X (above) are clearly similar to Halstead's operand count η_2 and N_2, respectively, although there seems to be no direct parallel for η_1 and N_1. These counts could be taken without converting the program into Myers & Kaposi's schema.

Taking cyclomatic complexity and data-flow fan-in/out as the classic structure metrics, it is difficult to see how either of these metrics could be defined using the measurements Myers & Kaposi propose. For instance, cyclomatic complexity is based on a count of decision points, but there is no representation of decisions in their model. The number of structure nodes, S, would be a poor imitation of decision points since S is more a measure of the number of arguments within a clause (head or body), and the number of times these arguments are partitioned within the clause body. This is similar to the measurements made by Markusz & Kaposi (1985), but not a metric which can be easily likened to cyclomatic complexity. A more sensible measure of decision points would seem to be a count of functors, the label with which Prolog looks to match one production rule (clause head) with its conditions (clause body). Again, these counts can be made directly from the code and need not rely on the representation put forward by Myers & Kaposi.

Furthermore, while the use of the word "out-degree" to define E_i makes it sound as if it could be a measure of data-flow, it is difficult to see whether structure nodes are the objects within the Prolog model upon which to base a notion of flow. Since Prolog is a declarative language (in that it makes no distinction between data and commands) it would be difficult to define any measure of "flow" because a piece of data can thus become ambiguous, while assigning a unique direction would rely on knowing the semantics of the program, i.e., the direction from which an item of data is to be instantiated or passed. The measurements Myers & Kaposi propose would seem to be more a measure of their model than of anything that can be clearly understood as Prolog code.

4.2.2 *Management metrics*

At the same time that measurements of KBS code were being developed, questions about the use of metrics to manage the development of KBSs were being raised. The

earliest example is by Kaisler (1986) who set out to suggest metrics to measure the growth of an expert system. Since expert systems tend to be developed using rapid prototyping, Kaisler suggests that the metric values should be dynamic, not just static, and begin by counting the growth in rules. The cyclical behaviour of a system can be measured with counts of the number of production rules considered, selected and executed per cycle, and the number of cycles required to solve a problem. These basic measurements would then be possible to tackle other issues, including:

- the effect of conventional vs. LISP machines (LISP being another declarative KBS language);
- the rise in domain complexity;
- the trend of the knowledge base to increase;
- the number of hypotheses and consequents of a rule (its complexity);
- the slot access time (time to retrieve data);
- the execution time;
- the application vocabulary (number of distinct predicates).

Kaisler does not provide any empirical evidence of the use of these measurements, however, and admits that 'substantial work' needs to be carried out to evaluate the use of metrics to support the development of KBSs (*ibid*, p119).

The only example of a metrics programme placed within a structured methodology is that put forward by Readdie *et al* (1989) as part of the ESPRIT project 1098 (the first European community-funded KADS project). Their goal was to generate an estimating model to predict resource-planning by focusing on the eight activities the KADS life-cycle model defines for the analysis phase. They are (*ibid*, p27):

1. *Determine scope of project.* Scoping the nature of the project and setting up management procedures.
2. *Analyse present situation.* Understand and document those relevant parts of the organisation which will influence the task and functional requirement of the KBS.
3. *Analyse static knowledge.* Acquisition and structuring the knowledge for the domain ("static") layer of the Conceptual Model.
4. *Analyse objectives and constraints.* Same as activity 2 but for systems issues (real- time, multi-user, etc.).
5. *Analyse expert and user tasks.* Eliciting the remainder of the Conceptual Model knowledge. Dependent on amount of interviewing (activity 2) and analysis of elicited knowledge (activity 7).

6. *Determine functional requirements.* Similar to activity 2 and 4 but focusing on system and project rather than client factors. Synthesis of data towards a functional specification.
7. *Construct conceptual model.* A model of the generic tasks which define the kind of expertise being implemented (e.g., diagnosis, planning, etc.).
8. *Estimate feasibility.* Estimating the likelihood that the project will be successfully completed.

Readdie *et al* then suggest that estimating the cost of each of these eight tasks is based on an understanding of what is produced, *FP*, and how the activity, *FA*, achieves this. On this basis, a nominal cost model is given as (*ibid*, p29):

$$C(A,P) = FA(A) * FP(P)$$

But how can the *FA* and *FP* factors be evaluated? Readdie *et al* attempted to isolate *FP* by conducting a questionnaire study of ESPRIT Project 1098 partners. Six "experimental" uses of KADS were studied which ranged in duration from 15 to 240 man-weeks. However, they found (*ibid*, p34) the results were too widely divergent and no patterns emerged. This reduced the original model to a naïve representation of cost, C_T, given as (*ibid*, p35):

$$C_T = \sum_{i=1}^{n} C(A_i)$$

or, the total cost is the sum of the costs of all the activities. This ignores overlaps and interplay between activities, an effect which Brooks (1975) stressed was a significant reason why conventional project managers grossly under-estimated the cost of a project. The advantage of Readdie *et al*'s naïve model, however, is that they can now forward ideas of how to measure the eight analysis activities in KADS (*ibid*, pp36-42):

1. *Determine scope of project.* Evaluate the influence of:
   - potential disagreement between users;
   - proportion of people involved in scoping and beyond;
   - method of agreement (committee or single manager).
2. *Analyse present situation.* Evaluate the influence of:
   - number of experts to be interviewed;
   - client "push" or project "pull" on these experts;
   - amount, quality and availability of documentation.

3. *Analyse static knowledge.* Evaluate the influence of:
   - number of concepts;
   - accessibility of the domain knowledge to layman;
   - amount of data;
   - number of - and agreement between - experts;
   - formal (e.g., engineering) or implicit (e.g., sales and marketing) knowledge;
   - availability of documentation tools to aid analysis.
4. *Analyse objectives and constraints.* Evaluate the influence of:
   - size and complexity of system requirements.
5. *Analyse expert and user tasks.* Evaluate the influence of:
   - lucidity and motivation of experts;
   - documentation reading time;
   - nature of documentation (reference manuals useful for detail but not for understanding; reverse for user guides);
   - skill of the interviewer(s);
   - familiarity of the interviewer with the domain;
   - accessibility (location and demand on time) of the experts.
6. *Determine functional requirements.* Evaluate the influence of:
   - size and complexity of functional specifications.
7. *Construct conceptual model.* Evaluate the influence of:
   - nature and complexity of the problem-solving task;
   - analysts' understanding of the domain.
8. *Estimate feasibility.* Evaluate the influence of:
   - effort required to design and implement the expertise.

Readdie *et al* suggest that project managers should use their experience to evaluate and adjust these factors and on this basis a notion of cost can be derived. It is notable that most of these factors are similar, if not identical, to issues and problems for conventional software development. Of course, at the heart of any KBS development is the time to elicit knowledge from the expert (activity 5). Readdie *et al* propose a model for this task where the interview time, t, to acquire knowledge from a number of experts, N, is given as (*ibid*, pp39-40):

$$t = t_0 + \delta_1 . N$$

where,

t_0 α size and nature of the total expertise to be elicited
$\delta_1.N$ = initial overhead getting to know the experts
t_0 >> δ_1

Where the experts agree and all the interview time is spent acquiring different parts of the expertise from each expert, the model incorporates the notion of an overhead in talking to more than one expert. In the extreme case where the experts disagree, each expert must be interviewed fully such that:

$$t = t_0.N + (\delta_1 + \delta_2 + \delta_3).N$$

Readdie *et al* note that the KADS representation scheme needs to be formalised further before the validity and accuracy of these metrics can be properly assessed. Furthermore, while the number of KBS development projects available for study remains small they also find it difficult to define any sensible product (i.e., quality) metrics. The method by which they would deduce such quality metrics is based on a modified version of the scientific paradigm and has five steps (*ibid*, pp51-2):

1. Specify the most dominant primitive factors.
2. Assume a simplified functional dependence of the metrics, e.g.,
   - linear, quadratic, etc.;
   - multiplicative dependence, etc.
3. Test the postulation experimentally.
4. Give convincing reasons for the results.
5. Refine the preliminary result by
   - including some further primitive features;
   - assume functional dependence, etc.

What is interesting about this method is that it usefully summarises the way to systematically derive any metric, KBS or otherwise. Unfortunately, the follow-up project (KADS-II, ESPRIT Project 5248) does not seem to have utilised this method.

Devising a methodology or framework by which to derive metrics for KBS development is characteristic of more recent studies in this field. For instance, Behrendt *et al* (1991) used the GQM-paradigm to derive a model which takes account of management and user perspectives. The GQM paradigm was developed by Basili & Rombach (1988) as a means of focusing the development of (originally, conventional) metrics in achieving management *G*oals by setting out a number of *Q*uestions which need to be answered before developing *M*etrics which will answer

these questions, and hence, achieve the required goals. For Behrendt *et al*, their goal was to provide metrics for up-front cost estimation and project planning, progress monitoring and KBS quality assurance. They begin by criticising 'hindsight' metrics - such as structural complexity as an indicator of maintenance requirements - on the basis that they:

- are isolationist, by not taking into account the context in which the maintenance is being carried out (available tools and skills);
- come too late, arguing that 'measuring maintainability when the code is written is too late';
- their application still fails to give project managers what they need to know, since maintenance requirements need to be thought of when the design is produced.

Furthermore, like the KADS study, Behrendt *et al* find it doubtful that 'models based on conventional software can be made applicable for AI software (*ibid*, p1059).' Instead they propose a KBS model which seeks to quantify the following six key characteristics (*ibid*, pp1061-2):

- the domain (expert tasks, level of expertise required, etc.);
- the type of KBS software (design type, number of rules, etc.);
- the client's objectives (business related decision-drivers, etc.);
- the type of KBS project (effort/risk analysis, etc.);
- project management methods (plans, schedules, use of methods, etc.);
- KBS quality issues (usability, reusability, efficiency and performance, correctness/completeness/accuracy, reliability/security, maintainability).

The model does not produce metrics automatically but is used as a tool which leads the project manager to consider which issues are important and need to be controlled. The applicability of the model was investigated by gathering data from 20 case studies which raised a number of common problems, including: the lack of a firm basis upon which to make size estimates (where first prototypes are seen to be linearly related to subsequent prototypes); the issue of re-usability; maintaining/updating the knowledge base; validating and verifying the system as a whole. There is no mention of their model being used to define or apply any metrics to solve these problems.

What is notable about these points is that after eliminating the word "KBS", Behrendt *et al*'s model is equally applicable to conventional systems development - in

the same way that Baumert *et al*'s (1988) description of a KBS methodology could be generalised. The issues raised by the case studies are certainly germane to all types of software development. A more acute problem that Behrendt *et al* found for KBS metrics is that only four of the 20 case studies used a methodology. Without such a structure from requirements document to final system it would be difficult to trace the relationship between early properties and final results. In effect, if the use of a method is more common for conventional projects, Behrendt *et al*'s model is perhaps more applicable to conventional rather than KBS projects.

4.3 Problems with KBS metrics

Since Project IED4/1/1426 is attempting to support the development of hybrid systems there is a need to define metrics for a system which contains both conventional and knowledge based components. However, the discussion of metrics in this chapter suggests that if it is difficult to define metrics for conventional software, knowledge based systems are equally problematic. Although code-based metrics would seem to be a simple (and logical) start, the development languages for KBSs are quite different. For instance, conventional systems tend to use procedural languages such as COBOL, FORTRAN and PASCAL (with explicit control transfers), while KBSs tend to use declarative languages such as PROLOG and LISP (where control transfers are more implicit). Taking the two key properties of cost and quality, the following argument might be used to explain why metrics cannot be extended to knowledge based systems (KBSs):

1. Cost is usually defined as a function of the size of the system (often, in terms of thousands of LOC). An idea of system size is only possible when the estimator has a detailed design to work from. Thus, it might be suggested that:

   Ability to estimate cost α Design specification detail

   But it is a characteristic of KBSs that there is no clear understanding at the outset of how the system is going to work because it is unclear what knowledge the "expert" is using or how best to represent it. Therefore, the first point at which there will be sufficient detail to form an idea of cost is when the system has been developed. This, of course, makes early estimations of cost redundant.

2. Quality is often defined in terms of a list of properties, such as portability, ease-of-use, etc. A lack of defects (or bugs) is a strong member of this list. Defining a defect in a KBS is problematic, however, since while KBSs characteristically deal with uncertain knowledge in complex domains there can be doubts over whether the

> KBS has or has not developed a bug. In other words, in the same way that physicians can disagree over a diagnosis, when objecting to the output of (say) a medical KBS it could be suggested that the KBS has simply entered the realm of medical debate. Although operational bugs such as system crashes are clearly easier to spot, when dealing with knowledge the term "defect" is not so easily applied, and so, a measurement of the quality of a KBS remains subjective and difficult, if not impossible.

While accepting the premise, this thesis rejects both conclusions. Firstly, although it is important to gain as much information as possible about the system to be developed, there seems no reason why the strategies employed in sizing conventional systems cannot be extended to KBSs. Whether lines of code, function points or some other method is the best approach is a matter of research and debate, as it is with conventional systems. Secondly, if the number of "bugs" is accepted as a surrogate measure for quality, then the existence of bugs in a KBS will be as feasible (and problematic) a measure as that employed for conventional systems. The bottom line, therefore, is that if KBSs are taken to be software systems - and as such, coded systems - with similar documentation and program artefacts as conventional systems, then there seems to be no reason why KBSs cannot be measured using a similar set of metrics.

This conclusion is supported by Shepperd & Ince (1991) who outline a number of KBS metrics loosely based on a re-interpretation of lines of code (LOC) to a KBS, such as: the degree of nesting within production rules as a measure of the readability or understandability of the rules; or, the rate of change of rules as a measure of the growth in the knowledge base. They also suggest (*ibid*, p9) that the software cost estimating model COCOMO (Boehm, 1981) could be modified for KBSs using, say, the number of predicate rules as input rather than LOC. The cost drivers within COCOMO are seen as being equally applicable to KBS as to more conventional systems. All the suggestions, it is admitted, are speculative.

If it is granted that KBSs have essentially the same artefacts as conventional systems then it would seem equally sensible to attempt to re-interpret the classic conventional metrics for KBSs. Can the metrics devised by Halstead (1972, 1977), McCabe (1976) and Henry & Kafura (1981, 1984) be applied to a KBS using a language such as, say, Prolog? Before these questions can be answered, however, a more fundamental problem exists, namely: what use would a project manager make of the results returned by a metric if - as Chapters 3 and 4 have suggested - their theoretical basis is almost always dubious?

The suggestion here is that project managers do not want raw numbers; rather, they want tools which will give them early answers to the crucial question of cost and quality.

Measuring the structure of a system's design or its code to evaluate quality is a clear and common use of metrics and is one of the questions which this thesis will seek to answer (see Chapters 8 and 9). The issue of cost has been tackled for conventional systems by using the data collected by metrics to develop software cost estimating tools. (The COCOMO model has already been mentioned.) If the metric→model→tool approach assumed in Chapter 1 is to remain intact, therefore, it is important to understand what sort of tool a project manager needs to support the estimation of cost. The question now becomes: what is a software cost estimating tool and how is it developed? This will be the subject of the next chapter.

5. Developing an Estimating Tool

"Annual income twenty pounds, annual expenditure nineteen pounds nineteen shillings and six, result happiness. Annual income twenty pounds, annual expenditure twenty pounds nought and six, result misery." Charles Dickens (Mr Micawber in 'David Copperfield').

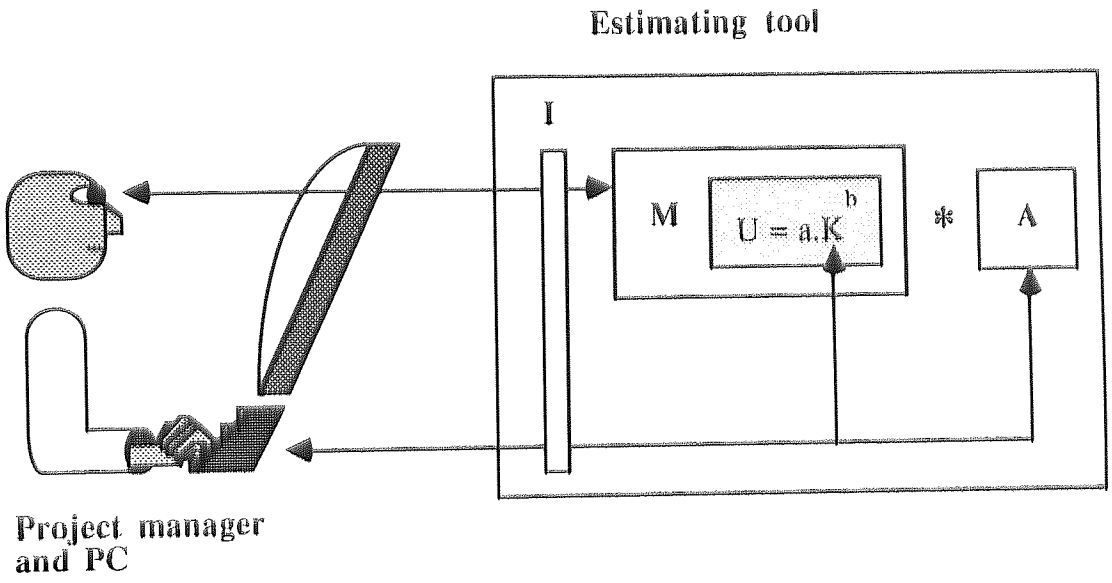
Summary: *This chapter describes the process by which raw (project) data can be converted into a software cost estimating (SCE) tool that estimates properties such as cost, effort (or manpower) and project duration. Such tools are developed by collecting historical data on the resources expended by a completed project and developing a regression model across the dataset. It will be argued that although many of the well-known SCE models have been developed into commercial tools, there is evidence to suggest that the models use poor statistical experimentation, they can be overly complex, and are generally inaccurate. This would seem to cast doubt on the suggestion that SCE tools can be useful to project managers.*

The metrics discussed in the last two chapters have focused on measurements of the size and structure of software because a system must always be developed within available resources and quality constraints. This was part of the rôle of a project manager given in Chapter 2. To continue the metrics→models→tools approach, the focus must now turn to whether such data can be sensibly used to define software cost estimating (SCE) models and tools. The suggestion to be put forward here, however, is that a project manager is only interested in the reverse of this technical process where the tool facilitates interrogation of a SCE model that provides output which is both meaningful and accurate. The tool→model→data process may be called the management process.

The difference between the technical and management process can be likened to the difference in perspective between a mechanic and driver over the use of a car: the mechanic is interested in the car's construction and how the parts interact; the driver must know something about how the car works but only in order to manipulate the vehicle and get from 'A' to 'B'. In software development terms, the 'A' is a project with uncertain properties while the 'B' is a set of accurate values for, say, the size, effort, cost and duration of the project. In other words, the argument here is that a project manager need not know how the

SCE tool is constructed and is more interested in the output, in the same way that a driver of a car need have no knowledge of how the engine works but is keenly interested in its ability to function correctly. This is the minimum understanding a project manager must have. Whether this is sufficient or a source of concern will be addressed later in this thesis.

Figure 5.1 : Anatomy of an estimating tool



More importantly, the SCE tool is the means by which a project manager can apply a consistent and repeatable method of getting from 'A' to 'B' (i.e., generating an estimate). As DeMarco (1982) points out, an estimate is not a guess. The former is the product of a repeatable process or method, while the later is the product of an ad hoc process which would be difficult to repeat or justify. For instance, while a project manager might be expected to estimate the cost of a project, (s)he could only hazard a guess as to the effect of any poorly defined requirements. An SCE tool is defined here, then, as the means of arriving at a repeatable estimating result. Such a tool contains three essential components (see Figure 5.1):

1. A (mathematical) model, M, which relates some knowable system property, K, to some other more useful unknown property, U, such as project cost and duration.
2. A series of adjustment factors, A, which configure the generic model, M, to a particular project.
3. An interface, I, whereby the user can input the project data and observe the effect of K and A on U.

Typically, a SCE tool will incorporate two related models. The first is a model of development effort based on a nominal value for system size, while the second is a model of project duration based on a nominal value of development effort. Thus:

Effort	=	$a.(Size)^b$ * Adjustment_factors	(in person-months)
Cost	=	Effort * £person-month^{-1}	(in £s)
Duration	=	$c.(Effort)^d$	(in calendar-months)

The cost of a project is a function of (person-months) effort, where, if the £ overhead that can be attributed to one person working for one month can be calculated, total cost is this figure multiplied by the total effort. In practice, this figure may be difficult to calculate given the different salaries of personnel working at different grades, and whether or not to include general costs such as maintenance of the building, the cost of heating, lighting, etc. Differences in these definitions - like those for LOC - would have a significant impact on the actual £ estimates given by two or more SCE models.

Duration (or schedule) is simply the actual "elapsed time" length of the project often given in calendar months. Finally, dividing effort by duration gives an indication of the number of people (or staffing level) required by the project. Thus, if two people work for one month at £5 000 per month, the effort would be two person-months, the cost would be £10 000 and the duration would be one month. The trick, of course, is to know this before the project is complete. For the purposes of this thesis, the terms "duration" and "schedule" will be regarded as interchangeable.

Adjustment factors reflect the variation of technical, performance and management constraints between projects. These factors can range from considerations of the memory requirements and experience of the development personnel to the number of miles travelled by the project members. These factors are often rated on a Likert scale from 1=very low (not important) to 5=extra high (very important) and a formula used to calculate a weighting applied to the generic model. Multiplying the (nominal) estimate produced by the model, $a.K^b$, by the weighted adjustment score, A, gives the final adjusted estimate, $U=a.K^{b*}A$. In this way, internal differences between projects can be modelled and the sensitivity of the estimating model across environments increased.

The need for an interface is based on the premise that if a tool is difficult to use it will not be used. A tool can be defined simply as an implement that reduces the effort required to perform a task and increases the efficiency, productivity or quality of the same task. The difference between an estimating tool and its underlying model, therefore, is that a model is meant to represent the properties of the software development event, while its use as a tool

presumes data have been collected which validate the model and make it operational. In other words, $U=a.K^b$ is a model while (say) a spreadsheet programmed with $Effort=3*(Size)^{1.12}$ is a tool.

The use of an interface does not rule out paper-based tools, but it seems reasonable to suggest that any tool which attempts to reduce the effort and increase the efficiency of producing estimates will naturally migrate towards an electronic platform. More sophisticated tools would include extensive help and data export facilities which a company might decide was the difference between competing tools (e.g., Fisher & Gorman, 1990). For the purposes of this chapter, no distinction will be made between estimates of global properties of a project (such as total effort, manpower, etc.) and estimates of local properties of individual tasks or products. The method of arriving at global or local estimates should be equally valid. The problem is how to construct the engine, M, that drives the SCE tool and provides estimates that are both useful and accurate. To resolve this problem, the rest of this chapter will set out to describe:

- techniques used to develop SCE models;
- SCE models developed for conventional systems development;
- problems in the use of SCE tools.

5.1 Techniques for developing an estimating model

Building a model using data gathered from measuring software development is a straightforward - although controversial - task. With measurements across a number of projects a variety of statistical techniques can be used to define the properties of a model. The most popular techniques employed are correlation and regression. A number of techniques have also been suggested for evaluating the performance of SCE models. These techniques are described below.

5.1.1 *Correlation*

Correlation quantifies the hypothesis that a change in one variable (K, in Figure 5.1), is accompanied by a change in another variable (U, in Figure 5.1). Applying a technique such as Spearman's Ranked Correlation Coefficient, r, returns a value between +1.00 (perfect proportional relationship), through 0.00 (no correlation) to -1.00 (perfect inverse relationship). The value of r at which a correlation is said to be "good enough" is a matter of choice. The co-efficient of determination, r^2, is the percentage of variance in one measure accounted for by the other. For instance, if the correlation between K and U was r=0.90, then the co-efficient of determination,

r^2=0.81. This suggests that 81% of the variance in U is accounted for by K. Conversely, 19% of the variance remains unaccounted for. Again, the point at which r^2 is high enough for K to be a useful predictor of U is a matter of choice.

A high correlation is used to verify the assertion that there is some stable relationship between the variables K and U so that the behaviour of U can be predicted if K is known. For instance, if the correlation between the number of control branches and, say, the number of bugs is high, then taking the same count of an untested program when the number of bugs is unknown will indicate how many should be found if the testing procedure is successful. A model, therefore, must also indicate which known properties should be measured in order to indicate the typical size of currently unknown properties.

It can be a matter of argument, however, whether the relationship shown is a meaningful or useful one to model. The model just described would depend on all programs being error-prone at the same rate, although this would not seem to be unreasonable, all other things being equal. What is more difficult to prove is that the number of control branches is directly related to the number of errors and not just coincidental. For instance, no-one would suggest a count of storks' nests is related to the UK (human) birth-rate, even if there happens to be a high correlation. If control branches are directly related to the number of errors, then understanding the nature of control branches should lead to some insight about the way errors are introduced into a program.

Such models also presume that the relationships being represented are meaningful, but Courtney & Gustafson (1993) point out that if a large number of variables are taken from a small number of observations, correlations of r>0.70 are easily achieved. This can be so even when the variables are functions based on random numbers. They call this many factors, few cases approach "shotgun correlation", saying:

> "In the shotgun approach to software measures, a hypothesis is not stated. The researchers are experimenting with many different aspects of software projects, and although preliminary ideas exist about what may be found, they are not stated in a manner that matches the statistical tests conducted...The reason for labelling this method as the 'shotgun' approach is that the researchers are loading numerous variables and taking a shot to see if they have hit anything (p5)."

Correlation, therefore, does not prove the meaningfulness of the relationship and is simply the technique by which some insight is given into the factors that have a relationship with the properties under study.

5.1.2 *Regression*

Regression is a technique which minimises the "least squares" distance between a series of (data) points and a linear or non-linear line of best fit. The parameters of the regression line define the magnitude of the change in U which accompanies a change in K and are reflected in the estimating model (see again Figure 5.1) by the parameters *a* and *b*.

If the relationship between K and U were presumed at the outset to be linear then b=1.00 and the model would take the form U=a.K+c. This is called linear regression. If the relationship is presumed to be non-linear then non-linear regression produces a model that would take the form $U=a.K^b+c$. The regression analysis calculates the values of *a*, *b* and *c* by producing a regression line, the equation of which describes the relationship between U and K. The model can be forced to go through the origin, where c=0, to give a model where U=0 when K=0. The differences between the data points and the regression line are plotted as residuals, and are used to judge the aptness (or goodness) of the regression model.

On many occasions, the co-efficient of determination between K and U may be deemed (arbitrarily) too low to generate a useful (i.e., accurate) model. In other words, there is too much variance unaccounted for. In this case K may become $K_1+K_2+..K_n$, where K_i are a number of factors which in combination have been found to have a relationship to U. In this case, the model of U must be found by multilinear regression which generates a model of the form $U=a.K_1+b.K_2+..z.K_n+c$. The same issues of *c* preferably being zero and studying the aptness of the regression model exist for multilinear models. Although a number of other statistical techniques are useful (Hamer, 1991), correlation and regression are effectively the verification and validation techniques of building estimating models.

5.1.3 *Factor analysis*

Factor analysis is a technique which can be used to reduce the number of factors in a regression model by grouping together those which are highly correlated to each other. This is one way of deducing that different models are measuring the same "aspect" of software development (Coupal & Robillard, 1990). For instance, lines of

code and the number of input fields might be related to each other as a measure of system size. The model of U might now contain six variables (K_1 to K_6), but after factor analysis it may have produced three groupings:

$$\{ K_1, K_4 \}, \{ K_2, K_6 \}, \{ K_3, K_5 \}$$

Each of the factors within {..} are related to each other and suggests the variance in U accounted for by K_1 is virtually the same as that accounted for by K_4. Thus, factor analysis can simplify the model by (say) choosing only one of the factors from each grouping. The model might now become $U=a.K_4+b.K_2+c.K_5+d$. In this way, fewer properties need to be measured to make the model work. What is still missing, however, is a means of evaluating the accuracy of the model. The closeness between the estimated and actual values of U requires some other form of measurement.

5.1.4 *Performance evaluation*

Boehm (1981) suggested that ten criteria should be used to evaluate the performance and usefulness of a model. They are (p476):

1. *Definition* (clearly defining what is being estimated and what is not);
2. *Fidelity* (level of accuracy given);
3. *Objectivity* (level of influence of subjective factors);
4. *Constructiveness* (clarity of the relationship between the estimate and the nature of the software job being done);
5. *Detail* (estimates given of sub-systems and units with phase and activity breakdowns);
6. *Stability* (relative influence on output of small changes to input);
7. *Scope* (contains data on the class of system being estimated);
8. *Ease of use* (clear meaning of the inputs and options);
9. *Prospectiveness* (not using information which is only available late in the project);
10. *Parsimony* (power without over-use of redundant factors, i.e., Ockham's Razor).

The problem with these criteria is that some of them are ideal rather than useful while it could also be argued that some are even harmful. For instance, given the nature of software development it may in fact be advantageous to keep at least one subjective factor ('3'), while insisting on a great level of detail may mislead the user into thinking the tool is more accurate than it really is ('5'). What is clearly important

is the accuracy of the model itself and a project manager may be prepared to go to great lengths to derive the estimate - as long as it was guaranteed to be accurate.

A definition of accuracy was given by Conte *et al* (1986) who suggested that accuracy was a function of both the general predictive ability of the model, Pred, and the size of error attributable to each estimate, MMRE. Accuracy is defined as (p176):

1. The estimate should be within 25% of the actual 75% of the time, where,

$$Pred(0.25) \geq 0.75$$

2. The mean magnitude of the relative error should be less than 0.25, MMRE≤0.25, where,

$$\overline{MRE} = MMRE = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(\frac{|Actual_i - Estimate_i|}{Actual_i} \right)$$

and,

$Actual_i$ = actual value for the i^{th} data point

$Estimate_i$ = estimated value for the i^{th} data point

This definition has been taken on board and widely used in SCE model studies although the value of ±25% is clearly an arbitrary threshold. For instance, without formally defining Pred as a relationship, Boehm (1981) took it that a SCE model could be deemed to be accurate if it produced estimates within 20% of the actual 70% of the time (*ibid*, p32).

There is even some dissent over the definition of relative error (e.g., Miyazaki *et al*, 1991) and MMRE used to rate the performance of a SCE tool. For instance, Fisher & Gorman (1990) set about choosing an estimating tool for use by software development teams within the UK's Inland Revenue. They focused on four commercially-available SCE tools, namely: Softcost-R, SLIM, Before You Leap Mk.II and Estimacs. The accuracy of these tools was studied using data from 16 completed projects. The relative error between estimate and actual was taken to be a percentage of the estimate rather than - as Conte *et al* 1986 suggested - a percentage of the actual. Fisher & Gorman's argument here is that a project manager is more concerned with being within range of the initial budgeted estimate rather than the statistically correct definition of MMRE (p25). This gives:

$$\text{MMRE} = \frac{|\text{Estimate} - \text{Actual}|}{\text{Estimate}} \quad \textit{not} \quad \text{MMRE} = \frac{|\text{Actual} - \text{Estimate}|}{\text{Actual}}$$

Using the data for all 16 projects supplied by Fisher & Gorman (1990, p86), and using their definition for MMRE and Conte *et al*'s definition for Pred(0.25), Table 5.1 shows the results of the four tools on the three major factors studied (effort, schedule, and staffing).

Table 5.1 : MMRE and Pred(0.25) results for four SCE tools (after Fisher & Gorman, 1990)

Model	Effort		Schedule		Staffing	
	MMRE	Pred(0.25)	MMRE	Pred(0.25)	MMRE	Pred(0.25)
SLIM	1.12	0.44	0.12	0.88	1.18	0.33
Softcost-R	0.37	0.44	0.24	0.63	0.64	0.40
Estimacs	0.44	0.38	0.43	0.25	0.70	0.27
BYL Mk.II	0.21	0.69	0.68	0.50	0.27	0.67

Fisher & Gorman's choice after the analysis was that BYL Mk.II performed best overall and, as can be seen, it does have better results on most of the ratings. However, SLIM can be seen to be considerably better for estimates of schedule. An alternative means of evaluating the best choice of SCE tool might be to score the accuracy of individual tools. One might like to score the performance of the tools in the following way:

- award 10 marks if MMRE<0.25
- award 5 marks if MMRE<0.40
- award 10 marks if Pred(0.25)>0.75
- award 5 marks if Pred(0.25)>0.60

The award of 10 marks represents a "pass" according to Conte *et al*'s MMRE and Pred measure. The award of 5 marks represents a "near miss" given the problems Fisher & Gorman experienced in calibrating the tools. One would expect, in other words, that a tool which came close to passing would improve over time and eventually exceed the performance criteria. The use of MMRE≤0.40 and Pred(0.25)≥0.60 is, like Conte *et al*'s more demanding threshold, entirely arbitrary. Given that there are three factors being assessed, two performance measures for each

Table 5.2 : MMRE and Pred(0.25) scores for four SCE tools

Model	Effort		Schedule		Staffing	
	MMRE	Pred(0.25)	MMRE	Pred(0.25)	MMRE	Pred(0.25)
SLIM	0	0	10	10	0	0
Softcost-R	5	0	10	5	0	0
Estimacs	0	0	0	0	0	0
BYL Mk.II	10	5	0	0	5	5

SLIM = 20/60 = 33%
Softcost-R = 20/60 = 33%
Estimacs = 0
BYL Mk.II = 25/60 = 42%
BYL Mk.II + SLIM = 45/60 = 75%

means a maximum score of 60 can be awarded. The result of this scoring system is given in Table 5.2.

This shows that BYL Mk.II is indeed the relatively best tool: but its performance can hardly be said to be outstanding with 25 marks out of 60 (i.e., 42%). What is more interesting to note is that if SLIM is used to estimate schedule and BYL Mk.II is used to estimate effort and staffing, then the conjunction of the two tools achieves a more impressive score of 45 marks out of 60, or 75%. One could conclude, therefore, that tools used in conjunction can outperform any singular tool. This point confirms the recommendations made by Boehm (1981) and reiterated more recently by Heemstra (1992). Given the common use of Conte *et al*'s (1986) definition and without any additional information a model will be deemed here to be accurate if:

$$\text{Pred}(0.25) \geq 0.75 \quad \textit{and} \quad \text{MMRE} \leq 0.25$$

However, while this section might suggest that as long as care is taken over how the regression analysis is carried out, well-formulated models can be designed for any software development environment, a number of questions still remain. What models have been developed for conventional systems development? Have they followed a reasonable development method? Are they accurate?

5.2 Estimating models

The earliest SCE models were developed in the mid-1960s and presumed a linear relationship between U and K, giving models such as U=a.K+b. The form of these models

came under criticism in the 1970s led, in particular, by Brooks' (1975) thesis that integration between activities generates an overhead that increases the cost of development non-linearly. This produced models of the form $U=a.K^b$. By the early-1990s, however, doubts began to be expressed as to whether the effort models should be non-linear models and whether the multiplicity of models were really significantly different from one another. This may suggest that many of the early studies in SCE models were correct in their original approach to building SCE models. To explain these points, some of the most well-known models are described below.

5.2.1 *Farr & Zagorski (1965)*

Tracing the beginning of research into quantitative models back to an initiative begun by the Systems Development Corporation (SDC) in 1962, Farr & Zagorski (1965) published one of the earliest effort models as part of research carried out at the US Air Force Electronic Systems Division (ESD). Their aim was to provide a quantitative model that could be used by both managers and buyers and solved the cost estimation problem. Farr & Zagorski (1965) saw four principal reasons for this continuing problem (p168):

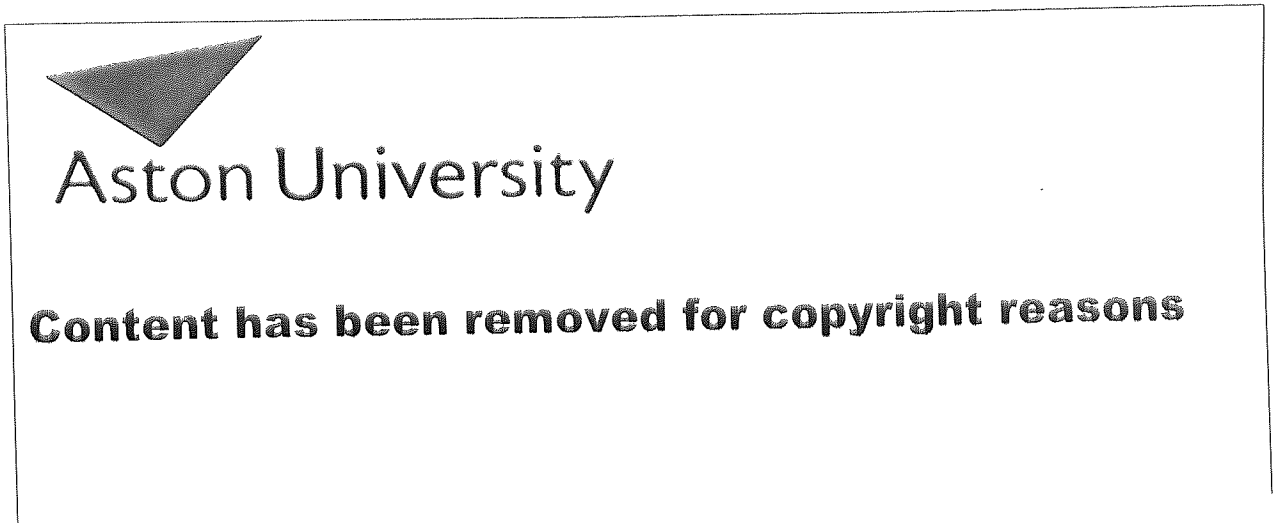
1. *A lack of agreement on terminology* (with no standard model to describe the process, products and personnel involved in software development).
2. *A poor understanding of product quality* (no standard measures of program performance or quality).
3. *The poor quality of cost data* (since it remained difficult to find any large body of data on cost by product or function that can be used on later projects).
4. *The non-quantitative nature of certain factors* (influence of "foggy requirements", "proficiency" of staff, "quality" of management).

While one might generously suggest that the development of structured methodologies in the 1970s and 1980s has partly helped to solve point '1', it is far from clear whether any of the other points have been significantly resolved or even properly addressed. Farr & Zagorski's method in approaching this problem was to:

a) identify factors which could logically influence cost;
b) collect data on these factors to support or reject their influence;
c) build a model which integrates these factors.

Parts (a) and (b) were approached by conducting a survey which asked project managers within SDC questions such as "Why did you over-run your budget?" and

Table 5.3 : Three regression models predicting man-months from a series of factors (cf. Farr & Zagorski, 1965, p174)



"Why did your program cost more or take longer to develop than another program that appears to be similar?" The results were classified into six categories (*ibid*, p170). Only programs within SDC were studied, which meant they were operations, utility and support programs with military application. The questionnaire asked for data on 15 sorts of cost information (time to develop, number of personnel, etc.) and 93 measures which might affect cost (rating of system complexity, number of agencies required to agree a design, etc.). Information was received on 27 projects. The method to generate the model in part (c) was then (*ibid*, p172):

1. Reject factors which had low variance or had some identity with other factors (around 10 were dismissed here).
2. Produce a correlation matrix and reject factors with low correlations to cost or which had no intuitive appeal (around 40 were dismissed).
3. Use knowledge of systems development, intuition and experience to filter the list (two groups were produced with 15 "most preferred" factors in one group and 21 "satisfactory" in another).

4. Use multivariate linear regression analysis to produce a model and reject factors with a very low assigned weight.

Large programs with very high counts were also rejected because they overly influenced the early regression analysis. Iteration at step 4 produced three equations to model cost in terms of man-months to design, cost and test a program (see Table 5.3). Although Farr & Zagorski admit that the models produced were not particularly accurate, the most dominant relation was found to be the number of program instructions defined as X_1, X_6 or X_{11}. Eliminating the six largest programs strengthened the relationship. The second significant factor was programmer experience (X_8 and X_{13}) which reduced cost, but Farr & Zagorski acknowledge that this factor may be difficult to measure meaningfully.

There are a number of problems with this study, however. Firstly, one could doubt the veracity of the results of a survey which asked such direct - and notably hostile - questions of project managers. Presumably, if the study aimed to find those factors which made projects expensive, Farr & Zagorski aimed their survey at project managers who had had the sharp experience of failure. In this mood, answering questions such as "Why did your program cost more or take longer to develop than another program that appears to be similar?" would undoubtedly produce defensive responses. This is probably one reason why such a large number of cost factors were received. Secondly, once the model has been produced, Farr & Zagorski do not make any comment about the fact that delivered documents (factor X_3) appears in all three equations. This factor is seen as a major plus for the FPA Mk.II model. Finally, it is possible that Farr & Zagorski's model could be reduced to a model which contained only three X factors, as in:

$$\text{Effort} = a.X_3 + b.X_{14} + c.X_{15} + d$$

where,

X_3 = Number of document types delivered to the customer

X_{14} = One or more of the dominant program instruction factors (X_1, X_6 or X_{11})

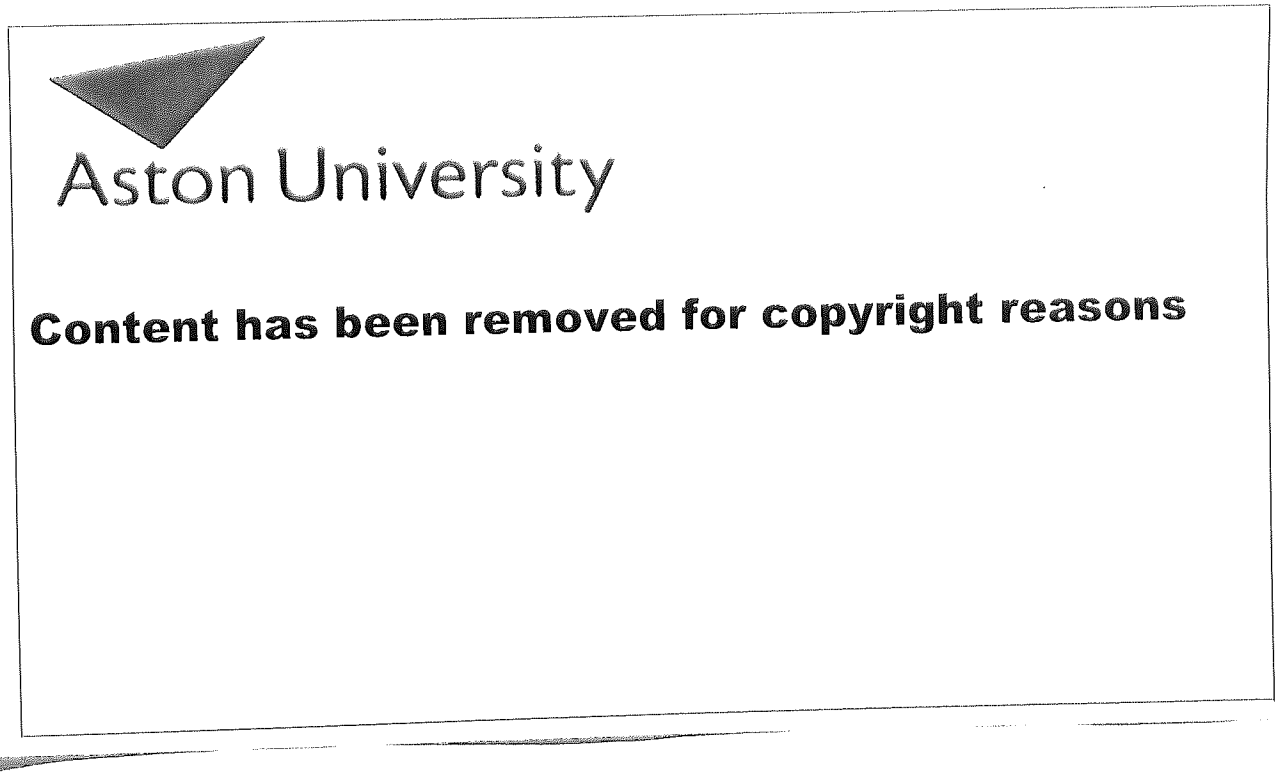
X_{15} = One or more of the dominant programmer experience factors (X_8 or X_{13})

This would not only make best use of the deductions Farr & Zagorski make for their method but also lead to a simpler model. Since Farr & Zagorski do not provide the

database from which the models are derived it is not possible to assess the performance of their models or the model given above.

Other studies have also sought to deduce linear models of effort (e.g., Nelson, 1966, Tausworthe, 1981) and can be criticised for similar reasons: Is a linear model the best representation of the way in which effort varies with system size? Does there need to be so many factors in the model? What is clear, however, is that the method of collecting and then assessing a number of factors remains the classic approach to developing a SCE model.

Figure 5.2 : Putnam's Norden-Rayleigh manpower curve (cf. Putnam, 1978, p349)



5.2.2 *SLIM (Putnam, 1978)*

By the end of the 1970s linear models had been almost universally replaced by non-linear models (e.g., Walston & Felix, 1977; Herd *et al*, 1979; Bailey & Basili, 1981). One of the most controversial of these non-linear models is the manpower model developed by Putnam (1978). The model (and resultant tool) is called the *S*oftware *LI*fe-cycle *M*ethodology, or SLIM. Putnam (1978) sought to develop a macromethodology which can accurately estimate the cost of reaching critical

milestones in terms of manpower, money and time. Furthermore, Putnam sought to answer the question of how far a manager can "push" a project before it begins to break down. So what is Putnam's model? Based on Norden's (1970) suggestion that homogeneous R&D projects have a manpower that varies approximately at the same rate as a Rayleigh curve, Putnam attempts to fit the same (manpower) curve to the software development process (see Figure 5.2). The Rayleigh curve is defined by:

$$y = 2.K.ate^{-at^2}$$

where,

$$a = \frac{1}{2.t_d^{\ 2}} \qquad \text{(or, the "pace" of development)}$$

The area under the curve, K, is the total effort of the project; while t_d represents the point at which the development effort has reached a maximum, i.e., its point of release. By studying data on the U.S. Army's Computer Systems Command (CMC) systems, Putnam found that projects tended to fall on a constant difficulty gradient defined by (*ibid*, p350):

$$\text{Difficulty}, D = \frac{K}{t_d^{\ 2}}$$

The magnitude of the gradient, Putnam suggests, was found to be dependent on the disorderedness (entropy) of the system. By plotting project difficulty against productivity Putnam found that when difficulty was small the systems were considered "easy", and vice versa. This, Putnam declares, is the "missing link" between an input (manpower and development time) and the output (quantity of source statements)(*ibid*, p350). For instance,

- if total effort, K=400 person-years, development time, t_d=3 years, then difficulty, D=44.4 person-years.year^{-1};
- if K=360 (i.e., decrease in effort), t_d=3, then D=40 (i.e., a 10% decrease);
- if K=400, t_d=2.5 (i.e., the more usual compression of a schedule), then D=64 (i.e., a 44% increase!).

So, what is a feasible effort-time region to aim for? Plotting K, t_d and D shows that as t_d decreases, D increases sharply. For the U.S. Army's CMC systems, Putnam

found three distinct values of D and suggested there were possibly 6 or 8 more values to be uncovered. The three known values were 8, 15 and 27, where,

8 => entirely new system with many new interactions and interfaces
15 => new stand-alone system
27 => rebuild or composite system with high re-use

The estimates of K and t_d are related to the product in terms of source statements (S_s) by observing that the area under the coding rate curve is the number of source statements produced at time, t. By integration and substitution of the productivity factor, it is deduced that (*ibid*, p353):

$$S_S = C_k . K^{1/3} . t_d^{4/3}$$

where,

C_k = a constant relating to the 'state of technology of the human machine system'

Putnam (1978) argues that this equation is a quantitative description of Brooks' law in that adding people is a very high cost way of accelerating a project. E.g., with S_s = 100,000, if t_d decreases from 5 to 3.5, then the effort involved (K) increases 5-fold from 5 to 25 person-years. Putnam's effort-time trade-off law, therefore, states that if S_s and C_k remain constant, then,

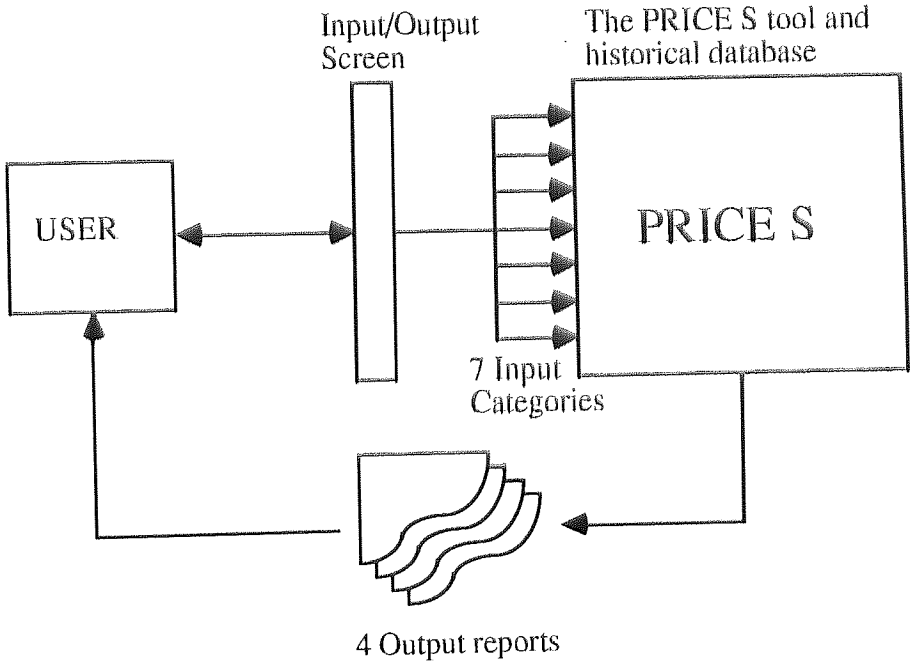
$$\text{effort} \propto \frac{1}{\text{time}^4}$$

It is this inverse fourth-power law that has come under most criticism. In later studies Putnam offers evidence to support this relationship (e.g., Putnam *et al*, 1983; Putnam & Putnam, 1984), but in each study the technology constant, C_k, had to be calculated for the completed project in order to show the trade-off law held. In other words, the required C_k constant was calculated before the SLIM model was applied. Since project managers are unlikely to be able to use this method when calculating their own project characteristics, Putnam's analysis does not seem to give much support to his model.

Jensen (1983, 1984) proposes a model similar to Putnam's but where the technology constant is found by multiplying a series of technology factors based on Boehm's (1981) COCOMO model (see §5.2.4). Conte *et al* (1986) also question

how realistic the Rayleigh-curve is, since it does not seem reasonable to suggest that a project begins with no personnel and rapidly increases. It is more likely that the project begins with a (perhaps) small project team which then expands as the project progresses. Expanding the technology constant to a full set of adjustment factors and deducing - rather than imposing - a model on the development process characterises the SCE models which followed SLIM.

Figure 5.3 : A representation of the PRICE S estimating tool



5.2.3 *PRICE S (Freiman & Park, 1979)*

The *PRICE* of *S*oftware model (PRICE S) was developed in the late-1970s by Freiman & Park (1979) but the literature on this model is an example of where the (commercial) tool rather than the model is described. PRICE S provides estimates of cost and schedules based on historical data which is assumed to have only three key development phases (design, implementation, and test and integration). The model has four modes of operation: normal (for new projects); resource calibration; application calibration, and; design-to-cost. Both types of calibration mode estimate projects based on extrapolating from historical data (of, presumably, similar projects). The design-to-cost mode estimates size and schedules feasible within given cost constraints.

The model (see Figure 5.3) produces estimates by gathering information from the user in the form of seven parameters and supports the estimates by producing four management reports. The seven principal categories of PRICE S input are:

1. *Project magnitude* (INSTRUCTIONS). In terms of the number of delivered, executable, machine-level instructions (DEMIs), rather than lines of code. Freiman & Park reason that since the effort to design, test, integrate and document software accounts for 70-80% of total cost is strongly driven by functionality, then DEMIs will summarise this better than LOC.
2. *Program application* (APPLICATION). The character of the project, defined by function with an assigned weight, from operating systems dealing with many interactions and with strict reliability and timing requirements (weight=10.95) to mathematical operations dealing with routine mathematical calculations (weight=0.86). It is not clear where these weights are derived from.
3. *Level of new design* (NEW DESIGN/CODE). An adjustment made to the effective number of DEMIs on the basis of the fraction of new design or code. Freiman & Park suggest that even off-the-shelf packages are not entirely free of work since a certain amount of time must be spent in familiarisation and modification of the package.
4. *Resources* (RESOURCE). Matching available skills to required tasks, seen by Freiman & Park as the most significant consideration for any cost estimating model. Some values are given but users are encouraged to extrapolate from their own historical database by running the model backwards in calibration mode - a feature called ECIRP. Resource values typically range from 2.5 to 4.0 and, once established, can typically remain constant unless the company radically changes its development environment.
5. *Utilisation* (UTILISATION). Hardware constraints such as speed and memory. These constraints effect the cost of design and debugging. Values typically range from 0.7 for airborne applications to 0.9 for space systems. This range does not seem particularly wide, however, given the different nature of the systems.
6. *Customer specification and reliability* (PLATFORM). Questions about the testing and end-use of the system, that is to say, where and how the system will be used. This factor directly affects the calculation of cost and schedules and summarises quality constraints such as transportability, reliability, testing and documentation. Values typically range from 0.6-0.8 for internally developed software, 1.7 for commercial avionics systems to 2.5 for manned space systems.

7. *Development environment* (COMPLEXITY). Taking into account unique project conditions which complicate a project. Freiman & Park warn that 'reasonable care' is needed to ensure there is no double-accounting between this and other categories. but give no indication as to how this can be guaranteed. The complexity factor is assigned 1.0 and decreases for simple, non-defence contracts, and increases with complex projects.

The model is calibrated to a particular year and uses forecasts of inflation to adjust the final output and the growth of technology which is bringing down the cost of DEMI's. PRICE S can also work with partial data and provides estimates for those input categories which are missing, including INSTRUCTIONS. Finally, four reports are produced with the output:

1. *Resource/Complexity Sensitivity*. Giving a 'What if?' assessment by varying the RESOURCE and COMPLEXITY parameters about their input values.
2. *Instruction/Application Sensitivity*. The same assessment as in '1' for the INSTRUCTION and APPLICATION parameters.
3. *Schedule Effort Summary*. Comparing the projected costs and effort with typical values on the basis of project scope, schedule inefficiencies and inflation rates.
4. *Monthly Program Summary*. A breakdown of total project effort and cost into phase-by-phase monthly summaries.

The advantage of PRICE S, Freiman & Park (1979) suggest, is that the model is process-driven and does not rely on a single cost-estimating relationship (CER). Rather than a single U=a.K model, PRICE S develops a "family" of CERs for each specific application by adjusting the relevance of each of the parameters. However, one could question how easily some of the influences would be to estimate at the outset (such as categories 1 and 4). Furthermore, although PRICE S is clearly heavily dependent on the historical database, Freiman & Park make no mention of their efforts or recommendations for controlling this database, building it up, ensuring the data is clean and uncorrupted, or making sense of the results. One could doubt, therefore, the ability of any manager to usefully employ the model. No evidence is given for the accuracy of the model.

5.2.4 *COCOMO (Boehm, 1981)*

The most well-known estimating model is called the *CO*nstructive *CO*st *MO*del, or COCOMO, and was developed by Boehm (1981, 1984) at TRW. The model

estimates eight major project activities from requirements analysis, through design, programming and testing, and including verification, validation and documentation. The model was derived from a database of 63 projects developed at TRW from 1964 to 1979 and which included a wide variety of applications (business, scientific, etc.) and programming languages (ASSEMBLY, FORTRAN, COBOL). The key to the model is that the project 'enjoys good management' and follows a well-structured development plan. Boehm's argument here is that no model can estimate where there is poor management of the resources used.

The COCOMO model has three levels of complexity (Basic, Intermediate, Complex) and operates in three modes (organic, semi-detached, embedded). The difference between each model is the level of detail required to generate an estimate; the difference between each mode is the constraints on the project being estimated. The basic COCOMO model gives estimates of man-months effort, MM, and time to develop (project duration), TDEV, as (1981, p57):

$$MM = 2.4 * (KDSI)^{1.05}$$
$$TDEV = 2.5 * (MM)^{0.38}$$

All of the parameters are deduced by regression. The KDSI parameter counts thousands of delivered source instructions (lines of code), and includes job control language, format statements, etc., but excludes non-delivered code unless it is developed with 'as much care' (review, test plans, documentation, etc.) as the rest of the system code. Estimates are in terms of direct-charged labour and where 1 man-month=152 man-hours or 19 man-days. The modes of COCOMO are defined as:

- *Organic.* Small to medium-sized projects (<32 KDSI), developed in-house with a familiar development environment;
- *Semi-detached.* Intermediate between organic and embedded;
- *Embedded.* Ambitious, tightly constrained requirements, developed on multiple sites or with new machinery.

Tightening constraints affect productivity and staffing levels. In organic mode productivity is found to be over 300 LOC.MM^{-1}, around 200 LOC.MM^{-1} for semi-detached mode and around 100 LOC.MM^{-1} for embedded mode. Although the schedule for large projects is about the same (around 24 months), in organic mode a typical project only requires 16 people, semi-detached projects require 24 people and embedded mode projects require 51 people. These effects are reflected in the change in the a parameter and b exponential as the mode changes (see Table 5.4). The

problem for Basic COCOMO is that it is perhaps overly-simple and only uses KDSI as a cost-driver.

Table 5.4 : Changes to Basic COCOMO parameters with mode (after Boehm, 1981, p75)

Mode	Effort=a.(Size)^b		Schd=c.(Effort)^d	
	a	*b*	*c*	*d*
Organic	2.4	1.05	2.5	0.38
Semi-Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 5.5 : The 15 Intermediate COCOMO cost-drivers (cf. Boehm, 1981, pp115-6)

The most well-known version is Intermediate COCOMO. Intermediate COCOMO (*ibid*, pp114-163) introduces 15 cost-drivers which adjust the nominal effort model and make COCOMO more sensitive to project-specific factors. The 15 cost-drivers are grouped under four attribute headings (see Table 5.5). Each cost-driver has one of six ratings from very-low to extra-high from which a value is assigned (from 0.70 for very low CPLX to 1.66 for extra high TIME). The product of the cost-drivers is then used to adjust the nominal effort model given by mode. The Intermediate COCOMO models for effort and duration are the same as for Basic COCOMO, except the a parameter in the effort model varies from a=3.2 in organic mode, to a=3.0 in semi-detached mode, and a=2.8 in embedded mode. Detailed COCOMO, on the other hand, differs from Intermediate COCOMO in that the cost-driver parameters are varied according to their effect on different phases and system components, allowing system and sub-system level estimates to be calculated. This amount of detail means that Detailed COCOMO has generally been ignored in follow-up studies (e.g., Miyazaki & Mori, 1985).

The most interesting property of the COCOMO models is the deduction of an exponential parameter of b>1.0 producing a diseconomy of scale which suggests that as the project size increases the effort increases at a non-linear rate. This is borne out by the reduction in productivity from organic to embedded mode. The best way to avoid the problem, Boehm (1981) suggests, is to reduce the scale of the project by either prototyping, incremental development, or pruning the wish-list (or "gold plating") of customer requirements. However, Boehm also points out that there are opportunities for economies of scale, since large projects are more likely to see the worth of investing in better equipment, programming and testing aids, etc. Banker & Kemerer (1989) suggest diseconomies of scale can be avoided by balancing project size against the use of these extra tools and personnel. The cross-over point from economy to diseconomy is found by understanding the nature of the project which has shown to be the most productive.

Using the original COCOMO database provided by Boehm (1981) it is possible to deduce the accuracy of the COCOMO models. As can be seen (see Table 5.6), there is some variability between the models. Basic COCOMO applied to projects of any mode is the worst performer, while there is little or no difference between Intermediate and Detailed COCOMO across all projects. Both of these models pass Conte *et al*'s accuracy criteria except for Pred(0.25)≥0.75 for projects in organic mode. Given that organic projects are fairly well represented with 23 data points in the COCOMO database, one might be led to infer that, either:

Table 5.6 : Accuracy of Basic, Intermediate and Detailed COCOMO by project mode (after Boehm, 1981, pp496-497)

Model	Project mode							
	All		Organic		Semi-detached		Embedded	
	MMRE	Pred	MMRE	Pred	MMRE	Pred	MMRE	Pred
Basic	0.60	0.29	0.62	0.30	0.66	0.42	0.57	0.21
Intermediate	0.19	0.75	0.21	0.65	0.18	0.75	0.18	0.82
Detailed	0.19	0.75	0.21	0.61	0.18	0.75	0.17	0.86

NB: Pred=Pred(0.25)

- there is a greater difference between organic projects than there is between semi-detached and embedded projects; or,
- the relative size of the semi-detached and embedded mode projects have "hijacked" the regression analysis, making COCOMO virtually redundant on organic mode projects.

What is peculiar about the organic projects in the COCOMO database is that they also tend to be older and smaller. This difference may be indicative of COCOMO's database containing projects which are unrepresentative of current software development. There is currently no clear method of how to maintain a SCE database and avoid these potential problems. The Semi-Detached and Embedded modes do not seem to suffer the same kinds of problem.

While the Basic COCOMO model is clearly inadequate, a typical criticism levelled at the Intermediate and Detailed versions is that there are too many cost-drivers and that the model could - and should - be simplified (e.g., Conte *et al*, 1986; Kitchenham, 1991). Given the levels of accuracy shown, however, it could be argued that a company may well be prepared to spend the time finding this extra data. Of course, a simpler model with the same level accuracy would indeed be a more parsimonious model, but it would be up to the critics to find simpler models which still have the same degree of accuracy.

5.2.5 *COPMO (Thebaut & Shen, 1984; Conte* et al, *1986)*

Thebaut & Shen (1984) propose a model which focuses on the effect of programming team size on the size of a project in order to estimate effort. The

resultant model is called the *CO*operative *P*rogramming *MO*del, or COPMO, which was revised into a generalised model by Conte *et al* (1986). Taking Brooks' (1975) proposal that adding manpower to a project makes it later, Thebaut & Shen (1984) go on to explain that:

> "...when tasks are partitioned among several programmers, the effort associated with communication must be taken into account. Brooks views this effort as being made up of two parts: that which is related to programmer training (i.e., assimilation into the team) and that which is related to the coordination of programming activity. He suggests that the former is likely to vary linearly with the number of programmers, while the latter could, in extreme cases, vary with the square of the number of programmers (p294)."

Thebaut & Shen model this situation by representing the communication paths between a number of team members (programmers and their supervisors), P. The number of communication paths can range from where the tasks are perfectly partitioned (e.g., a task which would take one person 18 months will take two people 9 months, and three people 6 months), to where there are communication paths between all members of the team. From perfect partitioning - defined by O(P) - the co-ordination effort becomes a steadily increasing function until, as Brooks (1975) has it, adding more people simply adds to the overall effort. The COPMO model has the form:

$$E = a + b.S + c.P^d$$

where,

E = effort (in person-months)
S = thousands of lines of code (KLOC)
P = average personnel level, where P=E/T
T = project duration (in calendar months)

The parameters are found by firstly finding those projects for which P≈1, so eliminating $c.P^d$ as a separate factor and allowing *a* and *b* to be found using linear regression. Secondly, with *a* and *b* known, least squares regression is used to find *c* and *d*. This method produces two versions of the model (see Figure 5.4), depending on whether the communications being modelled are simple (where P=1, so there are effectively no interactions), or complex (where P>1). Thus (*ibid*, p302):

$$E = a + b.S \qquad P = 1$$
$$E = a + b.S + c.P^d \qquad P > 1$$

Figure 5.4 : The effect of task interaction modelled by COPMO (after Thebaut & Shen, 1984, p295)

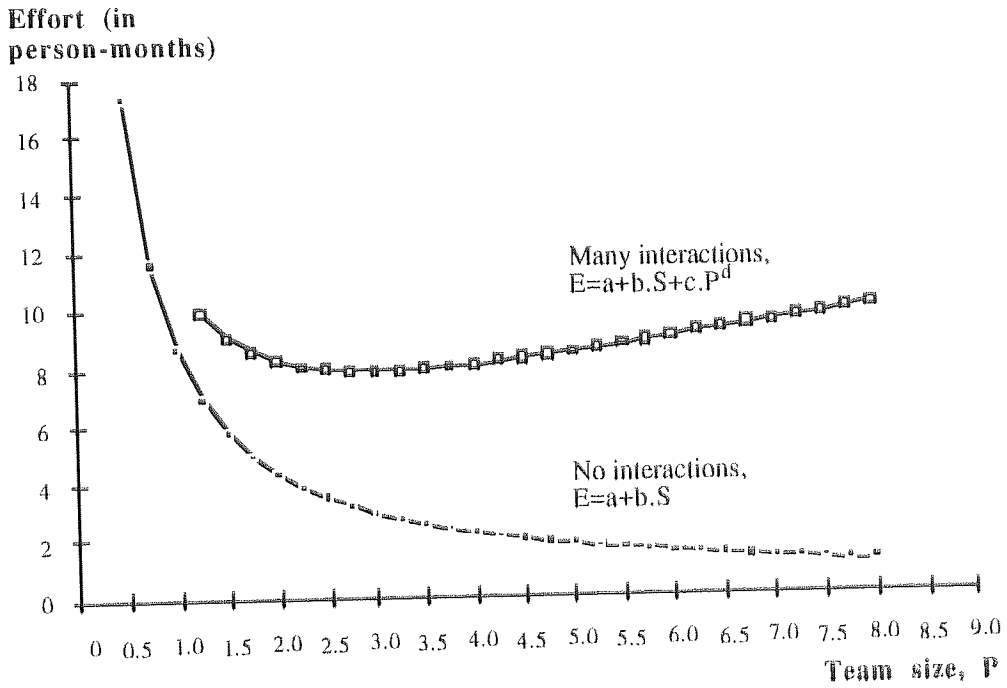


Table 5.7 : Parameters of the COPMO model (cf. Conte *et al*, 1986, p316)

Conte *et al* (1986) show how the parameters can be calculated by regression using datasets from a number of published sources. As can be seen (see Table 5.7), applying COPMO to all six datasets produces what Conte *et al* suggest is a "reasonable" performance, although below the required Pred(0.25)≥0.75 and MMRE≤0.25 criteria. The parameter c is said to define the standard cost of each communication path within an organisation, while d is said to represent the non-linear impact of co-ordinating the communication between increasing numbers of project

members. In each case, the value of d is found to imply a diseconomy of scale (where d>1.0). Conte *et al* (1986) criticise this first version of the COPMO model on the grounds that (p318):

- least squares does not necessarily give good values for Pred and MMRE;
- the model parameters are significantly different between datasets;
- the complexity of the projects is assumed to be constant but this cannot be;
- the model requires an early P value, but it would be better to have a more sensible surrogate for this input.

They attempt to overcome these problems by defining a more general version of the model. The generalised COPMO model has the form (*ibid*, p318):

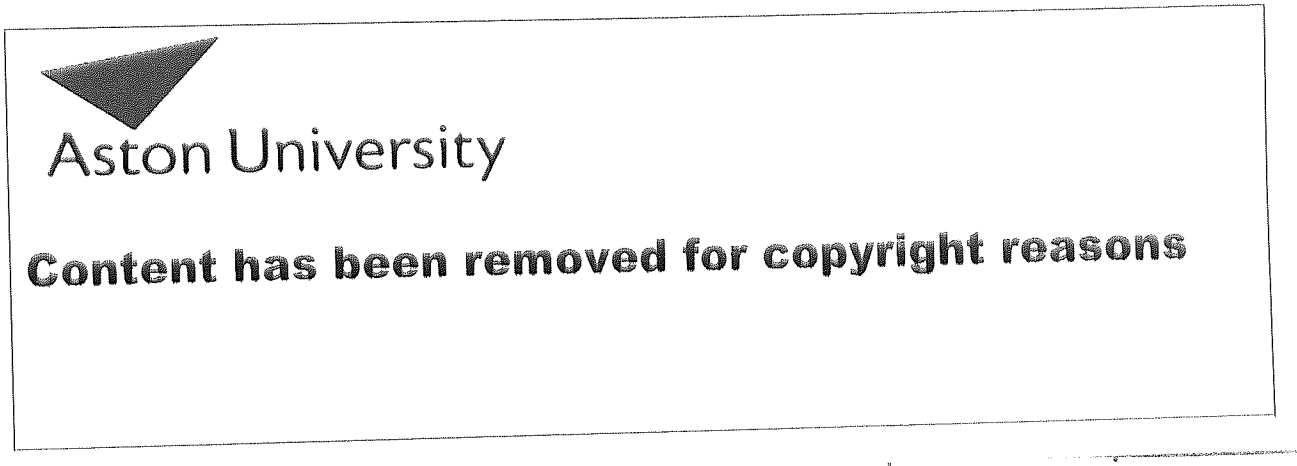
$$E = b_i.S + c_i.P^{1.5}$$

The value of a was arbitrarily set to zero because it was deemed to be relatively small in relation to the reported effort and thus insignificant to the model. Since the weighted value of d in Table 5.7 was 1.53, this parameter was also arbitrarily set to 1.5. The value of b_i and c_i are dependent on the nature of the project and the development environment. Conte *et al* attempt to approximate these effects by partitioning projects into suitable complexity classes. By assuming that complexity is inversely proportional to productivity, i.e., the more difficult the project the more time it takes to write the code, Conte *et al* divided the same six published datasets into ten classes. Since most projects had productivity rates of less than 1000 LOC per month, the tenth and highest productivity class was arbitrarily set to this figure. With values of S and P known, the values of b_i and c_i could then be deduced for each i^{th} complexity class. The overall performance of the generalised COPMO model across all six datasets was MMRE=0.21 and Pred(0.25)=0.75 (*ibid*, p321).

Finally, Conte *et al* show that the value of P can be replaced with an understanding of the general demands on a project. Boehm's (1981) 15 cost drivers are taken to adequately represent these demands. By partitioning the product result, $m(X)$, of these cost drivers into 7 classes, the same method as above can be used to deduce the value of b_i and c_i for each i^{th} complexity class (see Table 5.8). Applying this version of COPMO to the COCOMO dataset produces levels of accuracy on a par with Intermediate and Detailed COCOMO, with MMRE=0.21 and Pred(0.25)=0.78. Since productivity rates clearly follow the bands of $m(X)$, the COPMO model has the advantage that all of Boehm's adjustment factors can be replaced by an estimate of productivity. The problem is what to do with the values at the border of the classes.

A continuous function is proposed but discounted by Conte *et al* on the grounds that it implies a greater level of accuracy than is warranted, and, although no details are given, Conte *et al* admit that it doesn't work very well (*ibid*, p328)!

Table 5.8 : Application of the General COPMO model to the COCOMO database (cf. Conte *et al*, 1986, p327)



5.3 Problems for SCE tools

The models and tools so far described denote key points in the development of SCE tools. The history does not end with COPMO, however, with other models/tools addressing general estimation (e.g., ESTIMACS by Rubin, 1983; SPQR by Jones, 1986), while still others have focused on specific languages such as ADA (Boehm, 1988b) and object-oriented systems (Laranjeira, 1990), or specific issues such as estimating productivity (Pfleeger, 1991) or system size (Verner & Tate, 1992). But are they accurate? Empirical studies suggest they are not and suggest that both the models underlying SCE tools and the techniques used to develop them are flawed.

The greatest problem for SCE tools is that since their underlying models have been developed within a specific development environment, it is difficult to know whether they have captured anything universal about software development or simply produced a model that is specific to the database upon which it is based. For instance, Mohanty (1981) found that 12 SCE tools studied generated cost estimates from $362 500 to $2 766 667 and schedules of 13.77 to 25.8 months given the same outline description of a project. Mohanty (1981) claims that models which give very high estimates are likely to have a number of tightly constrained, high quality projects in the database (what Boehm, 1981, called

"embedded"). It was only in later SCE models where a quality factor was introduced that the cost of external constraints began to be realised.

Such variances in estimates are typical when comparing estimating tools, and later studies have found equally disparate results (Rubin, 1985; Kemerer, 1987, Kusters *et al*, 1991). These studies all suggest that it is essential to calibrate an estimating tool from one development environment to another (Jeffrey & Low, 1990b). "Calibration" is the process of assuming that the underlying model is correct and adjusting the parameters either by "tweaking" the values until the output is deemed to be reasonable, or, by building a new set of historical data and deducing the required parameters. Miyazaki (1991) even went so far as to improve the performance of COCOMO at Fujitsu by dismissing some of the cost-drivers that correlated poorly to cost.

In either case it would have to be assumed that the model is meaningful and the parameters adequately represent the behaviour of software development within an organisation. This assumption can often be doubted. For instance, if the general effort and schedule models are defined as Effort=a.(Size)^b and Schedule=c.(Effort)^d, then Kitchenham (1992) points out that $b \approx 1.000$ and $d \approx 0.333$ for 12 datasets studied as part of the ESPRIT-funded MERMAID project (P2046). In other words, where the model is built within a single organisation, a linear model for effort in terms of size is likely to be sufficient, while the difference in the d quotient from $^1/_3$ is insignificant across almost all published models and datasets. This would suggest that studies into diseconomies of scale suggested by both COCOMO, SLIM and COPMO have been misguided. Furthermore, Kitchenham suggests that adjustment factors have much less influence than had previously been thought. Specifically:

- looking at the COCOMO model dataset, only programming language ($\rho<0.001$) and working environment ($\rho<0.05$) were found to be significantly related to productivity, while language experience ($\rho<0.05$) was the only personnel experience factor which showed an increase in productivity as the level increased;
- looking at the FPA scores within the MERMAID dataset, only factors relating to data communications ($\rho<0.05$) and online systems ($\rho<0.05$) were related to productivity (TCFs 1, 6 and 8).

The COCOMO model suggests effort increases when timescales are altered, while SLIM suggests effort increases when timescales are decreased, and decreases when timescales are increased. However, Kitchenham (1992) shows that by defining effort compression, *eff_comp*, as (p216):

Figure 5.5 : Duration and effort compression for Basic COCOMO (N=58)(cf. Kitchenham, 1992, p216)

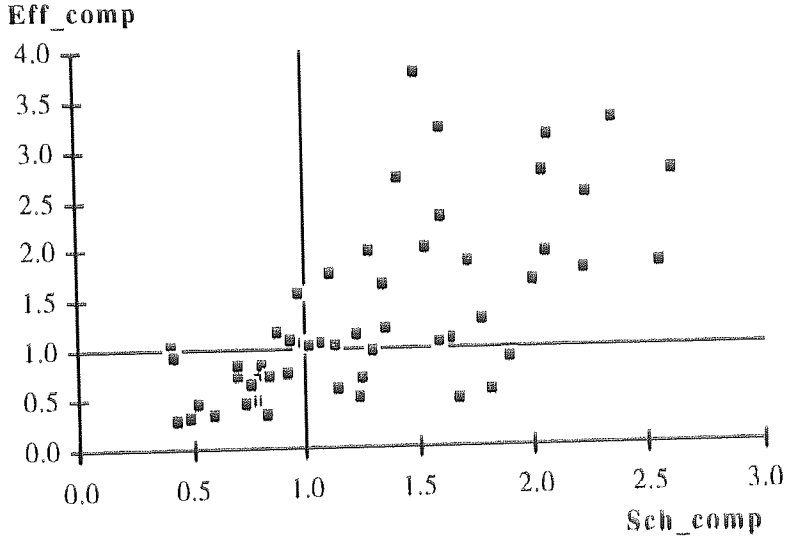
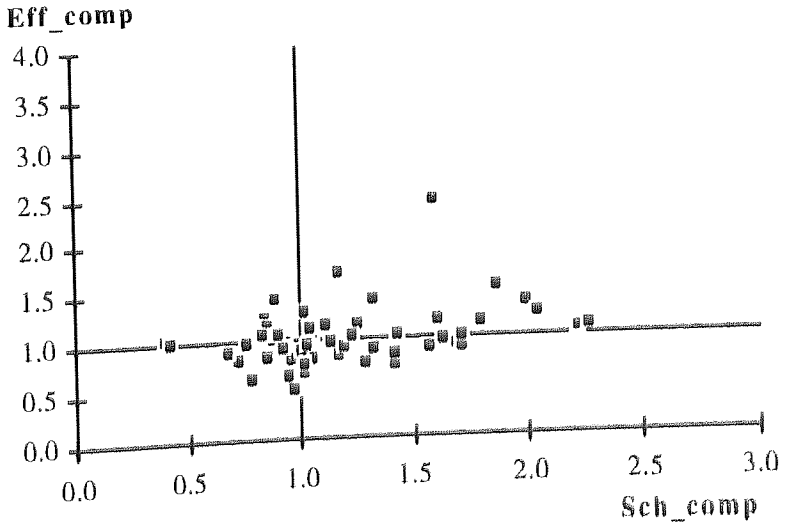


Figure 5.6 : Duration and effort compression for Intermediate COCOMO (N=58)



$$\text{eff\_comp} = \frac{\text{actual effort}}{\text{estimated effort}}$$

and schedule compression, *sch_comp*, as:

$$\text{sch\_comp} = \frac{\text{actual duration}}{\text{estimated duration}}$$

a scatter-plot of the COCOMO dataset shows that a number of projects were both effort and schedule compressed (see Figure 5.5). A value of less than 1.0 indicates compression, while a value of greater than 1.0 indicates expansion. In other words, the projects in the bottom-left corner of the scatter plot show effort decreased on a project where timescales were compressed, an effect which contradicts both COCOMO and SLIM models.

However, this criticism is weakened somewhat when it is realised that the scatter plot is based on Basic COCOMO, a model which Boehm (1981) acknowledged was not a very accurate model. The same scatter plot for Intermediate COCOMO - which was considerably more accurate - shows a less dramatic picture. As can be seen (see Figure 5.6) most estimates are clustered around the point where effort and schedule compression are near 1.0 (i.e., actuals close to estimates). There are virtually no data points in the extreme bottom-right corner, and those which are considerably amiss are where actual >> estimate. In effect, this follows a reasonable β-distribution where considerable over-estimates (low compression) are much more likely than considerable under-estimates (high compression).

The structure of SCE models may remain in doubt, therefore, but the nature of the adjustment factors which correctly identify what makes a project expensive remains open. In particular, Kitchenham (1992) does not seem to have removed the (intuitive) notion that the experience of the personnel involved in the project is the dominant and over-riding consideration in the equation of cost and duration. Given these mixed criticisms, it would perhaps be surprising if any project manager would put their faith in such technology. What is not clear, however, is whether the endeavour should be abandoned, whether the solution is to build better estimating models or whether project managers are looking for different sorts of tools. This will be the subject of the next chapter.

6. A survey of large UK companies

"The reasonable man attempts to adapt himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man." G.Bernard Shaw (Maxims for Revolutionists in 'Man and Superman')

Summary: *This chapter presents the results of a survey of large UK corporations and computing companies. The survey aimed to discover whether: a) estimation was seen as a problem; b) estimating tools were in use; and, c) companies had the sort of development environment within which an estimating tool could be developed or calibrated. It will be seen that while most respondents see estimation as a problem and could develop/calibrate an estimating tool, less than a third actually do.*

The theoretical problems which have been discussed in the last three chapters have shown that virtually all aspects of developing metrics to measure software development or building tools to estimate cost have suffered a number of criticisms:

- Metrics, because it is often not clear how the measurement relates to the property being assessed. This is particular true for complexity and structure metrics. Even though it may be accepted that KBSs are essentially like any other type of software development project, this is of little help when the metrics to control and manager the project appear to be flawed.
- Software cost estimating (SCE) tools, because there is no well-formulated method of building the underlying models which has proven itself to generate meaningful and useful tools. A "shotgun" approach to discovering the underlying cost-drivers produces models which are statistically significant but semantically dubious.

Given these criticisms, it would perhaps be surprising to find any project managers using such tools. Indeed, this would appear to be close to the current state within industry. A survey by the consultancy Information Processing Limited (IPL), found that of 36 UK organisations mainly in the real-time software industry, only 27% used 'any type of estimating tool' (IPL, 1989). In larger surveys carried out in The Netherlands, less than a quarter of companies used estimating tools (Heemstra & Kusters, 1989; van Genuchten &

Koolen, 1991). The most pessimistic conclusion based on this evidence is that useful metrics, accurate tools and the whole idea of controlling software development are essentially impossible. This will be called 'Option Zero', in which nothing of any benefit can be derived from metrics and so all research would stop here.

The rest of this thesis, however, will set out to prove that 'Option Zero' is false. Moreover - given the place of this research within Project IED4/1/1426 - it will be argued that not only can useful metrics, models and tools be developed, but a similar set can be extended to the development of KBSs (and thus hybrid systems). Given the current state of SCE tools, it may be difficult to believe that the technology has advanced very far from 'Zero', but the point here is to show that 'Option Zero' is a function of misdirected models failing to show their usefulness to project managers. The basic metric→model→tool approach, however, remains sound.

The key to developing and using a SCE tool is that there are no barriers to their use. Specifically, the ability to develop and use SCE tools is interpreted here as requiring a company to have a theoretical, practical and political framework in place. The theoretical component of this framework suggests that applying a model to a complex system assumes there are relationships within the system which are stable enough to be modelled. In effect, this point follows Browne & Shaw's (1981) axiom (see again §2.1), where it must be assumed that the subject under study is not chaotic but amenable to some form of understanding. In this case, it must be assumed that there is structure to the software development process. This will be assumed if a company has a structured methodology in place. Clearly, a company which states it has a methodology does not guarantee that it is precisely followed, while a company which denies the use of a methodology may have a very good understanding of their development process. However, since it can be doubted that any company has full control over any but the smallest projects, the assumption being made here is that a company which recognises the existence of a method at least provides a theoretical basis from which to model the development process.

The practical component of the framework suggests that to build an in-house model or calibrate a commercial tool it is necessary to collect data on the cost and duration of projects and the structure (and thus quality) of the resultant code. Kitchenham (1992) suggests that local (in-house) models are likely to be more useful but this would require a company to invest in the costly process of building a database of project information. As such, it can also be suggested that companies are more likely to use commercial tools because this costly database has already been collected and the practical use of the tool can be studied immediately. Adjusting the parameters of a commercial tool without some project data would be to effectively guess how the database of the tool related to a particular company. In this

case, since person-hours expended on a project is the first concern of most SCE tools, the minimum practical requirement will be assumed here to be the collection of time data. Again, it is questionable whether collecting data means it is subsequently used to test or adjust the accuracy of the SCE model or tool in use, while the lack of an historical database does not rule out ad hoc but effective adjustments to a SCE model. In both cases, however, it must be assumed that the accuracy of the model requires a comparison between the SCE model or tool with some actual data.

The political component of the framework suggests that given the apparent resistance Avison *et al* (1992) found to changes in the way systems developers work (see again §2.5), it is possible that the lack of use of SCE tools is a political rather than a theoretical or practical problem. This would mean that although estimating technology is useful and accurate, it would be a long time before their use is seen as acceptable by those project managers that would actually benefit from them. This would stunt the commercial exploitation of research into metrics but would not denigrate the potential usefulness of the technology.

The assumption being made here, therefore, is that a company which already uses technology closely related to SCE tools is likely to have already confronted and overcome political resistance to the use of similar tools. In this case, it will be assumed that companies using project management tools (such as PMW, etc.) are those whose resistance to SCE tools will be lowest. The connection between project management and SCE tools is simply that both seek to be useful early in the life of a project. It need not be assumed here that SCE tools are perceived as being identical to project management tools in any sense. What is important is that the use of tools to support project management tasks has already been seen as "acceptable".

The fact that 'Option Zero' is false will therefore be established by demonstrating that the following four key propositions are correct:

1. Project managers recognise the need for estimating tools. In particular, SCE tools assume that project managers see estimation as a problem, if not, then the technology becomes redundant.
2. There are no prohibitions to the use of SCE tools. Such as, the cost of collecting the data upon which to develop a SCE tool outweighing marginal increases in accuracy. This problem would demand the SCE tool places itself within everyday practice.
3. There is no other set of tools which can adequately address the issue of estimation. Otherwise, although SCE tools may be feasible, they would have been found to be unnecessary.

4. The techniques for developing metrics, models and tools for conventional systems development can also be extended to KBS development. Otherwise, the belief that the metrics approach is universally applicable would be undermined.

In other words, if project managers do see estimation as a problem, they ordinarily collect the right sort of data to develop or calibrate a SCE tool, no other tools have taken on the rôle of estimating-support tool and the same method is applicable to KBS development, then it would seem reasonable to conclude that 'Option Zero' is false and that the current problems with metrics, models and tools can be put down to the informal, ad hoc and even unscientific approach being taken by researchers in this field (MacDonell, 1991). These propositions require empirical support and this chapter will describe a survey which set out to provide the initial data. The survey began by attempting to validate seven propositions which would explain why SCE tools would not (and possibly could not) be used by project managers. The seven propositions are:

1. Estimating is not as problematic as has been previously reported, so the need for any kind of support tool is negated.
2. Estimating tools have recently become commonly used, so previous surveys are out-of-date.
3. Some other tool is used to support estimating decisions (e.g., a project management tool such as PMW) and is perceived as replacing the need for a specific estimating tool.
4. There is a failure of publicity, such that tools which could be used by development managers are unknown to them.
5. There is no framework in place from which to calibrate a commercial package or develop an in-house model, so the technology is unusable.
6. Where needed, a model is developed which is directly relevant to the company concerned and not purchased from outside.
7. Estimating tools have been used but found to be inadequate, so the technology is not perceived as being useful.

The survey was designed with the expectation that the first two propositions were false: i.e., there is a problem with estimation but estimating tools are not used. Otherwise, there would be little point in investigating propositions 3-7. The survey focused on the use of SCE tools because the tool is potentially more useful to a project manager than the raw metrics data or SCE model. As was explained in the previous chapter, however, all three elements are part of the same (SCE) technology. The rest of this chapter will describe:

- how the companies targeted by the survey were defined
- a definition of the types of companies that responded to the survey
- the results of the survey in answering the above seven propositions

6.1 Devising a survey of large UK companies

The target sample were business organisations and public utilities with around 2000 UK employees and computing companies with over 500 UK employees. It was felt that this definition would target software departments likely to invest the most money in supporting project development and thus those companies which would be more likely to use estimating tools. Such a population, therefore, would provide more information about their use than would a sample of all UK organisations/businesses. This would also overcome the problem that if so few companies use estimating tools, then not enough information would be gathered from any sample to test the above seven propositions.

The Higher Education careers book ROGET 91 was used to draw up a list of companies. ROGET 91 lists UK companies that employ graduates and gives a brief description of the company and its area of business. In this way, it is intended that graduates can make a more informed choice as to which company to apply for employment. The company description was also used to deselect any company which satisfied the size criteria but made no mention of a core DP department. From over 2000 entries, a list of 115 companies was produced.

The questionnaire itself (see Appendix A) consisted of 19 questions with a combination of multiple-choice and written answer questions. An example of the former asked "How many people are there in your department involved in software development?" and provided a choice of seven boxes (1-50, 51-100, ..., 301+). An example of the latter asked "What other types of data do you normally collect?" and provided a number of lines available for free-form text answers. The questionnaire was piloted and a number of errors removed.

While the questionnaire was still being developed, introductory letters were sent to each target company asking them to respond only if they did not wish to participate in the survey. This was meant to forewarn the company, and also to bias those who were undecided towards receiving the survey. The possibility was that until they saw the survey they may not be interested in participating. However, twenty-five declined to take part at this point. A few weeks later, questionnaires were sent to the remaining 90 companies. Fifty-four eventually replied to the survey (giving a good response rate of 47% of the original target sample). The results were processed firstly using a BASIC program developed especially for the survey and then with the spreadsheet package Excel running on an Apple Macintosh micro-computer.

Figure 6.1 : Characterising respondents by Business Area

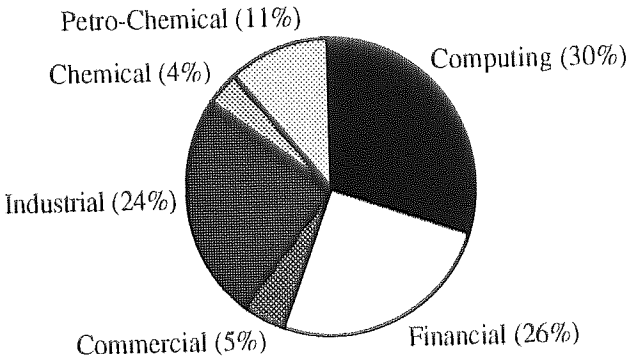
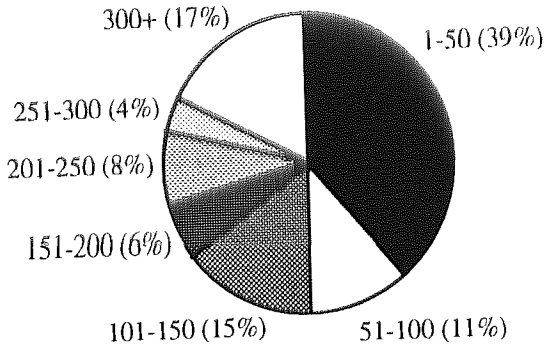


Figure 6.2 : Size of software development departments



6.2 Defining the survey respondents

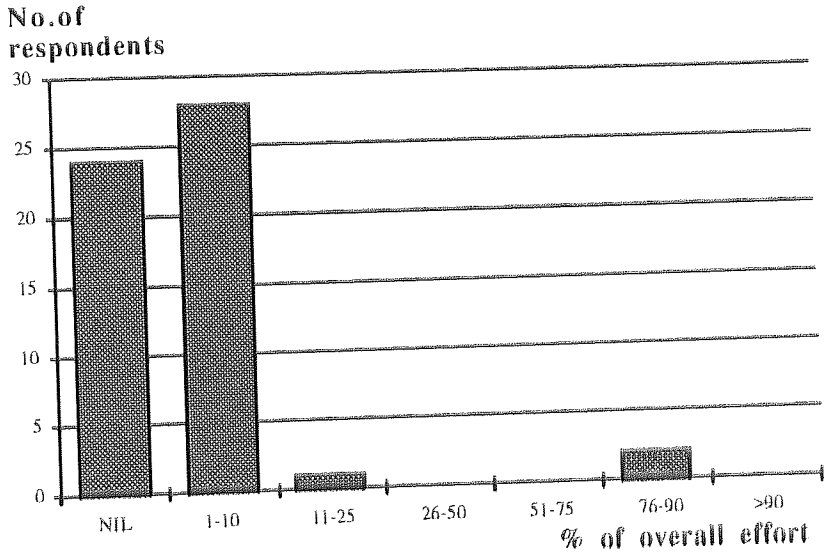
Using the description given in ROGET 91, the 54 responding companies can be divided into six business areas (see Figure 6.1):

- Financial (banks, assurance companies, etc.);
- Commercial (property, foodstuffs, etc.);
- Industrial (plant, engineering, etc.);
- Chemical (producers of dyes, pharmaceuticals, etc.);
- Petro-chemical (oil exploration and/or refinement, etc.);
- Computing (including telecommunications and independent computing subsidiaries of other companies).

As can be seen, respondents were mainly from the computing (30%), financial (26%) and industrial (24%) sectors. The size of the software development department varied widely

(see Figure 6.2). By business area, software development departments in the commercial, chemical and petro-chemical companies tended to be smaller than the rest of the sample (82% less than 50 people). Financial companies had the largest departments (44% with over 150 people).

Figure 6.3 : Percentage of overall development work devoted to KBSs



Knowledge based systems (KBSs) were developed by 56% of the respondents (30 out of 54) but, except for three respondents, tended to be a small percentage of their overall work (1-10%, see Figure 6.3). Where carried out, however, this KBS work seems to be well-established within the department, with 50% of companies reporting that the percentage of development person-years spent on KBS rather than conventional development is an increase on previous years, 10% reporting a decline and 40% reporting the percentage had remained the same. This also suggests that the need for hybrid metrics and models will continue to increase over the years. The conclusion seems to be that although the organisations may be deemed to be "large", the departments themselves are of variable size and can be taken to represent a good cross-section of all those developing software. Based on this sample, then, the propositions and findings are discussed in the following sections.

6.3 Survey results

The survey was designed to test the validity or otherwise of seven propositions which, on their own or in combination, might explain why estimating tools are not widely used in large UK companies. The results for each of the seven propositions are detailed below.

6.3.1 *Proposition 1*

Proposition 1 stated that: Estimating is not as problematic as has been previously reported, so the need for any kind of estimating tool is negated. The evidence suggests this statement is false since, when asked "Do you see estimation as a problem?", 91% responded "yes" while only 9% responded "no". Therefore, the vast majority of respondents see estimating as a problem. Such is the strength of this response, however, that it becomes even more important (and yet difficult) to answer the question: If estimation is a problem, why are tools specifically designed to support the task not being used?

6.3.2 *Proposition 2*

Proposition 2 stated that: Estimating tools have recently become commonly used, so previous surveys are out-of-date. This statement also appears to be false. A series of four questions asked whether respondents were using estimating tools, and if so which. When asked "Have you developed any Software Cost Estimating tools in-house?", 13% replied "yes" while 87% replied "no". Two further questions asked which, if any, commercial estimating tools were in use: 24% used a commercial tool, while 76% did not. In summary, only 30% use any form of estimating tool, with 17% of companies using a commercial tool only, 6% using an in-house tool only, and 7% using both types.

Table 6.1 : Comparison of survey results

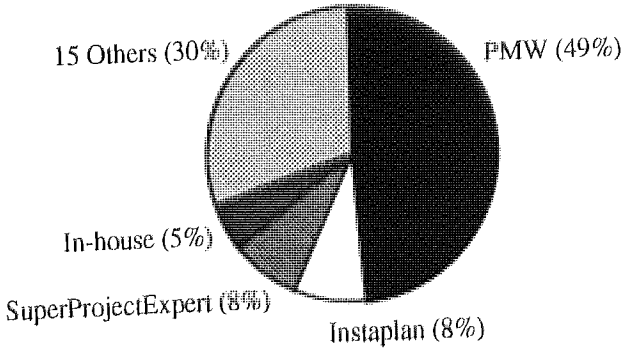
	IPL (1989)	Heemstra & Kusters (1989)	This survey (1991)
Location of sample	UK	The Netherlands	UK
Number of respondents	36	597	54
Response rate	-----	22%	47%
Use of an estimating tool	23%	14%	30%

These results can be compared (see Table 6.1) to those of the two earlier surveys (IPL, 1989; Heemstra & Kusters, 1989). Although still in the minority, this survey finds a higher use of estimating tools within the target sample. This would seem to confirm the selection criteria whereby larger companies were targeted in the belief that they would have more money to invest in support tools, such as estimating tools. The suggestion that Software Cost Estimating (SCE) tools are suffering from some problem which appears to militate against their use now becomes apparent. Specifically, even when targeting companies most likely to be using estimating tools (as in this survey), there is found to be a huge difference between recognising the problem of estimation and using tools designed to support project managers in solving the problem.

6.3.3 *Proposition 3*

Proposition 3 states that: Some other tool is used to support estimating decisions (e.g. a project management tool such as PMW). The evidence suggests that this statement is true, since, when asked "Do you use a Project Management tool?", 94% replied "yes" while only 6% replied "no". Since estimating begins by analysing the tasks of a project, then this statistic by itself could explain why estimating tools are not commonly used. Of 19 recorded, the most popular tool is PMW (49%) followed by SuperProjectExpert and Instaplan (both 8%). (See Figure 6.4.)

Figure 6.4 : Relative use of project management/planning tools

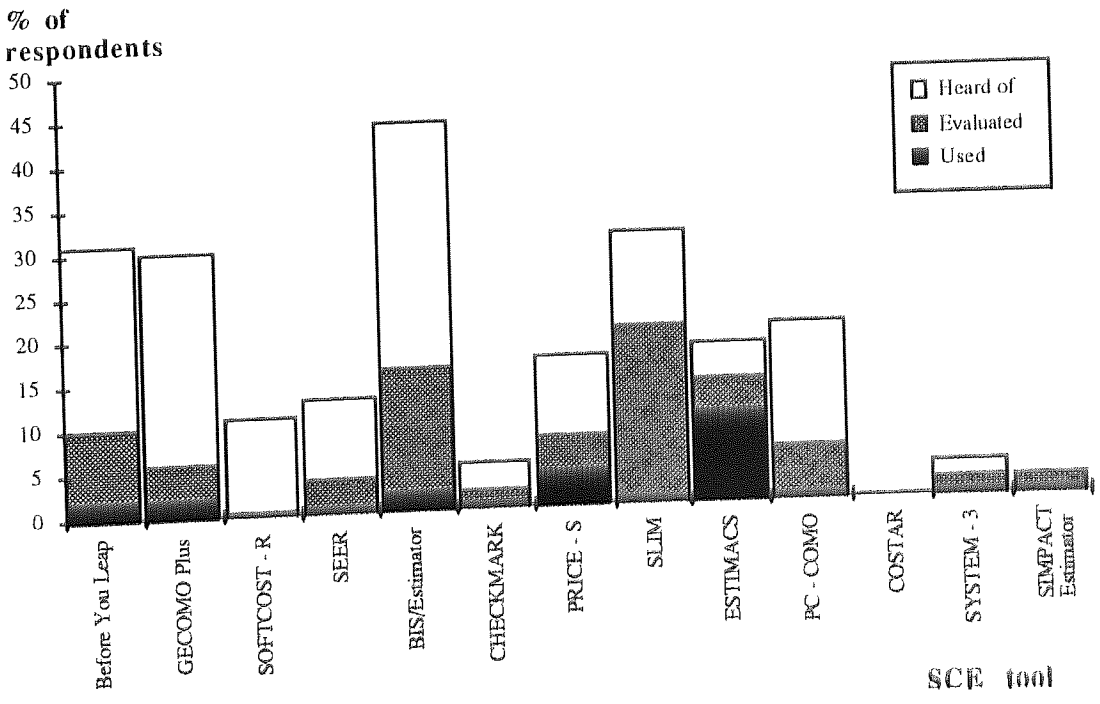


This result, however, may also suggest that estimating tools are not commonly used because they approach the problem from the wrong direction. Specifically, if a manager is already developing a plan, how far would a specialist estimating tool help in the planning process? The point of SCE tools is that they provide some of the numbers to apply to the plan. One could speculate, however, that what is difficult

about estimating is not estimating the actual cost (since many estimates are likely to be estimates-to-win or merely budgets which the project is expected to meet regardless) but having a means of justifying the development of the project itself. The speculation continues that even if an estimate is demanded of a manager before a detailed plan has been created, it is likely that at least a broad mental plan of the important tasks will be made, and - more significantly - it is on the basis of this plan that the estimate is both meaningful to the manager and justifiable to a third party (such as senior or user management).

In other words, it is possible that in the mind of a manager an estimate is derived from the plan and not the other way round. Once the plan has been created a more meaningful basis for estimation exists, and therefore the estimate from a SCE tool is either no more than a check or entirely redundant. Of course, a budget may be thought of as a "received" estimate which drives the plan, and it is quite likely that a number of software development managers feel they work within "unreasonable" constraints. This point is more a question of how development teams can produce software under such conditions rather than a question of whether the estimate is an accurate one.

Figure 6.5 : Relative "heard of", "evaluated", or "used" for 13 commercial tools



6.3.4 *Proposition 4*

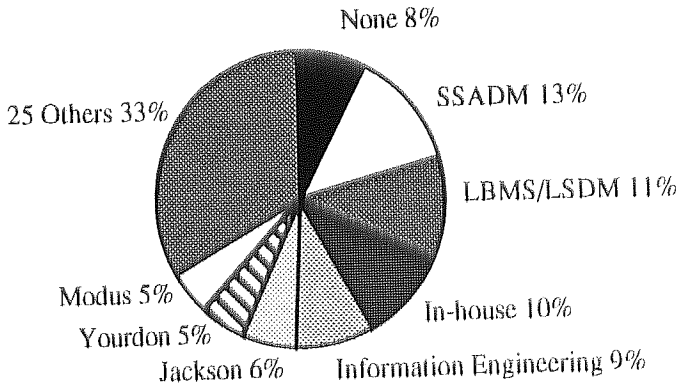
Proposition 4 stated that: There is a failure of publicity, such that tools which could be used by development managers are unknown to them. The evidence suggests that this statement is true. Respondents were asked whether they had heard of, evaluated or used one of 13 named commercial Software Cost Estimating tools. The list of 13 tools was taken from Fisher & Gorman (1990). Although COSTAR, SYSTEM-3 and SIMPACT ESTIMATOR were reported by Fisher & Gorman to be unavailable in the UK, they were included in case the situation had changed, and because the target sample may be equally familiar with U.S. tools. As it transpires (see Figure 6.5), SYSTEM-3 had been evaluated by one company, although as a whole these three tools were significantly less well-known.

The respondents were asked: "Have you heard of any of the following Software Cost Estimating tools?"; if yes, "Have you evaluated this tool?"; if yes, "Do you use this tool?". At first sight, there appears to be a reasonable level of awareness of the tools (61% having heard of at least one tool), although evaluation is much less common (39%), while only five of the tools were found to be in use. (Note that the analysis here assumes that "evaluated" implies "heard of", and "used" implies "evaluated".) Those in use were ESTIMACS (6 users), PRICE-S (2 users), Before You Leap (1 user), GECOMO PLUS (1 user) and BIS/ESTIMATOR (1 user). One company used two commercial tools, while two commercial tools not listed were found to be in use: PMS BRIDGE (2 users) and SYZYGY (1 user). Even though 61% claimed to have heard of at least one of the listed tools, the figures still suggest a failure of publicity for the following reasons:

- The respondents to the survey were exactly the sort of person at whom any publicity for SCE tools should be directed, and so any successful publicity campaign should result in all of the managers having heard of at least one tool.
  Target awareness = 100%
- In fact, 39% of this group had not heard of any of the listed commercial SCE tools.
  Apparent awareness ≤ 61%
- Those who had heard of a tool but carried out no evaluation (22%) probably have little appreciation of what the tools actually do.
  Apparent awareness probably ≤ 39%
- 56% of the positive "heard of" data was provided by just 19% of the respondents, so the majority of the data came from a small sub-group.
  Apparent awareness < 39%

Thus, the evidence suggests that at most 39%, probably fewer, could be deemed to fully appreciate estimating technology. It therefore seems reasonable to conclude that the initial figure of 61% was misleading and SCE tools have, in fact, suffered from poor publicity.

Figure 6.6 : The relative popularity of methodologies in use (out of 92 responses)



6.3.5 *Proposition 5*

Proposition 5 states that: There is no framework in place from which to calibrate a commercial package or develop an in-house model, so the technology is unusable. The evidence suggests that this statement is false. The framework proposed the need for:

- identifiable stages of development against which the model can be applied, either by using a structured methodology, or by recording time expenditure against different types of activities (e.g. meetings, interviews, programming, etc.);
- the collection of project time data, with time being the basis of many estimating models (thus allowing the development of an in-house model or the calibration of a commercial tool);
- the use of some form of project management tool, thus implying tools are typically seen as "acceptable" in supporting the creation of a work plan.

When asked "Which structured (documented) methodologies are used within the department?", 86% indicated at least one method was in use, 17% use an internal method, while 14% use no method. (Two respondents failed to answer so the

percentages given here are out of 52.) Nearly half of these 52 respondents (44%) use more than one method, with a total of 92 responses being recorded (see Figure 6.6 for a summary).

When asked how often person-hours were collected during the development of a project the choice being "Always", "Mostly", "Occasionally", "Rarely" and "Never", 70% replied they always collected person-hours, 11% mostly, 9% occasionally, 6% rarely and 4% never. Assuming "Mostly" is taken to mean there would be few occasions when person-hours would not be recorded, then 81% collect time data at a high rate. When asked the same type of question for time expenditure, two further companies recorded time expenditure at a high rate although they did not indicate using a structured methodology. This allows 87% to apply a model to identifiable stages of development. Together with the use of PM tools, the three conditions for the calibration and development of basic in-house and commercial tools are satisfied by 78% of the sample. But, if 78% of companies could be using estimating tools, it is still unexplained why only 30% actually do so.

Table 6.2 : Comparison of UK surveys

	This survey (1991) %	IPL (1989) %
Identifiable stages of development	87	60*
Standard collection of time data	81	63
Use of Project Management tools	94	77
Use of any type of estimating tool	30	23

* only the percentage using a method available

The importance of this "SCE" framework can be further supported by comparing this survey with the IPL survey results (see Table 6.2). Since both surveys were carried out in the UK, one could speculate that the underlying reasons which restrict the use of estimating tools are the same. In such a case, one could suggest that there is a fixed ratio which relates the proportion of companies who satisfy the conditions of the framework and the proportion of companies using estimating tools. Thus:

$$SCE_{users} = NUM_{framework} * R$$

where,

SCE_{users} = % using in-house or commercial tools

$NUM_{framework}$ = % satisfying all three conditions of the framework

R = constant ratio for UK companies

Using the data from this survey, R=0.385 (since SCE=30 and NUM=78). Although NUM is not available for the IPL survey, it cannot be higher than the lowest percentage of the three-point framework (i.e. 60% using a methodology, see again Table 6.2). It is also unlikely to be substantially lower than 60, since the three conditions of the framework would tend to be found together. Taking SCE=23 and NUM=60 would give R=0.383, so that the two values of R are in remarkably close agreement. Although such extreme accuracy is no doubt fortuitous, it seems reasonable to conclude that, once the framework has been accounted for, it appears that the remaining properties which enable companies to become SCE users have remained constant between 1989 and 1991. The factors which provoke potential users into becoming actual users is still unclear, however.

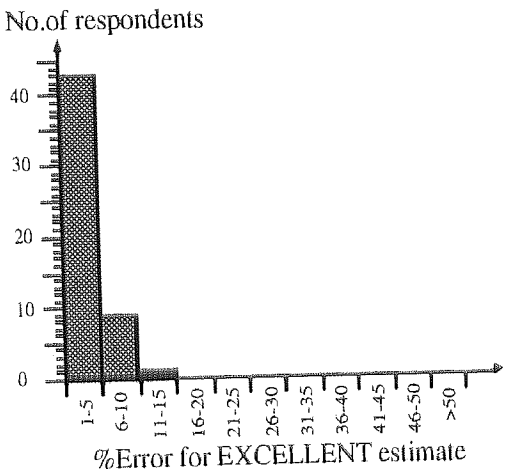
6.3.6 *Proposition 6*

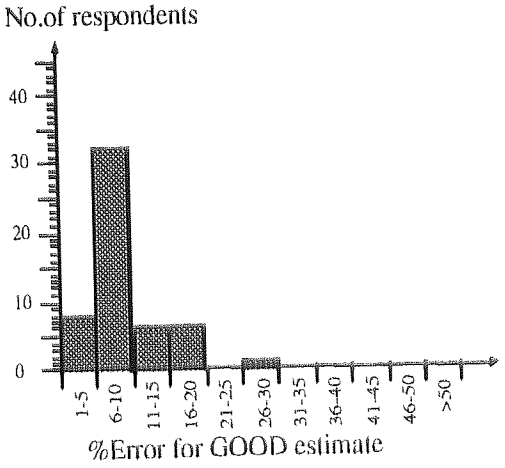
Proposition 6 states that: Where needed, a model is developed which is directly relevant to the company concerned and not purchased from outside. This proposition follows the recommendation of Kitchenham (1992). The evidence suggests, however, that this statement is false. Only 13% of the sample developed in-house models; this is less than the use of commercially-developed tools (24%). Although in-house models would be more sensitive to local conditions, budget constraints and lack of knowledge of metric models may explain this low level of in-house development.

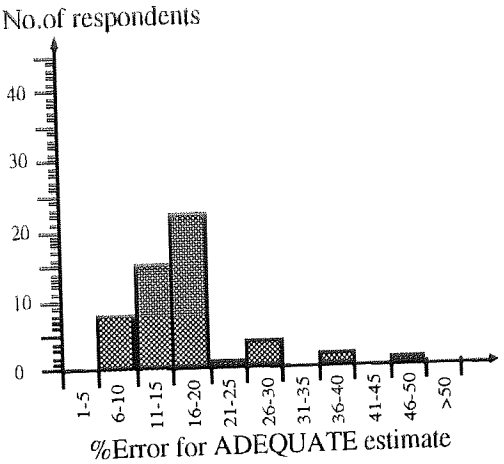
6.3.7 *Proposition 7*

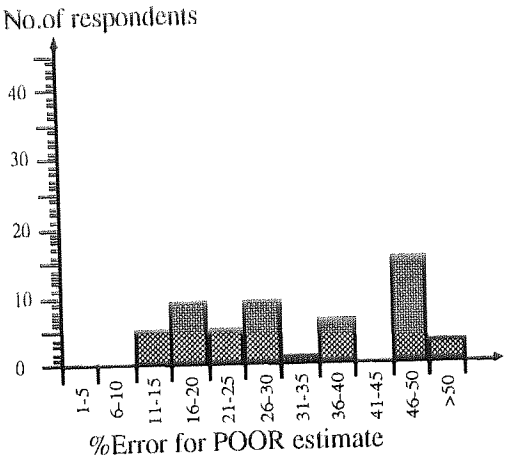
Proposition 7 states that: Estimating tools have been used but found to be inadequate, so the technology is not perceived as being useful. The evidence does not clearly deny or agree with this statement but the suggestion is that proposition 7 is probably false. None of the respondents indicated that they had stopped using a SCE tool. However, 9% of companies had evaluated at least one tool without becoming actual users. It was not possible to determine why, since this was not a question in the survey. However, one could speculate that there are many possible reasons including cost, ease-of-use and levels of support as well as inadequate functionality.

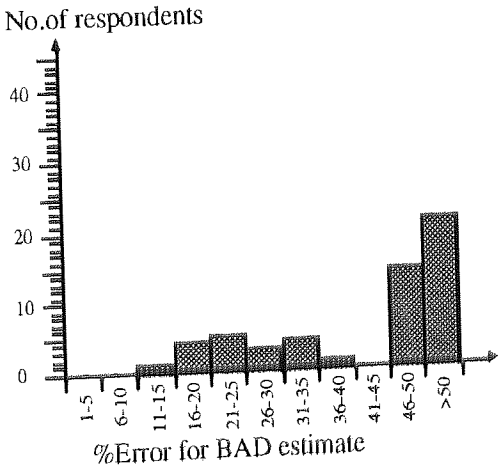
Figure 6.7 : Level of error associated with excellent/good/ adequate/poor/bad estimates











Summary Table

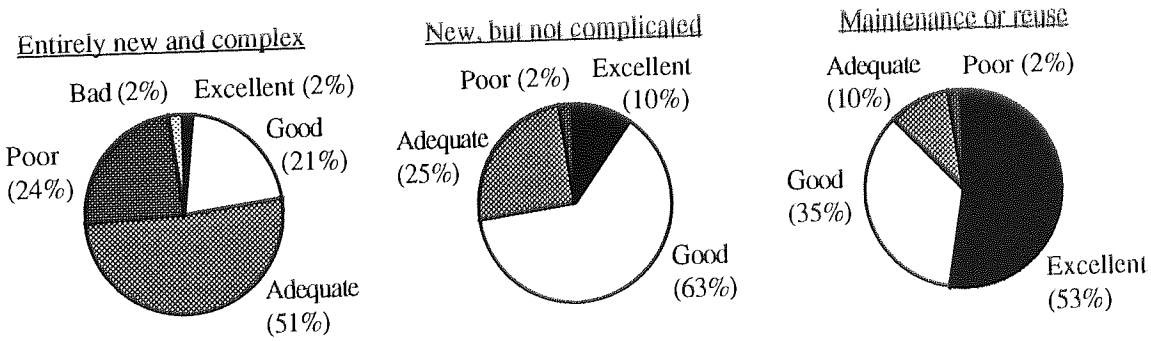
Estimate	Av. % Error
Excellent	5.4
Good	11.2
Adequate	18.8
Poor	36.1
Bad	58.8

Taking "accuracy of estimate" to be one of the key properties of an adequate SCE tool, however, a further question was included to investigate what level of accuracy managers expected. Firstly, the survey asked respondents what percentage error (±%) they would assign to estimates which they regard as being "excellent", "good", "adequate", "poor" and "bad". The results are given in Figure 6.7. If "adequate"

represents the level at which an estimator becomes useful, then most tools purport to be able to give results within this range, roughly ±20% or better. For instance, ±20% is exactly the level of accuracy Boehm (1981) stated as being a reasonable one for COCOMO.

However, the level of accuracy must surely change according to the nature of the project. For instance, Boehm changed the parameters of COCOMO according to whether the project was organic (straight-forward), semi-detached (between straight-forward and difficult) or embedded (difficult). Although not looking to match Boehm's ideas exactly, embedded, semi-detached and organic map onto the categories used here. The respondents were asked:"What accuracy of estimate for project costs and/or duration would be expected from a manager if the project was: entirely new and complex; new, but not complicated; maintenance or reuse of system code." A multiple choice of five answers was given for each type of project from "excellent" to "bad", as above.

Figure 6.8 : Expectations of the levels of accuracy for different kinds of system



The expectation was that the easier the project ("maintenance or reuse" being the easiest), the more accurate the expected estimate. This was found to be the case (see Figure 6.8). It should be noted, however, that a level of accuracy of "adequate" or better was expected by 74% of managers even when dealing with the development of entirely new and complex systems. It seems reasonable to suggest, therefore, that regardless of the nature of the project estimates are still expected to be within (at most) ±20%. If this is indeed the threshold against which estimates are judged (whether generated by humans or tools), then other studies have indicated that both are operating at a considerably worse level than the performance threshold of "adequate".

Kusters *et al* (1991) compared the performance of 14 project leaders against the data from a completed project. The subjects made initial estimates manually of the effort and lead-time given a description of a project. The subjects then made further estimates using Before You Leap and ESTIMACS. (Kusters *et al* report that the tools were not calibrated, which means the tools would lose some accuracy.) A final estimate was recorded once the subjects had experience of using the tools. While the actual effort was 8 months and the lead-time 6 months, the mean effort estimate ranged from 27.7 to 48.5 months, while the mean lead-time estimate ranged from 8.5 to 12.1 months. These mean values suggest that no-one in that study was able to estimate, either manually or using a tool, in the range expected by the respondents of this survey.

Table 6.3 : A summary of results

Proposition	Conclusion
1. Estimating is not regarded as a problem.	False
2. Estimating tools are in common use.	False
3. Some other tool is being used in place of SCE tools.	True
4. There is a failure of publicity.	True
5. There is no framework in place for their use.	False
6. An in-house model is more likely to be developed.	False
7. SCE tools have been found to be inadequate.	Probably false

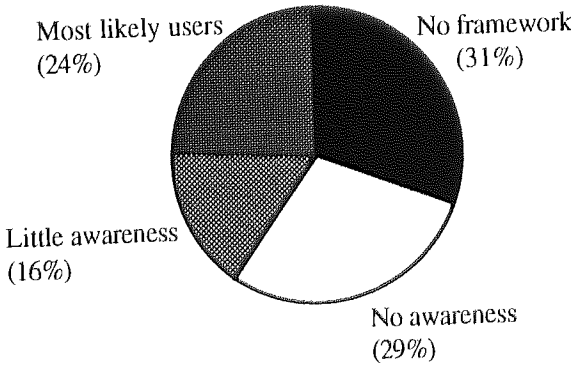
6.4 Survey conclusions

The propositions investigated here (and summarised in Table 6.3) were meant to provide the data by which 'Option Zero' could be dismissed. The empirical evidence required was that: a) project managers continue to see estimation as a problem; b) there is no prohibition to the use of SCE tools, and; c) no other tools are being used in their place. If these three propositions were true then there seemed no reason why some form of useful SCE tool could not be developed for KBSs. Even though estimation is regarded as a problem by almost all of the respondents, the use of commercial tools or the development of in-house estimating models remains low.

This result is in spite of the fact that over three-quarters of companies fulfil basic requirements for the development, calibration and use of these tools. The relevance of the three-point framework of use, set out at the beginning of this chapter and investigated in Proposition 5 (above), is supported by the fact that all 16 companies which used a SCE tool satisfied the requirements. Although this demonstrates the importance of the framework it is clear that other forces are at work, such as a lack of publicity. Even though almost two-thirds had heard of at least one commercial tool, most of the positive responses came from a fifth of the sample who had carried out a study and subsequently use one of the commercial tools. It was found that the 38 non-users could be divided into four categories (see Figure 6.9):

- no framework in place to support the use of estimating tools;
- no awareness of the technology (have not heard of any estimating tools);
- little awareness of the technology (have heard of no more than two estimating tools, no evaluation carried out);
- most likely users (good awareness of the technology and/or have evaluated at least one estimating tool, but none in use).

Figure 6.9 : Categorising the SCE non-users



No framework, or little or no awareness of the technology covered three-quarters of the non-users. The survey did not reveal why the last group of well-qualified potential users failed to become actual users. Since almost all respondents use a Project Management tool, it seems clear that tools to support project management are seen as acceptable (if not essential). This result, however, may also hint at a more fundamental problem for estimating tools. Since the general use of project management tools is found to be so widespread, it may be that in the mind of software development managers estimating is nothing more than planning the tasks which need to be carried out.

The suggestion here is that once a plan has been created, estimating time and budgets is an implicit number-assigning task, whereby the total effort and cost of the project is the sum of effort and cost numbers assigned to each task. If this hypothesis is true, then estimating tools currently approach the problem from the wrong direction: estimates do not help planning; rather, planning is the sole source of estimates. If this hypothesis is false, then there seems to be no clear reason why estimating tools fail to be in greater use. So, while the need for and feasibility of using SCE tools has been established there is doubt over whether SCE or project management tools are the best means of providing this support. This will be the subject of the next chapter.

7. Planning or Estimating Tools?

"No matter what you answer, executives will halve it, users will doubt it, and no one will stake a day's pay on it."
Definition of an estimate used at the beginning of an advert (inside cover, *Journal of Systems Management*, 1(12), 1992)

Summary: *Given that it is now known that project managers could but do not seem to be using Software Cost Estimating tools, the next question to be answered is "Why not?" A follow-up to the original survey is described in which a conflict between the use of estimating and planning tools is identified and resolved. It will be argued that by redefining SCE tools as those which can generate estimates before a plan (at a bidding/tendering stage) and supports the creation of plan-based (task-based) estimates, estimating tools do indeed have a necessary place in the armoury of current project managers.*

The third proposition which needs to be established in order to prove that 'Option Zero' is false is that SCE tools are necessary because no other tools can take their place in generating estimates. The high use of project management tools and the low use of SCE tools might suggest that this third proposition is false: project management tools are being used instead. The question to be answered by this chapter is how project management tools are being used and whether this use rules out the need for SCE tools. If it does, then all research carried out into SCE tools has been misguided; if not, then there needs to be an explanation of why those companies which see estimating as a problem and have good knowledge of SCE tools do not use them.

Although accuracy is clearly an issue 'Option Zero' suggests that until it has been established that SCE tools have a rôle to play in project management then accuracy is a false problem. If it can be found that SCE tools have been developed incorrectly then it would not be surprising to find they are inaccurate and any accuracy could be put down to small samples deducing narrow models tied to a particular development environment. If it can be found that project managers are using SCE tools incorrectly then the method described in Chapter 5 of developing and using such models and tools can be defended. The rest of this chapter will therefore set out to describe:

- a definition of what makes SCE users different from non-users;
- the potential for conflict between planning (project management) and SCE tools;
- a resolution of the problem by studying the early estimating and planning stages of project management (the EEPS model).

7.1 Distinguishing between SCE users and non-users

If most companies surveyed in the previous chapter do not use Software Cost Estimating tools, what is it about those companies that do use them which makes them different from the rest of the survey sample? The follow-up gathered data from three principal sources:

1. The original survey (Chapter 6);
2. Face-to-face interview at the company's own site;
3. A telephone poll.

It should be remembered that all companies in the follow-up comply with the framework for using SCE tools defined in Chapter 6 (since they all use a method, collect time data at a high rate and use a project management tool). For the purposes of this comparison, the 16 companies which use SCE tools will be referred to as Group 1, while the 9 companies which appear well-qualified but remain non-users will be referred to as Group 2.

7.1.1 *The original survey*

The data provided by the companies in the original survey showed that Group 1 companies tend to:

- be over-represented by the Financial sector, while Computing companies are over-represented in Group 2 (see Table 7.1);
- have larger departments than Group 2 companies. This division seems to be at around 150 personnel (see Table 7.2);
- be more susceptible to changes in user requirements than Group 2 companies (see Table 7.3).

If large departments tend to be working on large projects, then overruns would be particularly costly and more likely where requirements tend to be volatile. Under these conditions, the use of SCE tools is understandable. The importance of being a Financial company is not clear at this point.

Table 7.1 : Contingency table based on Business Area

	Financial	Computing	Other
Group 1	6	5	5
Group 2	1	6	2

Table 7.2 : Contingency table based on Department Size

	<150 people	≥150 people
Group 1	6	10
Group 2*	7	1

*One Group 2 failed to answer

Table 7.3 : Contingency table based on Requirements Volatility

	≤Occasional	>Occasional
Group 1	1	15
Group 2	5	4

7.1.2 *Face-to-face interviews*

In a bid to uncover further differences, all Group 1 and 2 companies were contacted in order to carry out personal face-to-face interviews. However, only six of Group 1 and none of the Group 2 companies agreed to take part. On the basis of the six companies which did take part the following points emerged:

- One of the companies had such a change in corporate culture that the in-house SCE tool and methods tool IEW were no longer in use. More importantly, clients were no longer charged for development time. This may suggest that only the clear need for an accurate estimate will dictate the use of a SCE tool.
- Four of the five remaining Group 1 companies interviewed suggested that the SCE tool was used for comparison rather than as the main method of estimating. As such, it is not clear whether estimating tools are being used in

the belief that they are accurate, or as part of a need to show senior management or a client that all attempts are being made to produce an accurate estimate.

Both these points suggest that even where needed, managers might be quite happy to use their own method of estimation rather than a SCE tool if it were not for extra (outside) pressure on how that estimate is derived. The influence of the client on the way project managers estimate will be the subject of further investigation later in this Chapter.

7.1.3 *A telephone poll*

In order to answer the above questions and overcome the low response rate of the personal interviews, it was decided to conduct a telephone poll of all Group 1 and 2 companies. The poll assumed that the need for an accurate estimate would be based on a client being charged and/or the size of project being developed. The first point was tested by asking:"Do you charge a user (department or company) for the time spent on software development?" On the evidence of the interviewed companies, Group 1 companies should answer "Yes". The second point was tested by asking:"What is the size of a typical project (in person-months)?" Since overruns on larger projects constitute more of a "threat" to resources, Group 1 companies might be expected to develop larger projects than Group 2. The telephone poll obtained responses from 10 of the SCE users and 7 of the non-users. The results showed that Group 1 are more likely to:

- charge for development time than Group 2 companies (see Table 7.4), although some in Group 1 do not charge, and;
- be working on projects of 24 person-months (or more) than are Group 2 companies (see Table 7.5), although some in Group 1 typically develop very small projects.

Table 7.4 : Contingency table based on Charging a Client

	Yes	No
Group 1	8	2
Group 2	4	3

Table 7.5 : Contingency table based on Person-Months Effort

	<24 person-months	≥24 person-months
Group 1	3	7
Group 2	5	2

These results suggest that factors likely to increase the need for an accurate estimate are positively associated with the use of SCE tools. However, the fact that some Group 1 companies do not charge and develop small projects suggests a critical need for accuracy is not a sufficient reason for using SCE tools.

7.1.4 *A rule to characterise SCE tool users*

Combining the results of the original survey and telephone poll, it is possible to identify five separate factors which increase the likelihood that a company capable of using SCE tools will actually do so. These factors are:

- being a Financial company;
- having a large department (≥150 people);
- experiencing high requirements volatility ('mostly' or 'always');
- charging a client (department or company) for development;
- developing large projects (≥24 person-months effort).

In order to judge both the relative importance of these five factors and their completeness, a table was constructed (see Table 7.6). In terms of completeness, it can be noted that the two companies to which all five factors apply (User 1 and User 5) are both in Group 1, while the company to which none of these factors apply (Non-user 5) is in Group 2. Although based on only three observations, this does give some measure of belief that all the relevant factors leading potential users to become actual users have been identified.

Between the two extremes, where some factors are present and others absent, the picture is less clear-cut. An additional complication is that there appear to be two anomalies in the sample:

- User 8, the only group 1 company which does not have high requirements volatility. This company is also the only Financial company which does not have high requirements volatility. However, there seems no meaningful explanation for the significance of such an interaction of factors. In this case, the combination of large department size, charging and being a Financial company clearly outweighs the low requirements volatility. User 8 is omitted from the analysis, therefore.
- User 7 and Non-user 1 have exactly the same pattern of answers - high requirements volatility but none of the other factors. There seems no way round this anomaly except to conclude that one of the observations is somehow

Table 7.6 : A summary of responses

Company code	Factor				
	1	2	3	4	5
User 1	Y	Y	Y	Y	Y
User 2	N	N	Y	Y	N
User 3	N	Y	Y	Y	Y
User 4	Y	N	Y	N	Y
User 5	Y	Y	Y	Y	Y
User 6	N	Y	Y	Y	Y
User 7	N	N	Y	N	N
User 8	Y	Y	N	Y	N
User 9	N	N	Y	Y	Y
User 10	N	Y	Y	Y	Y
Non-user 1	N	N	Y	N	N
Non-user 2	N	N	N	Y	N
Non-user 3	N	Y	N	Y	Y
Non-user 4	Y	N	Y	N	N
Non-user 5	N	N	N	N	N
Non-user 6	N	N	N	Y	N
Non-user 7	N	N	N	Y	Y

KEY: 1 - Being a Financial company
2 - Having a large department (>150 personnel)
3 - High requirements volatility ("mostly" or "always")
4 - Charging a client for development time
5 - Working on large projects (>24 person-months effort)
Y - The company has this factor
N - The company does not have this factor

"wrong". The decision taken is that User 7 is "wrong", since the company typically undertakes very small projects indeed (a few person-weeks), whereas commercial SCE tools are typically aimed at projects of several person-years. For this reason, User 7 is omitted from the analysis.

More pragmatically, regarding Non-user 1 as the "correct" observation makes it possible to produce a simple and meaningful rule which correctly classifies the remaining 15 cases. The rule is that a potential SCE tool user company will become an actual user if:

either it has high requirements volatility and tackles large projects;
or it has high requirements volatility and charges clients.

7.1.5 *Discussion of results*

The rule produced in §7.1.4 is a characterisation rather than a definitive analysis of SCE tool users since only a small sample was available for study, and of these, two data points had to be removed before the rule could be derived. Even given these methodological problems, however, it does seem possible to reject the assertion that if estimation is perceived as a problem those companies which could use SCE tools will do so. This result strengthens the possibility that 'Option Zero' is, in fact, true. For those companies which do use SCE tools the most dominant factor seems to be the volatility of requirements specifications: the likelihood that requirements will be changed during the life-time of a project. What meaning can be attached to this result?

Intuitively, if the cost of a system is directly related to the functionality defined by the requirements specification, then there is an additional overhead involved in developing a system where this functionality is poorly defined or continually changed. As the requirements change, work which has already been completed may become redundant, while the introduction of new functionality means another costly cycle through the analysis, design and implementation phases of the development process. In this scenario, a SCE tool may be seen as providing the means of applying a consistent method of estimation to such a troublesome and chaotic development environment.

However, while dealing with high requirements volatility may appear to be a good reason for using SCE tools it should be borne in mind that no current SCE model or tool has adequately represented the effect of this problem. For instance, although

COCOMO provides a cost-driver which allows for the volatility of requirements, RVOL, the estimate itself is still crucially based on an understanding of what the system is meant to do. The question here is what to do when changes are made to user requirements and system functionality.

Currently, the only action a project manager could take is to re-generate an estimate based on the new functionality and compare the old estimate with the new result. But it is doubtful whether the difference, if any, would have any meaning. Specifically, one must presume that - in the same way that adding personnel non-linearly affects the complexity and effort of a project - continually redefining the functionality of a system will have the same effect. Rather than RVOL as a cost-driver, then, this non-linear effect would produce a model of the form:

$$\text{Effort} = a.(\text{Size})^{b+c} * \text{Cost drivers}$$

where,

a = first parameter
b = exponential parameter
c = RVOL adjustment

Of course, if a company typically experiences these problems then any SCE tool which has been properly calibrated will have adjusted to these effects. This is the same as setting the RVOL cost-driver to "extra high" and misses the point that there is no explicit means of relating an initial to a later estimate. How can a project manager understand the effect of such volatility when all that current SCE tools can report is what the (new) functionality would have cost had it been the original functionality? COCOMO could assume that the changed system is actually a new system which will reuse components from a previous system. But what adjustment should then be made for the fact that some work is only partially complete, or even scrapped? Since the effect of changing requirements is not a central part of any of the underlying SCE models, the initial conclusion must therefore be that SCE users in this study do not have a proper reason for using current SCE tools. So what other tool might they be using?

7.2 A potential conflict between planning and SCE tools

Although it appears possible to characterise those companies most likely to use SCE tools, the small sample and the fact that two of the data points needed to be removed before a definitive statement could be made also suggests that there may be other forces acting on the

use or non-use of SCE tools. The point to be addressed here is that there is a conflict between the use of SCE tools and the use of some other tool to fulfil the same (estimating) function. Notably, it was found in Chapter 6 that 94% of project managers now use a project management tool. The same survey also asked for the uses of the tool, and it was found that:

- 100% are used for planning;
- 86% are used for monitoring the project;
- 77% are used for reporting purposes;
- 26% are used for charging/cost allocation.

The figure of 100% is regarded as sufficient justification for regarding "planning tools" and "project management tools" as synonymous in the rest of this thesis although it is accepted that there is more to project management than simply planning. It is also interesting to note that of the 44 respondents (81%) who agreed that such tools are now in common use, 43 said this became true from about 1983 onwards. This figure would seem to accord well with the period in which cheap, powerful desk-top PCs became available, and so project management techniques could at last be made easier to use and revise on a day-to-day basis.

The use of a planning tool to estimate assumes that as the planning process identifies the relevant tasks, estimates of time, cost and performance can be assigned and the total summed for the project as a whole. This is called bottom-up estimating. However, while current planning tools allow estimates of task duration and cost to be recorded and aggregated, they provide no support for making the relevant estimates. Do commercial SCE tools clearly have this functionality?

7.2.1 *The functionality of commercial SCE tools*

Commercial (rather than in-house) SCE tools are those which project managers in the Chapter 6 survey were most familiar with. The way the functionality of an estimating tool is portrayed was investigated by contacting the companies which sell them and asking questions about the types of estimates produced and the development stages at which the tool can be used. Of 19 known tools, 16 addresses were found and 11 were contacted by fax and the other 5 by post. The survey was addressed to "The Sales Desk". Ten replies were received and the results are summarised in Table 7.7. As can be seen, the tools are designed to estimate a range of properties with all the tools giving estimates of manpower. All except BIS/Estimator give estimates for cost. All but ESTIMACS give estimates from Requirements onwards. All but SEER can estimate without plan-based information being available, while estimates in terms

of a Work Breakdown Structure (WBS) are possible for all the tools except Before You Leap, ESTIMACS, SEER and SPQR.

Table 7.7 : Estimating tool vendors survey

	Before You Leap	BIS/Estimator	Checkmark	Costar	Estimacs	SEER	SLIM	Softcost-R	SPQR	Price-S
Estimates provided										
Cost	X		X	X	X	X	X	X	X	X
Duration	X		X	X	X	X	X	X	X	X
Manpower	X	X	X	X	X	X	X	X	X	X
Lines of code			X			X		X	X	X
Error rate			X				X		X	X
Stage usable										
Requirements	X	X	X	X		X	X	X	X	X
Specification	X	X	X	X	X	X	X	X	X	X
Design	X	X	X	X		X	X	X	X	X
Implementation	X	X	X			X			X	X
Testing		X	X			X			X	X
Maintenance	X	X	X	X	X	X	X		X	X
Before a plan	X	X	X	X	X		X	X	X	X
For a WBS		X	X	X			X	X		X

It should be noted that the information in Table 7.7 is how the company selling the tool portrays its functionality. According to this information, the tools listed are clearly designed to be usable at the earliest development stage, and before a project plan has been produced. If project managers recognise the need for accurate estimates then the vendors at least suggest their tools can provide this much needed functionality at the earliest possible stages of a project. If there is no problem over the use of SCE tools early in the life-cycle, what other problems would make their use difficult?

7.2.2 *The TAD-law of data input*

The availability of data upon which to make an estimate is crucial to the viability of using a particular tool. Specifically, that planning tools could be used as estimating tools does not prove that they are used in this way. The greatest difference between SCE and planning tools is the type of data which is used to model (implicitly or explicitly) the cost of a project. While planning tools provide the framework within which to construct a model of the project - in terms of the planned tasks - estimating tools impose a model of the development process and require instead information on the peculiar complexities and constraints of the project being estimated. This is one reason why calibration within and between development environments is so important.

The key to the usefulness of any project management tool is whether the data required as input is available at a time when the project manager requires the output. Behrendt *et al* (1991) noted this problem with many of the "hindsight" structural metrics (see again §4.2.2). Although this rule (called here the 'Temporal Availability of Data' or TAD law) may appear obvious, it is not always clear that estimating tools abide by it. In particular, a number of estimating tools require an estimate of lines of code (LOC) as input when making an estimate of person-months effort and/or calendar months project duration. If the value of LOC is taken to be an estimate of the actual size of the system to be developed, then LOC can be described as TAD-late, since no reasonable estimate can be made of the actual system size until well into the project.

On the other hand, if LOC is taken to be a comparison between properties of the current project and those of past projects (in other words, estimating-by-analogy), then LOC becomes TAD-early since the final LOC size of a past project can always be made available at the beginning of a subsequent project. The goal of estimating tools, therefore, should be to ensure that all input data are TAD-early when the estimate is produced. Note that the distinction between TAD-early and TAD-late data is always relative to the point at which the estimate is produced. Unfortunately for estimating tools, it is not always clear whether the input data comes from the current or previous projects or exactly what measurements are needed to calculate their values. So what types of data are available within a development project?

7.2.3 *A characterisation of project data*

A certain amount of this confusion over the type of data input required can be explained by the fact that a measurement of software development can be a measurement along a number of dimensions. A measurement can be:

EITHER *of the development process* (such as time schedules, team size, etc.);
OR *of the development products* (such as code, documentation, etc.).
EITHER *direct* (where the metric is a count of an attribute of the "element" or "object" being measured, e.g., counting the number of boxes on a data-flow diagram);
OR *indirect* (where an attribute of one object stands as a measurement of another, e.g., the complexity of a program's structure measured as the number of control statements).

To these two "What?" dimensions, a further one can now be added which relates to the "When?" of taking a measurement. It is:

EITHER *TAD-early* (where data first becomes available no later than the point at which the estimate is produced, e.g., number of tasks planned counted during or after the planning stage);
OR *TAD-late* (where data first becomes available only later in the development process, e.g., system size or productivity rates at the outset of a project).

TAD-early estimates are based on the information available at the point at which the estimate is made. This would include measurements of the products produced to-date on the current project, plus all the historical data collected on past projects. Thus, at the design stage when data-flow models, E-R diagrams, requirements, specification and design documents have already been produced, SCE models which took as input measurements of any of these products would be classified as TAD-early direct measurements of the products. If the SCE model modelled the risk involved in completing the project on the basis of these measurements, the output would be classified as TAD-early (because of the data used) indirect measurements of the process. Note, however, that the "risk" is the risk at that point in the project, not the risk as it would be calculated at the end of the project. The later is a TAD-late estimate.

TAD-late estimates are based on information which would be measurable (directly or indirectly) only later in the development process. In this case, TAD-late estimates are based on surrogates for the missing information. If the SCE model modelled the delivery date of the system as a whole at (again) the design stage, the same information would be available for the TAD-early estimates, but the output would be TAD-late because the delivery date is clearly a point in the future which is as yet unattained. The same "unattained" description can be attached to many other types of estimates produced by SCE models, including person-hours effort and calender-months duration.

Note, however, that if the input to the SCE model was based on sizing the current stage of the project, the information available would be TAD-early and the output would also be TAD-early (since the actual effort and duration are measurable). Measuring a stage of the development process when it has been completed will always be TAD-early input and output. Measurements of expended effort and duration would be useless as estimates, but very useful in judging the current performance of the development team against estimates values. These dimensions of software data are illustrated in Table 7.8 (although the examples given are not meant to be exhaustive). The examples of TAD class data are as at step 2 of the EEPS model, i.e., when the outline requirements are the only information available for the current project.

Table 7.8 : Characterising project management data

		Process	**Product**
TAD-early	**Direct**	planned tasks available staff	no.of requirements pages of reqmts doc.
	Indirect	risk project size	requirements difficulty design complexity
TAD-late	**Direct**	delivery date person-hours	no.of reported bugs lines of code
	Indirect	person-hours effort productivity	system size code complexity

As can be seen, planning tools can be fitted neatly into Table 7.8 by defining their use as modelling direct TAD-late properties of the process using direct TAD-early

information of the tasks and experience of the assigned personnel. Once completed, resources for the project could then be calculated by measuring the likely project size and risk. Planning tools, therefore, are very much process-oriented and abide by the TAD law. They are particularly useful at answering the sort of questions that a client would ask, such as:"How much will it cost?", and "How long will it take?"

A plan, however, is no more than an approximation of the project and two managers could very well disagree on the details. Only when reliable (and accurate) estimates of development effort and productivity are available can any confidence be placed in the plan itself. Planning, therefore, is still reliant on indirect TAD-late information such as system size and rates of productivity.

Estimating tools, on the other hand, have been developed to address virtually every box in Table 7.8: Direct/indirect measures of the process are parameters typically used to define adjustment factors, and when related to lines of code give the COCOMO model (Boehm, 1981); code complexity related to the number of reported bugs gives the cyclomatic complexity model (McCabe, 1976); delivery date related to effort gives the Norden-Rayleigh manpower model (Putnam, 1978); functionality related to system size gives the Mk.I function point model (Albrecht & Gaffney, 1983); while the importance of documentation is seen as adding to the later Mk.II version of function points (Symons, 1988, 1991). This apparent flexibility, however, is also the fog that cloaks the potential usefulness of estimating tools because the logical connection between the process/product attributes being estimated and the nature of the data used as input becomes unclear - especially regarding whether the data is meant to come from the current or previous projects.

Given that SCE tools should ideally fit into current working practices, one could suggest that estimating tools should begin like planning tools with measures of the development process and work from top-left to bottom-right in Table 7.8. In this way estimates of process effort and product size should (and only ever) begin with details of the method and personnel being used and adjusted using attributes of whichever products have been produced at that point (requirements documents, etc.). Used in this way, a (task-based) estimating tool would operate in exactly the way as estimating with a planning tool, except using a task-based estimating tool would:

- provide a sound basis for incorporating the previous indirect system size and productivity rate information (required, but not provided by, planning tools), since estimating would now be structured by a well-defined model;

- provide estimates of some of the important product measures (such as the number of likely bugs in a system of a certain size which would be used to size the unit test and system testing phases), since the estimating tool is derived from a wealth of historical data;
- make use of the same TAD-early requirements data (that ultimately drives the planning task itself), since this is exactly the point at which estimating tools would be most useful and planning tools are of no help.

It is these points which are crucial when it comes to deciding which tools can best support the difficult and error-prone task of assigning the right level of staffing and money to a project. The question that remains, however, is: Can this distinction between planning and SCE tools be identified in the way project managers currently estimate? The next section will argue that it can.

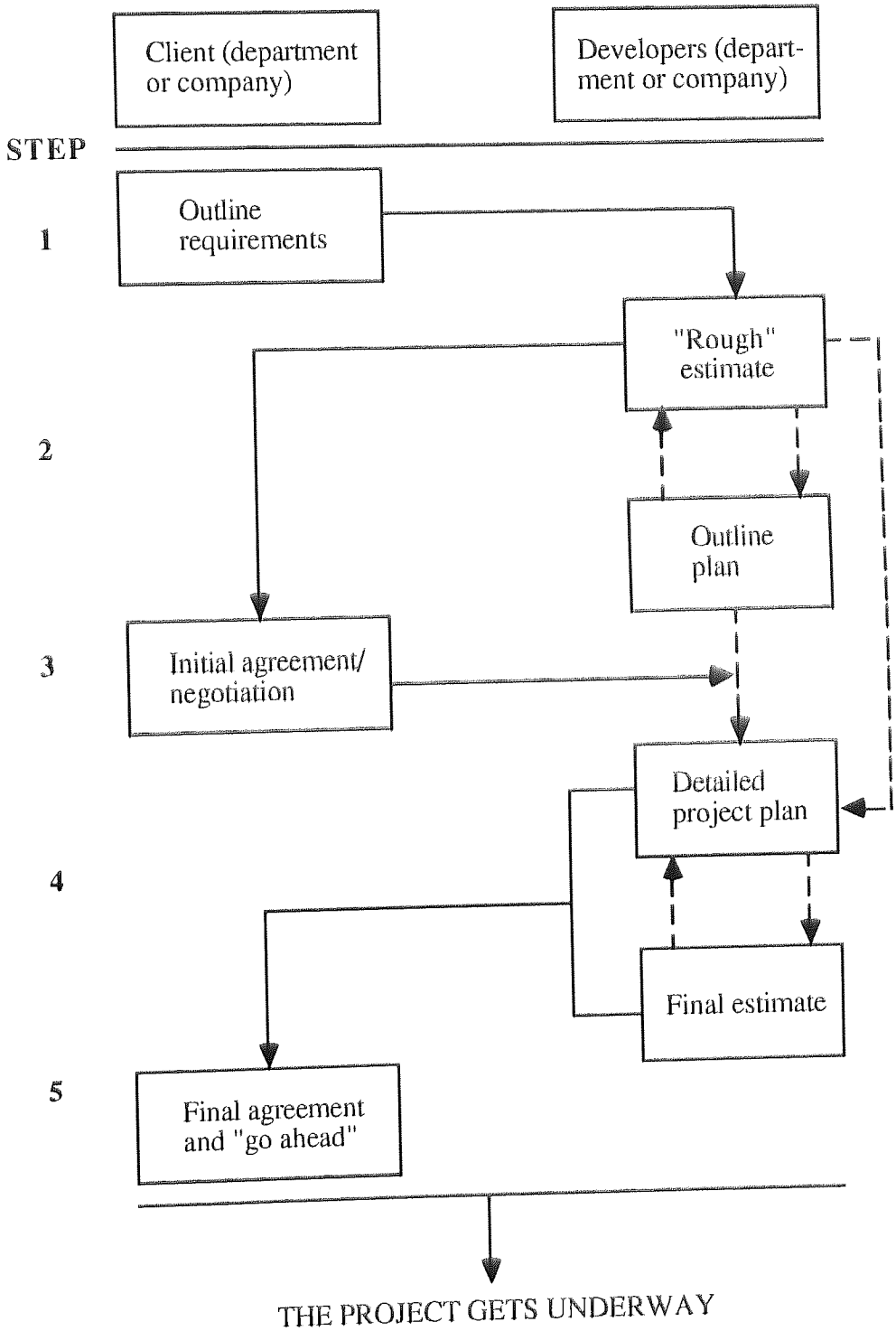
7.3 The EEPS model

The arguments presented in the previous sections suggest that although planning tools are of a different form to SCE tools, they may be used as estimating tools if the true task of estimating is seen to exist at a point in the life-cycle which is much later than SCE tools are commonly seen as being useful. This point will be investigated by making use of the rule deduced in §7.1.4. While 'tackling large projects' is almost certainly a by-product of the Chapter 6 survey which targeted large companies only, the second 'charging a client' factor is the most intriguing. This factor holds out the possibility that project managers are only using SCE tools where they can be used in financial negotiations with a client. In particular, it should be noted that 'high requirements volatility' and 'charging a client' can both be traced to the customers influence: the client is the source of the former and demands explanations of the latter. Furthermore, the only case where a company stopped using their SCE tool was also the company which stopped charging (inter-departmental) clients (see again §7.1.2). Does this "need to show" issue dictate the type of SCE tool that is actually being used by current project managers? Furthermore, are project managers right to dismiss the use of SCE tools when the "need to show" is removed? These questions will be addressed below.

7.3.1 *Modelling the influence of a client during estimating*

The EEPS (early estimating and planning stages) model (see Figure 7.1), focuses on the involvement of the client in the negotiation of the cost and functionality of the proposed system. In particular, the EEPS model represents the way in which the clients involvement can lead to a conflict in the time at which estimates and plans are produced. The model is based on a naïve understanding of the process and the

Figure 7.1 : The early estimating and planning stages (EEPS) model



experience of the IED4/1/1426 project collaborators who have substantial practical experience in developing software systems. The EEPS model has five key steps:

1. The client sends an outline statement of requirements to the development department (or company).

2. An initial 'rough' estimate is generated by the developers based on these requirements (possibly using an outline plan).
3. The client accepts the initial estimate (possibly negotiating more detailed specifications).
4. A detailed project plan is created by the developers, and iteration between the plan and estimate ensures the plan is within budget.
5. The client agrees to the budgets and the system is developed.

Steps 1 - 3 would form part of the feasibility/requirements stage of a "Waterfall" life-cycle, while step 4 would certainly be completed by the end of the analysis stage. As can be seen, however, the EEPS model is inconsistent over exactly what an estimate is. Specifically, is an "estimate" a rough guide (as at step 2), or the result of applying numbers to the detailed project plan (as at step 4), or do both types of estimate exist? If an estimate is a rough guide, then current estimating tools would be particularly useful since they can be used at the beginning of the project (e.g., as early as the feasibility/requirements stage), and are based on minimal information (allowing estimates to be generated even before a project plan). However, if an estimate is a bottom-up, plan-based number-assigning task, then three important conclusions can be drawn:

- *estimating tools are approaching the problem from the wrong direction* (since managers perceive estimating to be anchored to the task of planning and thus SCE are estimating too early to produce sensible results);
- *the initial (rough) estimate is not viewed as a "true" estimate* (since an estimate cannot be created before a detailed plan if estimating is truly seen as being bottom-up);
- *the real process of estimating is perceived to be at the planning stage* (which is currently supported by project management tools, not estimating tools).

The critical point in the EEPS model, therefore, is step 3, the point at which top-down planning based on an initial estimate gives way to bottom-up estimating subsumed within the planning process. But at which point are estimates produced, and does this conflict between an estimate as a top-down guide or as a bottom-up task really exist?

7.3.2 *Validating the EEPS model*

To test the validity of the EEPS model, a telephone survey was carried out in which the following three questions were asked:

1. "On receiving the users' global requirements, which do you do first: create a plan or generate an estimate?"
2. "Do you see estimating as a top-down or bottom-up task?"
3. "Do you see planning as a top-down or bottom-up task?"

The questions on the survey were designed to test whether there is a place for high-level estimating tools at a stage before a detailed plan has been produced; whether the apparent inconsistency the model suggests between the two types of estimates really exists; and whether bottom-up estimating can be tied to a top-down process of planning.

Table 7.9 : Results of the EEPS telephone survey (N=17)

	Yes	No
1. Do you estimate before producing a plan?	11	6
2. Is estimation a top-down task?	5	12
3. Is planning a top-down task?	14	3

Since the survey attempted to distinguish between points early in project management where the use of estimating and planning tools may be considered and/or rejected, it was important to gather a sample of managers who had knowledge of both types of tool. By "good knowledge" it is meant that the project manager belonged to a company which used, had used, or had evaluated both estimating and planning tools. From the Chapter 6 survey, the sample of 25 such companies were used of which 17 could be contacted and all of those contacted agreed to take part in the telephone survey. The smallness of the sample reflects the fact that relatively few companies have any knowledge of estimating tools and as such, no statistical tests can be presented here. Nevertheless, the results (see again Table 7.9) suggest that:

- most project managers produce an estimate before a plan;
- most project managers see estimation as a bottom-up task;
- most project managers see planning as a top-down task.

The first point ("Do you estimate before producing a plan?") is a borderline result and reflects the confusion that project managers seem to have over the process of estimation. Can an estimate really be produced before any planning has been carried out? According to this survey, 11/17 project managers would say "Yes". This result also confirms that there is a place for estimating tools within the early feasibility/requirements stage since estimating tools are the only tools which can provide estimates at this point.

According to this survey, 12/17 project managers see estimating as a bottom-up task. However, the second point ("Is estimation a top-down task?") questions the nature of the initial (rough) estimate and places the task of estimating at step 4. But, how can estimation be a bottom-up process if it typically occurs before a plan is produced? From what top-level information does the estimate derive if it is produced before a plan?

The only possible answer is that the project manager is using the outline requirements to search for past projects with analogous functionality and then generating estimates of cost and project duration by analogy. However, while it is known that estimating-by-analogy is the most popular method of generating an estimate (e.g., Heemstra and Kusters, 1991), unless a procedure for gauging the similarity of current and past projects is in place, the technique should be more properly described as "guessing-by-analogy". Although there have been recent attempts to structure estimates produced this way (e.g., Corbett & Kirakowski, 1992), a lack of rigour here may force the project manager to manage a project which is already based on wildly inaccurate estimates. When projects fail, it is perhaps exactly this point at which the first seeds of disaster are sown.

The third point ("Is planning a top-down task?") has the most support from the sample, since 15/17 project managers support the view that planning is a top-down task. This result would seem to suggest that the bottom-up nature of estimating is because it is related to the step 4 planning stage of the EEPS model. This would also lead to the conclusion that project managers are justified in expecting estimating tools to generate estimates in a bottom-up (not top-down) mode. As §7.2.1 has pointed out, however, estimating tools are designed to be used at the earliest stages of project management, and so they suffer from the fact that project managers may perceive them as producing estimates which have no real validity (those at step 2). This is in spite of the fact that estimating tools would allow a more methodical approach to estimating, since as the project progresses the accuracy of project information would tend to increase, and thus more accurate estimates could be produced.

7.3.3 *Summary of the EEPS results*

These results suggest that there is, in fact, a conflict between the use of project management and estimating tools stemming in part from a contradiction in project managers' minds about what estimating is. Although an early estimate is typically produced before a plan, the "true" process of estimation seems to be subsumed within the planning process. At this point estimating tools would appear to be less useful, since if the client wishes to further negotiate the cost, the argument would be more sensibly based around the proposed project plan rather than on the way in which an estimating tool generates its result. Since project management tools are in almost universal use, and would be seen by project managers as operating at the right level, it would appear they are taking on the role of an estimation support tool.

While it is true that a number of estimating tools have been developed within a suite that includes planning tools (for instance, the estimating tool PMS BRIDGE and the planning tool PMW), it should also be remembered that 91% of managers continue to see estimation as a problem. The reason would now seem to be clear: a project manager would rather estimate using a planning tool which does not constrain the way in which an estimate is produced even though it is only SCE tools which can apply meaningful numbers to this plan.

7.4 Conclusions

The EEPS model presented here suggests that there is a conflict in how an estimate is perceived, and that this conflict seems to result in project management tools - rather than estimating tools - being used to support estimating. The conflict appears to be the way in which managers see the "true" process of estimating as being at the detailed planning stage, where a more detailed picture of the project is available. At this point estimating models still for the most part use information which was TAD-early at step 2 in the EEPS model, although the plan is only an approximation of the project and two managers could disagree on the detail. Since few estimating tools have been designed to deal specifically with task-level information, it appears they have failed to address the problem of estimation at a level which is in tune with the way project managers perceive the task.

What seems to be needed is a clear distinction between two types of estimating tool. Firstly, the tool which generates estimates at the early (step 2) stage of the EEPS model should not be confused with the estimation process which goes on at the (step 4) task level. The initial figure is a negotiation point, and the tool which generates this figure would be more properly classified as a bidding or tendering tool. Secondly, when described as an

estimating tool, what managers expect is a tool which specifically operates at the task level. Without such a clearly defined functionality it is no surprise that industrial experience of such tools continues to be poor while the literature still produces recommendations to project managers not to use estimating technology (Lederer and Prasad, 1992). This negative view must be balanced with other studies which suggest that industry is attempting to develop a well-formed metrics programmes (e.g., Grady & Caswell, 1987; Hager, 1989; Linkman & Walker, 1991; Goodman, 1992), although there are few examples of where such programmes have produced quantifiable results.

Where results have been reported they have tended to be spectacular. For instance, following the SEI process maturity framework (Humphreys, 1989), Humphreys *et al* (1991) implemented an improvement programme at Hughes Aircraft Corporation. After an investment of $445 000 over two years Humphreys *et al* suggests this programme brought an annual saving of $2 million to Hughes Aircraft Corporation (p11). Using a goal-driven approach as part of the ESPRIT-funded PYRAMID project (Project 5425), Möller & Paulish (1993) reported that consortium partners achieved improvements in bug-detection and quality of up to 35% (p7). Although the improvements in both cases could be put down to something being done and reservations have been raised about the bluntness of SEI's 101-question assessment form (Bollinger & McGowan, 1991), it is clear that some advances are being made in the control of software costs.

The argument presented here, therefore, suggests that 'Option Zero' is false only in so far as project managers are forced to generate estimates before producing a plan. Thus, planning tools are not in use at this point and this is where SCE tools can provide invaluable help. This is not a necessary state of affairs, however, since project managers might still insist that there is not enough information for even SCE tools to work. If this were true, then one could question the project managers' ability to generate estimates. After all, being derived from an historical database, SCE tools are simply mechanical forms of past experience. But it seems clear that the goal of future research should be to build accurate, task-based estimating tools which fit current practice and prove to project managers that such tools can, in fact, provide genuine and much needed help with estimating software costs. This goal is indeed an important one, since the cost of overruns can have serious effects on a company's profitability.

This conclusion also holds out the possibility that although current SCE tools may be inadequate for the job for which they are currently used, at least the nature of the required SCE model has been identified and revolves around the need to estimate with poor data and then re-estimate throughout the life of a project taking into account the effect of changing a system before completion.

Furthermore, if high requirements volatility and charging a client are taken to be symptoms rather than causes of the use of SCE tools, then a more interesting suggestion surfaces. Specifically, the changes and charges reflect the interaction between the developers and their clients. These symptoms might reflect the developers need to show a client that every effort is being made to produce an accurate estimate. If this were true, then, as stated at the outset of this chapter, there need be no belief that SCE tools increase accuracy but only that it can be demonstrated to a client that more than one method is being used. Used in this way the SCE tool is a method by which issues of software development are brought to the mind of the project manager rather than specifically solving the problem. This would also imply that the use of SCE tools has not been taken seriously and it would be unlikely that a company would expend too much effort or money collecting the data essential for calibrating the tool. But if this is the current state of industry, what changes should be made to make SCE tools more suitable for use by project managers in general and for hybrid projects which contain both conventional and KBS components in particular? This will be the subject of the next chapter.

8. Defining A Task-Based SCE Tool

"A philosopher and theologian were having an argument. 'You're just like all philosophers,' the theologian scoffed, and quoted: 'You're a blind man in a dark room, looking for a black cat that isn't there!' 'Agh!' replied the philosopher, 'but you *would* find *it!'* " Attributed to William James (in Michie & Johnston, 1984, p188).

Summary: *Building on the empirical results of the previous chapters, a definition and specification of a task-based estimating tool is presented. The tool is here called TABATHA. Given that the research is based within IED4/1/1426 it is assumed that the nature and number of tasks within a project are to be provided by the RUSSET tool. The general task-based model is presented along with the seven measurements required to instantiate and validate TABATHA. It will be argued that such a tool can deal with re-estimating as user requirements change because the effect of project changes occurs at the task level, exactly the point at which TABATHA generates its estimates.*

It was originally envisaged that the task of producing suitable metrics models and tools for hybrid systems would be able to follow the same metric→model→tool approach already extensively used for conventional (i.e. non-knowledge-based) systems development. However, the theoretical problems uncovered by the literature review cast doubt on the validity of the approach. It was thus conceivable that the approach itself was invalid and not worth extending to hybrid systems (which, by definition, contain conventional systems components). Much work, therefore, had to be devoted to clarifying what was required of metrics models and tools and establishing whether it was the approach, or the specific metrics and tools so far developed, which were at fault.

In the event, it was found that the conventional metrics approach remained sound but the models and tools themselves were not well-matched to the needs of the end-users. In particular, estimating was seen to be carried out at more than one level of detail during the early stages of a development project. In particular, the existing approach needed to define a task-based estimating tool, rather than the usual attempt at sizing individual components or the system as a whole.

The major problem with proposing a new form of estimating model is that validating such a model would (typically) require at least 3-5 years of data collection. As an example, Boehm's (1981) data on 63 projects stretched back 13 years while data collection programmes are seen as adding from 5-10% to the cost of a project (e.g., DeMarco, 1982; Putnam, 1991). Such demands are beyond both the resources and time-frame of this research. For these reasons, the attempt here will be to provide enough detail to allow a follow-up project to construct a task-based model. The rest of this Chapter will therefore set out to describe:

- a basis upon which to define a task-based estimating tool;
- the architecture of such a tool (which is here given the name TABATHA, the *TA*sk-*BA*sed estimating *T*ool-*HA*));
- an example of how TABATHA would generate a bottom-up task-based estimate;
- the method by which TABATHA would cope with volatile requirements;
- an identification of what remains in order to instantiate the model and produce a viable SCE tool.

8.1 Basis for a task-based estimating model

From the research carried out so far, there are known to be four important properties of the way (UK) project managers currently perceive estimating:

1. Estimating is a problem.
2. An early estimate is often produced before a plan.
3. Later estimates are bottom-up task-based estimates associated with planning.
4. Requirements volatility is a key issue for companies using SCE tools.

Point 1 comes from the original (Chapter 6) survey and identifies the need for estimating tools; points 2 and 3 come from the EEPS model survey in Chapter 7 and pin-points when estimates are typically generated, while; point 4 comes from the definition of SCE users also in Chapter 7 and identifies the motivation behind using SCE rather than planning tools. These points suggest that SCE tools are still potentially useful, should be task-based, and allow for continual redefinitions of the user requirements. Furthermore, following Kitchenham's (1992) suggestion that SCE tools are simpler than has previously been suggested, it will be assumed that models of effort and duration have the simplified general form:

$$\text{Effort} = a.(\text{Size}) * \text{Adjustment\_factors}$$

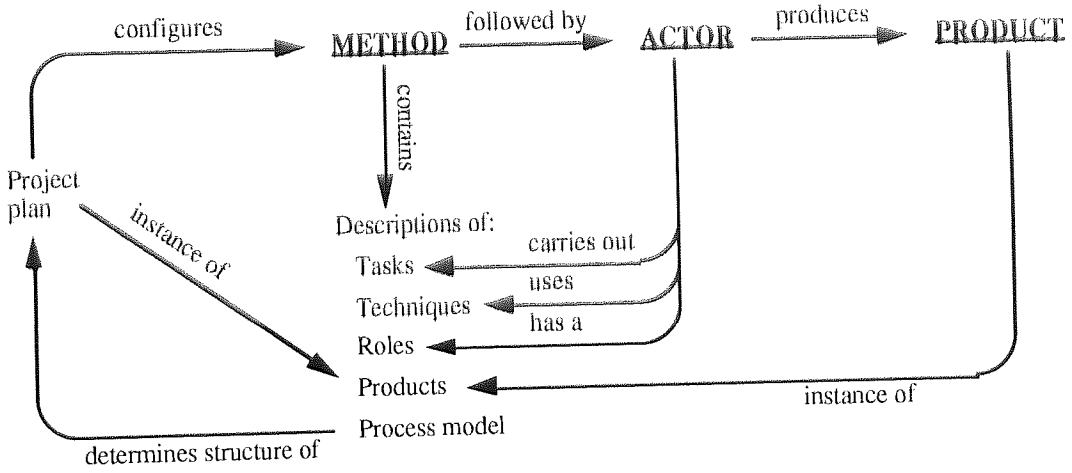
$$\text{Duration} = b.(\text{Effort})^{1/3}$$

Since duration is derived from effort, the focus here will be on defining a model for effort which is clearly based on an analysis of tasks. This gives a top-level (most abstract) model where effort is defined as:

$$\text{Effort} = a.(\sum \text{Tasks}) * \text{Adjustment\_factors}$$

But how should tasks be modelled and what should be measured in order to gain an understanding of their size? These questions will be answered below.

Figure 8.1 : RUSSET's representation of the development process



8.1.1 *A process model of software development*

The elements of software development which are open to measurement will be assumed to follow the development process defined within RUSSET. It should be remembered that RUSSET is the methods integration tool described in §1.1.2 and §1.1.3. The advantage of the model given by RUSSET (see Figure 8.1) is that it contains no reference to the type of methodology in use, and so, RUSSET is able to cater for both conventional and KBS methodologies. Furthermore, by linking the estimating tool to RUSSET's representation of methods, the SCE model can assume that the nature and number of tasks involved in a project will be available from RUSSET. At the heart of RUSSET's model is the statement that software development has three essential components:

1. *A Method.* Otherwise, there can be no identifiable tasks because there is no structure within which the tasks are defined and ordered. Of course, any

project which contains programming can be said to have at least the task "Programming" within its methodology, but this misses the point that the point that the most useful estimating model is one which can be related to a planning of a number of interdependent tasks.

2. *Actor*(s). The people who carry out the tasks within a method. These are the team members and the active ingredients of any software development project. Taking into account the differences in experience and productivity between actors is a key problem in defining an accurate SCE model. For instance, the overhead contained in co-ordinating and communicating between actors forms the basis of the COPMO model.
3. *A Product*. The goal of any development project is to produce products which will be of use to an end-user. Estimates of system size are measurements of this attribute. When measuring the products developed as part of a project, however, it is important to realise that the program itself is only a fraction of the deliverables presented to the user. Other products include reports and all other paper-based documentation.

In other words, according to the RUSSET model, a task is a feature of the process by which a method is followed by actors (or agents) to produce products. This is what software development is all about. As can be seen, these are a number of features which are all candidates for measurement. Namely: the method must give a description of the tasks, but (may) also contain information on the techniques to be used, the rôles an actor can carry out, the specific products to be produced, and a description of the way in which the method is to be configured (e.g., simple Waterfall, Spiral, prototyping, etc.). As Harris-Jones *et al* (1993) pointed out, however, not all methods will give descriptions of each of these features, but they are clearly important features of a project and a project manager will have to give some thought to each point when developing a project plan.

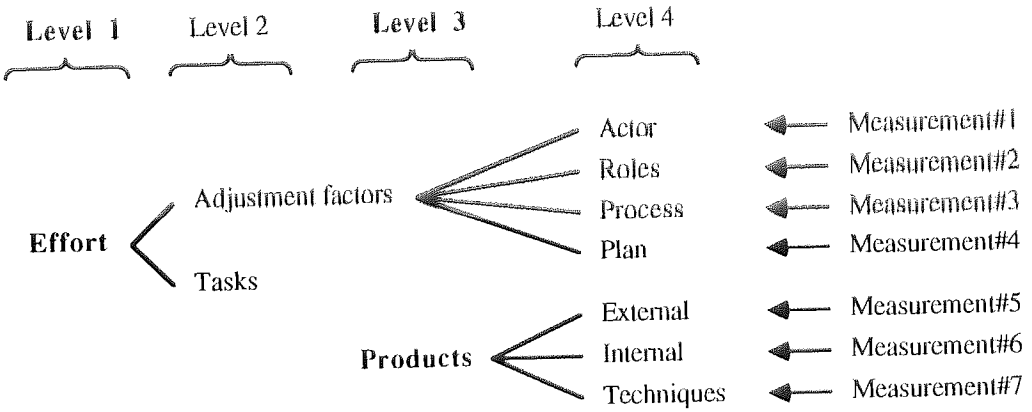
The project plan itself is structured according to the model of the development process stated or implied by the method. It is at this point that the task-based model relies on RUSSET having an adequate representation and configuration of tasks within a method. What is important here is that the task-based nature of software development can be captured by the relationship between a project plan and an understanding of the development tasks involved. The next step is to combine these elements into a model of effort.

8.1.2 *Defining a set of measurements*

The degree to which each of the elements described above influences effort can be shown by attempting to deduce a model whereby each feature is defined within a task-based framework. Similar to the approach taken by Readdie *et al* (1989) in §4.2.2, the general model is defined here simply such that:

$$\text{Effort} = \sum \text{Tasks} * \text{Adjustment\_factors}$$
$$\text{Task}_i = \sum \text{Products}$$
$$\text{Product}_i = \text{External}_i * \text{Internal}_i * \text{Technique}_i$$
$$\text{Adjustment\_factors} = \text{Actors} * \text{Roles} * \text{Process} * \text{Plan}$$

Figure 8.2 : Metrics classification tree defining the task-based estimating model



In other words, the effort to carry out a task is defined as the sum of the effort to develop each product (e.g., document, code, etc.) associated with a task. The remaining elements from the RUSSET model are taken to be adjustment factors which affect the nominal effort for the project as a whole, i.e., the number and quality of the personnel, development strategy used and constraints on the plan itself. Based on these features, a metrics classification tree (after Porter & Selby, 1990) can now be drawn (see Figure 8.2). This classification tree suggests that a task-based estimating tool would require seven measurements to be taken of the development process. These are described below:

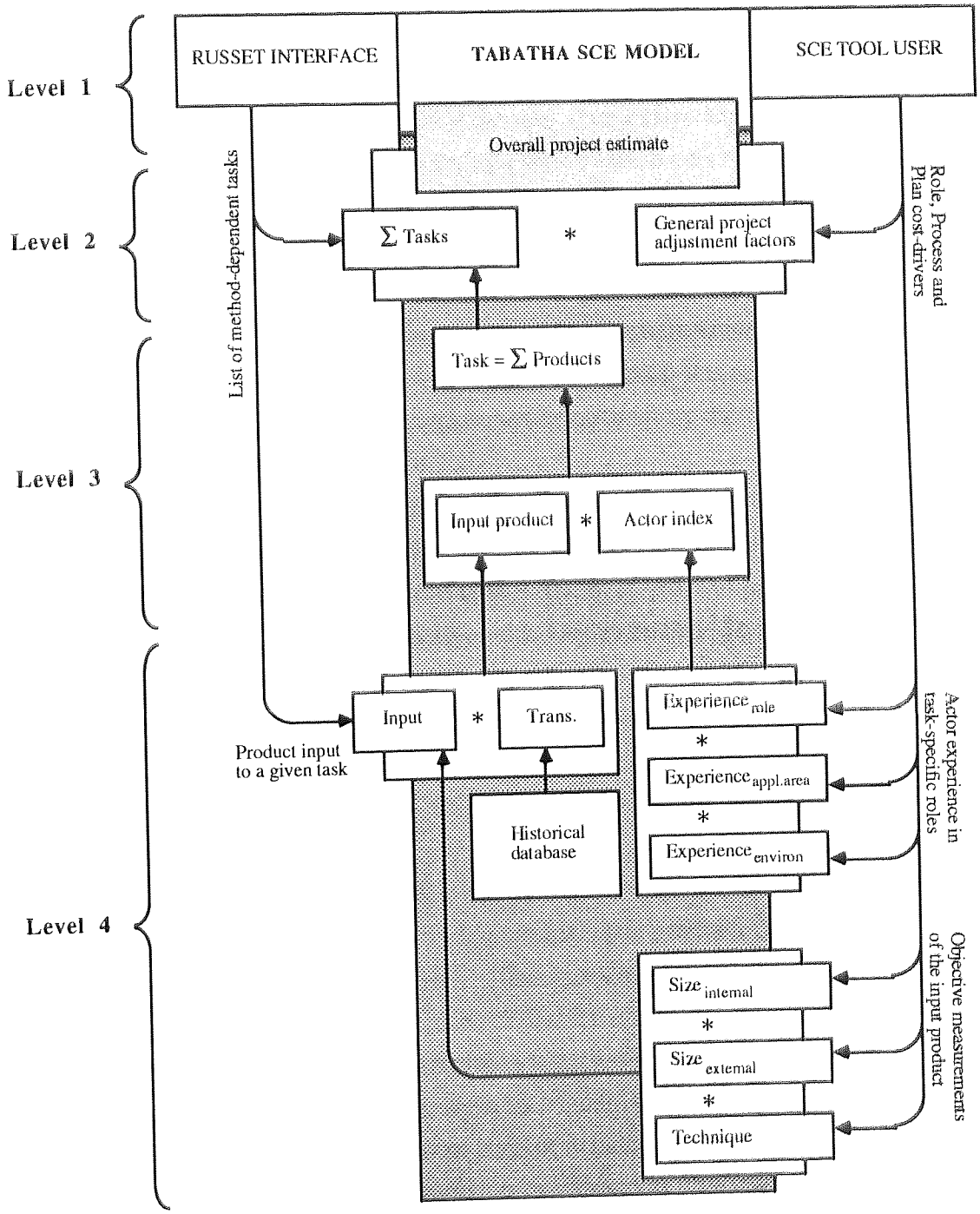
1. *Actor.* An adjustment based on an assessment of the qualities of the team members. This is equivalent to personnel factors contained in most SCE models but applied to each estimated task. Personnel factors are, however,

the most difficult measurement to take since a direct measurement of individual productivity may encounter difficult political problems within an organisation, while the use of subjective ratings is often seen as a drawback of current SCE models.

2. *Rôles.* An adjustment based on the different functions performed by different actors. It should be noted that a person could have more than one rôle in a project (e.g., analyst and programmer), but rôles is here simply a high-level measure of the co-ordination and communication complexity of a project.
3. *Process.* An adjustment based on the type of methodology used in development. For instance, Boehm (1981) found that incremental development projects were more efficient than simple Waterfall strategies. This point, in itself, may have a bigger influence on the cost and effort required by a project than any other factor.
4. *Plan.* An adjustment based on the constraints placed on a plan. In effect, this is a measure of the (external) size and (internal) complexity of a documented product. However, the specific importance of measuring the plan itself is to calculate scheduling and manpower constraints placed on the project. This is equivalent to the COCOMO product cost-drivers.
5. *Product external.* A measurement of the size of the product to be developed. To differentiate this factor from a measure of (internal) structure, size is defined within TABATHA as a measure of external size. This measure will change depending on whether the product is a (paper-based) document or system code.
6. *Product internal.* A measurement of the (internal) structure of the product. As explained in §3.2, a notion of internal complexity is required to adjust the apparent size of a product. Again, this measure will change according to the nature of the product itself.
7. *Technique.* An adjustment based on the effect of using different techniques. A measure of the sophistication of techniques being used reflects a notion of the productivity expected from a project. This is equivalent to the COCOMO project cost-drivers.

Interpretations of these seven factors are based on an understanding of the features of software development represented within RUSSET's software development process model (see again Figure 8.1). As can be seen, all seven features are at least potentially measurable while the adjustment factors 1, 2, 3, 4 and 7 reflect a number of properties already contained within a number of SCE models. The question now is how this information would be combined into a usable tool useful to project managers.

Figure 8.3 : TABATHA - a task based estimating model



8.2 A description of TABATHA

This section will give an outline specification and description of the task-based estimating tool which is here given the name TABATHA (*TA*sk-*BA*sed estimating *T*ool-*HA*). The description provided attempts to explain how the model outlined in the previous section would interact with RUSSET and a (human) user. The architecture of TABATHA is

represented in Figure 8.3. As with most SCE models, a notion of the "size" of a system is the key input to the TABATHA models of effort and project duration. There are four levels of relationship which are defined here top-down and then used bottom-up by TABATHA to generate an overall estimate of system size. These four levels assume that:

1. *The size of a project is a function of the sum of task sizes multiplied by the product of adjustment factors.* A task is an activity which transforms one or more products into another set of one or more products. The size of a task is therefore related to the size of the product(s) it uses and the size of the product(s) it aims to produce. The range of tasks involved in the project and their relationship is the key functionality provided by RUSSET; TABATHA models a range of adjustment factors based on the features Rôles, Process and Plan and a means of calculating the relationship between the tasks. The constraints on the project would be rated by the user in order to produce the required adjustment quotient.
2. *The size of a given task is defined by the nominal size of the product(s) produced adjusted by the skill level of the actor(s) carrying out the task.* The actor index is an adjustment based on skill defined here in terms of years of experience in the project rôle, application area and development environment. In other words, the more experienced the actor the smaller the effective size of the task, even though the product(s) remain the same. The experience values are provided by the user. $Experience_{role}$ appears because it becomes relevant here to deduce the impact of individuals on a specific task. "Rôles" appear in '1' as a more general description of the project team.
3. *The nominal size of a given task's product(s) is calculated as the size of the input product adjusted by a transformation index.* The product(s) used as input to a task would be provided by RUSSET. The transformation index is the typical ratio between the size of an input product and its output. The values of these indices would be calculable by gathering data from a number of development projects. A transformation index is effectively a size model operating on a micro-project level. The task level is seen as the lowest level of detail required.
4. *The nominal size of an input product is calculated by applying a number of metrics to the products generated during the project.* The metrics would seek to determine the physical size and structure of the product ($Size_{external}$ and $Size_{internal}$, respectively). The suggestion here is that both size and structure are needed to determine the actual amount of "work" contained within a product. These values would have to be calculated as the products are produced, and with a complete set of transformation indices in place, would imply the size of subsequent products at each point in the life-cycle.

These four levels represent an instantiation of the metrics classification tree described in §8.1.2. At the highest level, the estimate is produced on the basis of the tasks involved in the project using the historically-derived transformation indices and represent estimating using only levels 1 and 3 of TABATHA. This type of estimating is equivalent to the way current SCE tools work and where the entire project is the single task "develop system." At a more detailed level when the user is creating the project plan, the user would assign personnel to carry out each task and therefore would have details of their experience in the rôle, application area and development environment. This would allow estimating using levels 1, 2 and 3 of TABATHA. As the project is carried out, the size and structure of the actual products are calculated and used to re-calculate the estimate as the project progresses. At this point, estimation would be carried out using all 4 levels of TABATHA.

8.3 TABATHA applied at the task-level

As an example of how TABATHA would create bottom-up estimates at the task level, consider the gross simplification that a method follows a 6-stage Waterfall model. In this case there are only 6 tasks, 6 input products and 6 output products. With example metrics in parentheses and possible transformation indices also given, TABATHA would represent the task list as follows:

Task 1 = produce user requirements
- Input product 1 = user requirements (number of bullet points)
- Output product 1 = user requirements document (lines of text)
- Trans index = 10

Task 2 = produce system specification
- Input product 2 = user requirements document (lines of text)
- Output product 2 = system specification document (lines of text)
- Trans index = 2

Task 3 = produce system design
- Input product 3 = system specification document (lines of text)
- Output product 3 = system design document (lines of text)
- Trans index = 2

Task 4 = implement system design
- Input product 4 = system specification document (lines of text)
- Output product 4 = implemented system code (lines of code)
- Trans index = 150

Task 5 = integrate system code
Input product 5 = implemented system code (lines of code)
Output product 5 = integrated system code (lines of code)
Trans index = 1.1

Task 6 = test system code
Input product 6 = integrated system code (lines of code)
Output product 6 = tested system code (lines of code)
Trans index = 1.1

To carry out a task-based estimate, it is assumed the project manager would begin to apply a number of project and personnel values to the project plan. For instance, assume three people are assigned to the project. "Person1" is the project manager and has 10 years experience as a project manager, 1 years experience in the application area and 5 years in the development environment (actor index=0.2). "Person2" is an analyst programmer and has 2, 2 and 1 years experience respectively (actor index=0.5), while "Person3" is a programmer and has 1, 0, and 0.5 years experience respectively (actor index=1.0). The actor index in each example is fictitious but can be based on formulae produced by, for instance, COCOMO. Tasks with more than one actor take the average value.

Task 1 and Task 6 are to be carried out by "Person1" and "Person2", while "Person2" and "Person3" carry out Tasks 2-5. Assume also that the users define 20 bullet points of functionality, then:

Task 1 = produce user requirements document
where,
Nominal product size = 20 bullet points * 10
Task size = 200 nominal lines of text * $\frac{0.2+0.5}{2}$
= 200 nominal lines of text * 0.35
= 70 effective lines of text

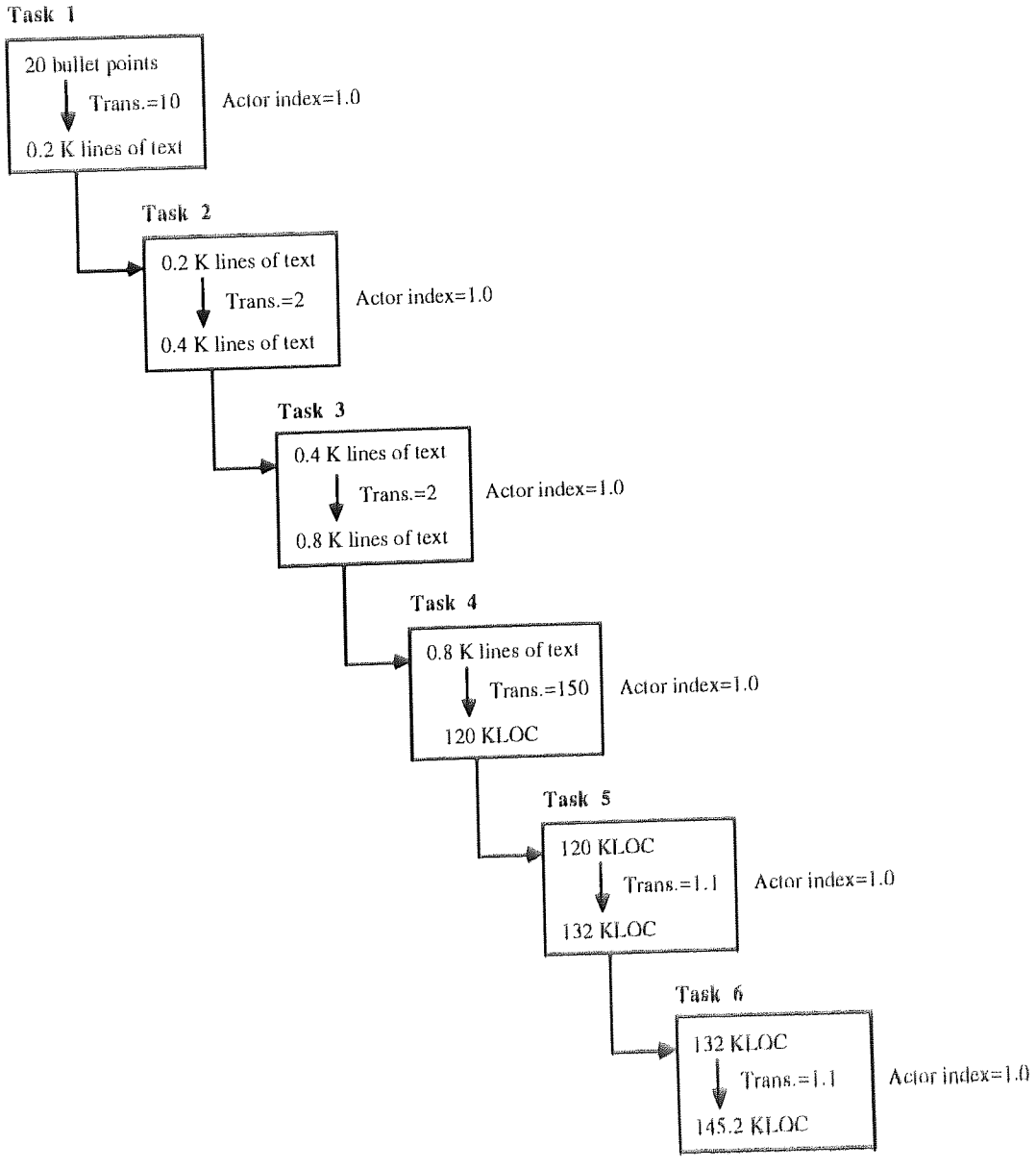
From this information, all later product sizes can be similarly estimated and thus, an overall estimate adjusted by the size of the first stage product can be calculated. The indices given above suggest that 20 bullet points of requirements at Task 1 stage would be transformed into 145.2 KLOC by the end of Task 6 if the actor index and general project adjustment factors are set to one (see Figure 8.4). It is this 145.2 K size estimate which becomes the first input to the effort and duration models. For instance, using this figure and borrowing

the *a* parameter from Intermediate COCOMO's effort and duration models in Semi-Detached mode (see again §5.2.4), TABATHA would generate the result:

$$\begin{aligned} \text{Effort} &= 3.0*(145.2) \\ &= 435.6 \text{ person-months effort} \\ \text{Duration} &= 2.5*(435.6)^{1/3} \\ &= 18.95 \text{ months duration} \end{aligned}$$

Dividing effort by duration suggests that 23 people are required, which is very close to the average of 24 people which Boehm (1981) suggests is typical of projects in semi-detached mode.

Figure 8.4 : Transformations of products in a Waterfall life-cycle



The approach described here has some similarities with that taken by Detailed COCOMO (Boehm, 1981, pp344-474). Detailed COCOMO seeks to increase the sensitivity of Intermediate COCOMO by changing the effort model parameters by phase and the cost-driver values by granularity down to estimating individual modules. Since a task is defined in TABATHA in terms of its products and a COCOMO module can be seen as a product there is a superficial similarity between TABATHA and Detailed COCOMO. However, the key difference between the two models is that Detailed COCOMO does not estimate on the basis of the input→output interaction between these products or provide a framework within which to link the task products together.

In Detailed COCOMO, estimates are strictly independent estimates of system or sub-system components which are then summed to produce an overall estimate. The only notion of there being a structured method is on the assumption that the statistical data is collected within an environment where a method has been used. TABATHA, on the other hand, requires an understanding of the development method being used in order to structure the estimate around the tasks which produce the products. Thus, unlike Detailed COCOMO, TABATHA would be able to continue to adjust the overall estimate and take into account imposed deadlines and changes to requirements as the project progressed.

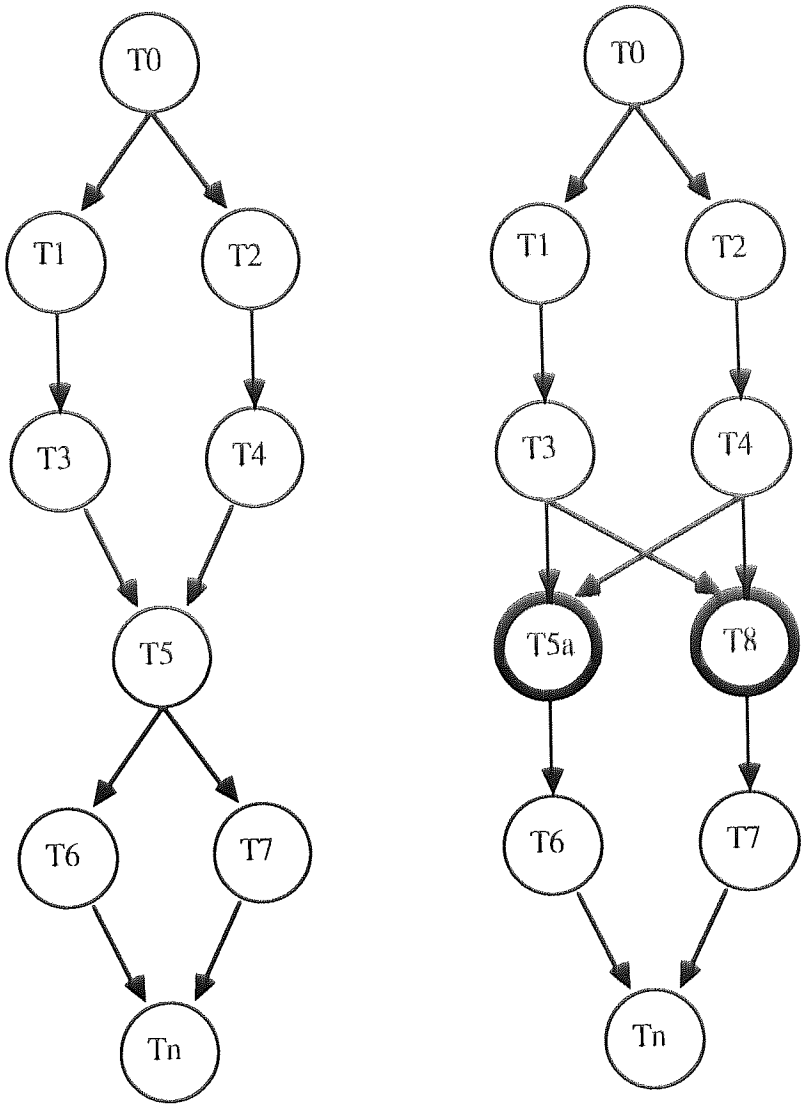
8.4 The effect of requirements volatility

Those companies which used SCE tools were characterised by the experience of requirements changing on most or all projects (see again §7.3). This effect would be modelled by TABATHA not as an adjustment factor within the model itself, but as a method by which the effect of changes can be calculated. The argument here is that if requirements typically change within an organisation then the parameters of the SCE model should reflect this volatility. There would be no point introducing an adjustment - such as COCOMO's RVOL - if this situation is typical since subsequent projects would be in line with the historical database and RVOL would always be rated as "average". What is needed instead is a method of gauging the effect of RVOL on the estimate already produced.

Consider a change to a project's structure by representing the project in PERT-like form (see Figure 8.5). The beginning of a project is represented by the node T0, and the end by Tn. The intermediate nodes are the tasks numbered T1 to T7 in project 1. Presume now that the user requirements change such that task T5 is subdivided into two sub-tasks, the majority of the work in T5 being preserved in T5a while the new task T8 implements the changes in the requirements. How will this change affect the overall cost and effort of the project? By analysing the project on a task-by-task basis, TABATHA allows a revision of the classic

approach to be applied, whereby the extra effort is the rework required to develop tasks T5a and T8.

Figure 8.5 : The effect of requirements volatility on a project's structure



However, the SLIM and COPMO models also suggest that there is an overhead incurred by the interaction of personnel on a project. In the same way, therefore, it must be assumed that there is an overhead involved in a change to a project. This can be seen by the increase in complexity of the PERT-like chart. This will affect both effort (in separating and designing T8 from T5 and T5a), and in the duration of the tasks since T6 and T7 are now dependent on T5a and T8 being complete. Therefore, while the initial estimate must be based on a linear model, re-estimating must be based on a more complex model which focuses on those tasks which are affected by the change in requirements. Such a model would have the form:

$$Estimate_{new} = Estimate_{old} - Estimate_{tasks} + a.(Re-estimate_{tasks})^{b}$$

where,

$Estimate_{new}$ = Estimate for the revised project (Project 2)
$Estimate_{old}$ = Original estimate (Project 1)
$Estimate_{tasks}$ = Estimate for the affected tasks (T5)
$Re\text{-}Estimate_{tasks}$ = TABATHA estimates for the revised tasks (T5a and T8)

The re-estimate model given above would be a local model based on the specific transformations between tasks using the same data as that which develops TABATHA. Such a model would also overcome the problem of deducing the effect of requirements volatility, since it is only the affected tasks within the project which are re-estimated rather than the project as a whole. The model also reflects the non-linear relationship between work added to a project and the effort expended in carrying out this additional work. In this way, TABATHA effectively contains SLIM- and COPMO-like micro-models working within a re-estimation framework which sits more reasonably within the current working practice of UK project managers.

8.5 A metrics programme to validate TABATHA

As mentioned earlier, while the relationships between products are assumed to be linear - following Kitchenham (1992) - the actual values of the transformation indices and the value of the parameter *a* for a particular organisation are a matter for substantial data collection. Such a metrics programme would collect the seven sets of data specified in §8.1.2. There are five adjustment factors and two size measurements within this set. The metrics programme required would therefore have 10 steps, as follows:

1. *Decide on the level of task-based reasoning to be carried out for the estimate.* It is not necessary to collect data on every product generated by the method. The key point is to have a level of detail which coheres with the level of detail required by a project manager identifying the tasks within a plan.
2. *Identify the products associated with each of these tasks.* If the key products are specification and design documents, milestone reports and coded programs, then the tasks which generate these products are the ones to be measured.
3. *Devise a set of metrics to measure the size and structure of each of these products.* The meaningfulness of the resultant model will critically depend on this step, but at present, it is difficult to be sure how to guarantee a certain metric captures an

important property. The complexity of the process itself suggests that simple - rather than sophisticated and detailed - sets of measurements are likely to be adequate.

4. *Devise a means of recording the results of '3', the five adjustment factors and the cost, effort and duration of individual projects.* Ideally, these measurements would be taken automatically, and there is some research being conducted into this problem (e.g., Côte & St.Denis, 1992). What will continue to remain a problem is guaranteeing the veracity of the data collected.
5. *Begin collecting the relevant data on a series of live projects.* As has been pointed out by a number of researchers (e.g., DeMarco, 1982; Grady & Caswell, 1987; Kusters & Heemstra, 1992), it is clearly vital to get the co-operation of the people involved in the data collection such that the quality of the data can be relied upon.
6. *When sufficient data has been collected, perform linear regression to deduce the transformation indices from one product to another.* This step produces the size model which has been presumed here to be a linear relationship between a product input to a task and its resultant output product. It is a matter of debate when "enough" data has been collected.
7. *Use linear regression to deduce the relationship between size and effort.* Following Kitchenham (1992), it has been assumed throughout that there is a linear relationship between the measured size of a product and the effort required to produce it.
8. *Use regression to deduce the relative effect of the five adjustment factors on the nominal size model in '7' to produce a micro-effort model for each task.* This step seeks to adjust the nominal size model deduced in the previous step to a model of the effective (i.e., actual) effort required to produce each product.
9. *Use regression to deduce the relationship between effort,* E, *and duration,* D. Again, following Kitchenham (1992), it has been assumed throughout that the relationship between effort and duration has the form $D=a.E^{1/3}$. This could be presumed in the calculation by performing regression between D and $E^{1/3}$. Alternatively, to test the original assumption by taking logs such that the equation now becomes $\log_{10}D=\log_{10}a+1/3.\log_{10}E$. Since this equation now has the form $y=c+mx$, by plotting $\log_{10}D$ against $\log_{10}E$ it should be seen that the gradient of the line is 0.333 and the y-intercept is $\log_{10}a$.
10. *Use linear regression to deduce the relationship between effort and £cost.* This step only makes sense on the assumption that the work carried out during the development project is costed and charged to a client. If no direct charging mechanism is in place, then the transformation from person-months effort to financial cost becomes a redundant step.

The actual cost of carrying out this programme would depend on the mechanisms already in place to collect project data. The survey in Chapter 6 suggest that most companies (78%)

collect at least the basic effort data on most projects while 94% produce plans upon which to define step 1. There is reason to believe, therefore, that the cost of validating TABATHA would be in the lower end of the 5-10% range cited earlier.

8.6 Remaining problems

Given the specification of TABATHA presented here, there are still a number of problems which would hinder the development and validation of the tool. These problems are identified and discussed below.

8.6.1 *Measurements of paper products*

What is particularly novel about the description of TABATHA given here is the need to measure the non-code products such as reports and other documentation. The cost of such products was highlighted in FPA Mk.II which saw the Documentation technology factor (TCF) to be the most expensive element in the FPA model, commanding double the degree of influence (DI) of other factors.

For early estimates it is important to measure the requirements and specification documents themselves. But how can this be done? Superficially, one could suggest that lines of text could "size" the amount of work contained within a document in the same way that lines of code is used as a (simplistic) measure of program size. However, in the same way that lines of code is extremely sensitive to individual programming style, this measure of text will also be sensitive to individual writing style. The example given in §8.3 suggests that requirements are given in bullet points but this would also assume that items of functionality can be readily separated and noted. Given the propensity of requirements to change a simple method would seem to be appropriate. What remains clear is that some representation must be used.

8.6.2 *Identifying relevant adjustment factors*

It is assumed that if a proper set of adjustment factors are to be identified for TABATHA, then there must be some way of isolating the effects of a particular cost-driver from other influences. If a scientific approach were to be used, this would mean conducting two development projects and deliberately changing one aspect of the development environment and holding all other factors constant. The likelihood of studies such as these being carried out, however, remains remote. Software development is expensive and it is unlikely that any company would be willing to

conduct such experiments. A means of identifying these factors is the greatest challenge for software metrics.

TABATHA assumes that only four factors influence the cost of a project, namely: the experience of individual programmers, the rôles carried out by the development team, the process of development employed, and the constraints applied to the plan itself. This is considerably less than other models have suggested. Care would therefore have to be exercised over the quality of the data used to establish exactly how these properties affect cost. While the years' experience, number of staff and the use of (say) a Waterfall method would be fairly straightforward to collect, it is the fourth set of data which would be difficult to capture. What is it about the project plan which needs to be measured? The supposed complexity? Or timescales imposed on the project? Or something else?

An ability to link the problems of generating the plan to generating estimates is important since it is only at this point in the EEPS model that project managers perceive the true process of estimating to exist. Planning and estimating are inextricably linked. This is why the process model given in §8.1.1 demands that in the same way estimates are meant to affect the plan, there must also be some mechanism by which properties of the plan affect the estimates. Without such a mechanism it would be difficult to say how one influenced the other. The adjustment factor "Plan" might therefore explode into many of the subjective factors used by other SCE models. By focusing on the problems and risks associated with planning rather than estimating, however, TABATHA holds out the possibility of a new assessment of what counts as a problem in the minds of project managers.

8.6.3 *The cost of a metrics programme*

Metrics programmes are expensive and in the competitive world of software development, or, more importantly, where only fixed-priced contracts are put out for bid, the cost of collecting such data may be the end to any profit. Even though 'Option Zero' may be seen to be false it could well be that companies are not prepared to invest the money to develop accurate and useful SCE tools. The number of adjustment factors that could have a potential impact on the cost of a project implies a complex SCE models and increases the demand for large metric databases. All of these problems add to the cost of developing or calibrating a SCE tool.

The solution to this problem would seem to be to develop simple models with only a few adjustment factors. The simpler the model, the more likely a project manager is

to use it. The less data required to validate the model the cheaper the development process. This point seems to be the direction SCE research is now going, where the measurement of a problem must be as simple and as direct as possible. For instance, Pfleeger (1993) notes that:

> "The greater the distance from measurement to problem, the less likely developers are to use the measurement. For this reason, simpler measures are better than complex ones. If a problem can be understood with one piece of data instead of several, so much the better. For metrics, more is not necessarily better (p73)."

Of the two models identified in Chapter 7, the simplest form of SCE model is the bidding/tendering tool placed at step 2 of the EEPS model. This high-level model is a surrogate for TABATHA since it operates before a plan has been produced and when the details of the project are still unclear. Although such models have been developed for conventional systems, the question still remains whether the same conventional metrics and techniques can be used to develop size and structure metrics and SCE models for KBSs. If so, then the control of hybrid projects becomes the use of the same set of project management techniques, regardless of the nature of the system. In other words, can conventional size and structure metrics be extended to KBSs, such that a measure of quality and SCE models of effort (and thus duration) can be developed? This will be the subject of the next chapter.

9. Developing Hybrid Metrics and Models

"Our doubts are traitors and make us lose the good we oft might win, by fearing to attempt it." William Shakespeare ('Measure for Measures').

Summary: *In order for TABATHA to be useful for hybrid projects, it is necessary to assume that the languages used to develop knowledge based components would also be susceptible to the same form of (metric) analysis as conventional languages. Prolog is taken to be a mainstream KBS development language and so a description is given here of an analysis of 80 commercially-developed Prolog programs. It will be argued that by refining the definition of three "classic" metrics (Software Science, cyclomatic complexity and data fan-in/out), excellent models of both Prolog structure and size can be developed. This is the first step to developing the more sophisticated models required by TABATHA to estimate the size and cost of Prolog components.*

Embedding the TABATHA model within RUSSET allows TABATHA to receive information about the number and range of tasks within a proposed project plan. But RUSSET also aims to support the integration of conventional and KBS methodologies, i.e., TABATHA would receive information about both conventional and KBS tasks. The method of estimation proposed by TABATHA assumes that the products of both styles of task can be measured in the same way. However, while there have been some attempts to define (static) code metrics for KBS code (see again Chapter 4), there have been no known attempts to use this low-level data to develop the high-level SCE models or tools required here for KBSs. Can the same types of metric be applied to both conventional and knowledge based systems?

The TABATHA model requires a substantial amount of data collection in order to be validated but would generate estimates at exactly the right point: while the project plan is being developed. The EEPS model in Chapter 7 also identified the need to generate estimates before the plan has been developed. Clearly, this higher-level model could not be based on an analysis of tasks. The EEPS model suggests this point is at the earliest stages of the development process when the project manager has only the highest-level details about the project. This is where conventional SCE tools are at their most useful since they can

generate estimates based on only nominal information, using a large historical database in the same way that a project manager would rely on years of experience. The advantage of the SCE tool is that it gives a disciplined approach to estimating, such that, anyone who has the same input data will generate the same output estimate. But can such a high-level model be developed for hybrid projects which also contain KBS components?

Since the history and body of knowledge for conventional metrics is substantially greater than for KBS metrics, it would seem sensible to attempt to extend a number of conventional metrics to KBS, rather than the other way around. Because of their standing as "classic" metrics, the metrics chosen for extending to KBS are taken to be:

- non-comment, non-blank lines of code (LOC);
- Halstead's (1972, 1977) Software Science (see again §3.1.1);
- McCabe's (1976) cyclomatic complexity (see again §3.2.1);
- Henry & Kafura's (1983, 1984) data-flow fan-in/out (see again §3.2.2).

Although FPA (Albrecht & Gaffney, 1981; Symons, 1988, 1991) can also be seen as a "classic" approach to metrics, it was not investigated here because it was thought unlikely that the necessary specification/design documents would be available for a KBS project from which to investigate a count of Prolog function points. The advantage of the remaining four metrics above is that they are code-based, and so, as long as some KBS programs can be found then the simplest form of analysis can be carried out. There are a number of languages and tools for developing knowledge-based including development languages such as Lisp, OPS5 and Prolog, and expert system shells such as ProKappa and Interleaf. Because of the author's familiarity with Prolog and its popularity throughout Europe, however, Prolog is taken here to be suitable for study as a mainstream KBS development language.

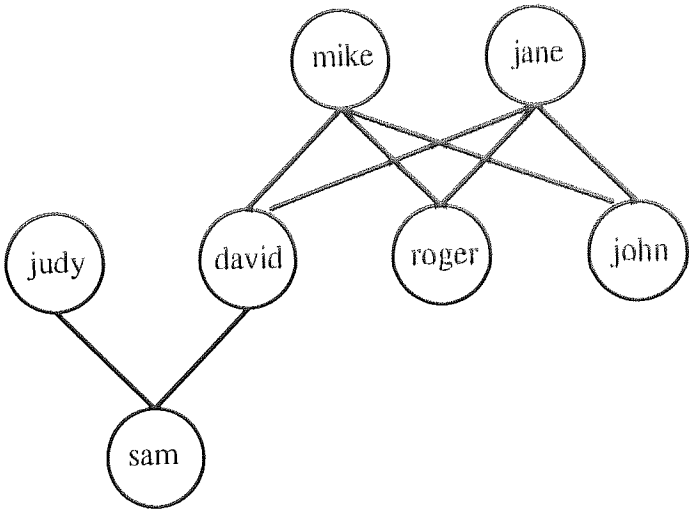
The goal here is to build models of both size and structure: what TABATHA calls $Size_{external}$ and $Size_{internal}$. A measure of $Size_{external}$ is the basis of sizing models which feeds into models of effort and duration. A measure of $Size_{internal}$ balances a notion of the amount of work required to produce a product given its structural complexity, and is also used to detect error-prone programs. Lines of code (LOC) is taken to be a baseline for both size and structure. The Halstead metric competes with LOC as a measure of $Size_{external}$, while McCabe and Henry & Kafura's metric competes with LOC as a measure of $Size_{internal}$. What needs to be answered here is:"Which is best?" and "Is the 'best' good enough?" The rest of this chapter will therefore set out to describe:

- the nature of the Prolog language;
- a re-definition of Software Science, cyclomatic complexity and data flow for Prolog;

- a tool to collect the (above) metrics;
- an analysis of the structure of LPA Prolog programs;
- deriving a simple Prolog sizing tool.

9.1 The Prolog language

Prolog (*PRO*gramming in *LOG*ic) is a declarative language which defines functions as logical statements in first-order predicate calculus form. The goal of a Prolog program is to satisfy all conditions of a relationship using mechanisms such as tree-based data structuring, pattern-matching, backtracking and recursion. For instance, consider a family of seven people, namely: Mike who is married to Jane, their children David, Roger and John, David's wife Judy and their child Sam. Each name in the family are atoms of data which can be represented as a tree such as:



Each atoms is declared in Prolog as a factual clause. A fact does not require any further processing, it is known to be true. The relationship defined between two atoms may be called **parent**, i.e., who is a parent of whom? Thus, we have eight clauses:

```
parent( mike, david).
parent( mike, roger).
parent( mike, john).
parent( jane, david).
parent( jane, roger).
parent( jane, john).
parent( david, sam).
parent( judy, sam).
```

The comma "," separates elements of the clause, while the full-stop "." denotes the end of the clause. The parentheses "(...)" group arguments together. The relationship here is that the first term in parentheses (the first argument) is the parent of the second term (second argument). The term "parent" is called a predicate or functor and is the means by which the Prolog system looks for, and finds, required pieces of information. All names assigned to predicates or variables are defined by the user. Any name may be used. Prolog simply looks for and attempts to prove the truth of any statement given to the program. For instance, if Prolog was asked to find **parent(mike, david)** it would reply **yes**, since this fact exists in the above program and so is known to be true. The query **parent(mike, david)** becomes a Prolog goal which it attempts to satisfy. The declarative nature of Prolog means additional functions can be simply added to the program without specifying the procedure by which the new clause is used by the rest of the program.

If Prolog were asked to find **married(mike, jane)** it would reply **no**, since the relationship "married" has not been defined and so Prolog cannot prove that it is true. Notice also that the first letter of the name is in lower case rather than (for pronouns) the correct upper case letter. But, if Prolog was asked **parent(Mike, David)** it would be able to match with any of the above eight clauses. This is because a variable beginning with an upper case letter is an uninstantiated variable, i.e., one that will match with any atom. Thus, **Mike** is equivalent to **X** and will match with any clause argument. Variables which do not begin with an upper case letter -such as **mike** - is called an instantiated variable and will only match with an uninstantiated variable or another atom of the same name. A "match" is where Prolog attempts to find a clause with the same pattern of predicates and arity (i.e., number of arguments). In the above eight clauses the arity is 2. Where the argument is an atom (or instantiated variable such as **mike**) the match must be exact; where the argument is a variable (or uninstantiated variable such as **X**) then it will match with any argument in the correct position.

Thus, **parent(mike, Mike)** will first return **Mike=david** because Prolog searches depth-first, breadth-first, and so the fact at the top of the eight clauses above is the first fact which satisfies the required relationship. The uninstantiated variable **Mike** matches with and is instantiated to the atom **david**. A semi-colon ";" instructs Prolog to assume **Mike=david** is false, backtrack up the data tree and search for other solutions which may satisfy the relationship. Thus, giving Prolog ";" after the first solution will then return **Mike=roger** and then **Mike=john**. Similarly, asking Prolog **parent(jane, X)** would return **X=david**. However, if Prolog were asked to find **parent(mike, jane, david)** it would reply **no**, because only relationships with two arguments (arity=2) are known. If Mike could not instantiate itself to some value then Prolog would return something like

Mike=_9083, where **_9083** is simply the variable number assigned to **Mike** as it attempts to satisfy the goal. A term beginning with an underscore "_" is also a variable.

A "rule" is a clause where conditions are attached which must be satisfied before the clause can be known to be true. For instance, the relationship "grandparent of" can be defined using the same eight clauses above with the rule:

```
grandparent_of( X, Y) :-
        parent( X, Z),
        parent( Z, Y).
```

This rule reads that X is the grandparent of Y if X is the parent of Z who is the parent of Y. The terms X, Y and Z are uninstantiated variables. The predicate **grandparent_of** is the head of this rule-clause and has two **parent** sub-clauses in the body. The operator :- is equivalent to an "if". Using the same eight facts stated earlier, if Prolog were asked **grandparent_of(X, Y)** it would reply with **X=mike** and **Y=sam** and then after an ";" it would reply **X=jane** and **Y=sam**. A second ";" would return **no** because there are no other solutions. It should be noted that the relationship **grandfather_of** or **grandmother_of** cannot be defined because the relationship "male" or "female" is not a known fact.

More complicated processing can be carried out using recursion, which is when a rule contains a sub-clause in the body which is identical to the head of the clause. For instance, the following rule is recursive:

```
member( X, [ X | L]).
member( X, [ _ | L]) :-
        member( X, L).
```

This rule reads that if **X** is a member of a list [..] which has **X** as its head, then stop; otherwise, remove the head of the list and try again. The head of a list is simply the first item. The operator "|" separates two elements, in this case, the first item from the rest of the list, **L**. The underscore operator "_" will match with any variable and is a "don't care what it looks like" match. If the list **L** no longer has a head which can be removed, i.e., it is exhausted, then the processing stops and the **member** relation fails. For instance, if Prolog was asked **member(mike, [david, roger, john, mike, jane])** after three cycles around the second recursive rule, Prolog would match on the fourth attempt on the first clause with **member(mike, [mike | jane])**. Prolog would then backtrack through the cycles and return a **yes**. Similarly, if Prolog was asked **member(X, [david, roger, john, david, jane])** five solutions could be found for **X**.

In summary, therefore, the power of Prolog rests in its ability to declare functions that can "run" backwards or forwards such as **parent(mike, X)** or **parent(X, david)**, the use of automatic backtracking which allows a search for more than one solution, and in its use of recursion to elegantly define an infinite number of the same processes. There is more to it than this, of course, but space does not allow a more detailed description of the language. More complete descriptions of Prolog exist elsewhere (e.g., Clocksin & Mellish, 1981; Bratko, 1986; Dodd, 1990). The question, however, is whether such a language is so different from conventional languages that no metrics can be extended to capture its inherent properties.

9.2 Defining Prolog metrics

The "classic" metrics which are to be extended to Prolog must be described in such a way that the elements which need to be counted are close to elements of the Prolog language. The metrics to be defined are:

- lines of code (LOC);
- Software Science;
- cyclomatic complexity;
- data flow fan-in/out.

Lines of code (LOC) is primarily a size metric which can be used as a measure of structure on the basis that the longer the program the more likely it is that an error has been introduced. The lines of code definition used here is the common non-comment, non-blank lines definition and LOC is included simply to see if such a simple metric can out-perform the other metrics. However, there is debate even for conventional languages of how to define "operators" and "operands", while imposing a clearly procedural notion of control flow or data flow to a declarative language would seem to stretch the notion of what these metrics are meant to be measuring. However, if such metrics can be extended to Prolog, then it must be on the assumption that Prolog is just another language and defining what is meant by an "operator" is the same as, say, Pascal or COBOL. After all, computer languages have the same implementation at the machine code level. The following definitions of the "classic" metrics are proposed on this basis.

9.2.1 *Software Science for Prolog*

The need here is to define a counting strategy whereby counts of operators and operands can be defined for any Prolog program. The calculation of program length remains:

$$N\hat{a}t = \eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2$$

Following the discussion given in §3.2.1, operands seem to be the easiest to define and are taken to be any instantiated or uninstantiated variable. Thus, numbers, instantiated and uninstantiated variables, and the underscore (null) variable are all counted as operands. The in-built operators **true** and **fail** are also taken to be operands on the grounds that they are values which Prolog takes to satisfy or fail a clause. Therefore, although a Prolog programmer does not need to define the value of **true** or **fail**, it acts in the same way as **X=10** is either true or false. However, this definition cannot guarantee to count only operands since predicates - which must be taken to be operators - can appear to be operands. For instance, consider the clause:

```
grandparent_of( mike, Y) :-
        parent( X, Z),
        parent( Z, Y).
```

It might initially be assumed that along with **X**, **Y** and **Z**, **mike** is an operand. However, if the next clause read

```
mike :-
        not parent( jane, _).
```

then it would be clear that **mike** was a predicate with no arguments that ruled out **jane** the variable from being detected as a grandparent. This is permissible in Prolog and is seen as one of its programming strengths, but makes it impossible to be 100% certain of correctly classifying **mike** unless the clause **mike** was in the same program. Where systems are broken down into modules it cannot be guaranteed that the use and the definition are in the same program. One solution would be to classify all atoms as potential operands and confirm the classification after analysing all the other programs and finding no other definition of the atom **mike**. This would greatly increase the cost of counting operands, however, and was not considered a viable solution to the problem. The potential of misclassifying a predicate as an operand remains in this counting strategy.

Figure 9.1 : Example Prolog program (in Edinburgh syntax)

```
process_term( ?-( Term), _, '$none') :-
        call( Term, '$none') -> true
    ;
        true.

process_term( call( Term), _, call( X, _)) :-
        X=end_of_file,
        !,
        fail
    ;
        call( Head :- Tail( '$none')).

:- call( _).

:- op( 150, xfx, ::).

call( ::( Term, _)) :-
        call2.
```

Table 9.1 : Example Halstead-table (for the program in Figure 9.1)

Operator	j	$f_{1,j}$	Operand	j	$f_{2,j}$
(..)	1	12	_	1	5
,	2	11	Term	2	4
:-	3	6	'$none'	3	3
.	4	5	true	4	2
call/1	5	4	X	5	2
process_term/3	6	2	fail	6	1
call/2	7	2	end_of_file	7	1
;	8	2	150	8	1
: :/2	9	2	call2	9	1
?-	10	1			
op/3	11	1			
->	12	1			
!	13	1			
xfx	14	1			
=	15	1			
Head/0	16	1			
Tail/1	17	1			
	η_1=17	N_1=54		η_2=9	N_2=20

$$N = N_1 + N_2 = 52 + 20 = 72$$

$$N\mathit{hat} = \eta_1.\log_2\eta_1 + \eta_2.\log_2\eta_2 = 17.\log_2 17 + 9.\log_2 9 = 98$$

Similar to Halstead's definition, an operator is taken to be all other elements of code which are not operands. Thus, all predicate names, in-built operators (such as **bagof**, mathematical notations, etc.), parentheses "(...)", square brackets "[...]", commas "," , full-stops ".", the rule operator ":-" and user-defined operators (using the function **op**) are counted as operators. The possibility of misclassifying a predicate as an operand affects the count of operators in the same way as mentioned above.

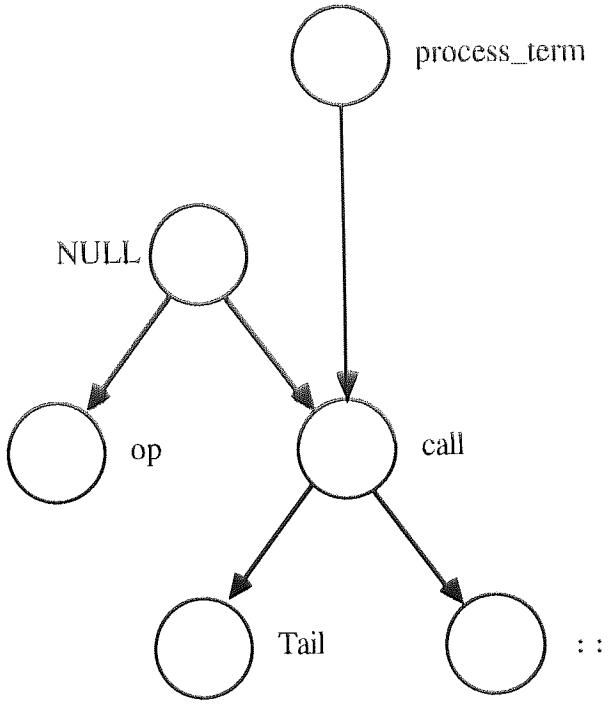
The definitions given above would allow a count of N_1 and N_2 to be produced. However, a count of η_1 and η_2 would depend on whether items with the same name are counted as the same or different, depending on their function. For instance, since variables are locally bound, should the various instances of the operand **X** in the **member** relation defined earlier be counted as occurrences of the same operand? Within the same clause clearly it should, but since there are two **member** clauses defined, what about the **X** in clause 1 and the **X** in clause 2? Technically, there is no relationship. The two **X**'s in clause 1 could be replaced by two **Got_it**'s and the function would still perform correctly. However, since Software Science is meant to be based on an understanding of the psychological make-up of a programmer, one can suggest that whenever the same name is being used (for a variable or a predicate), then there is a connection between one and the other. This suggests that there are four occurrences of the same operand **X** defined across two clauses.

This is not quite satisfactory, however, since Prolog does make a distinction between predicates which otherwise appear to be the same. Specifically, when Prolog seeks a match with a predicate the number of arguments that a predicate has defines whether the match is successful. Thus, **parent(X, Y)** will not match with **parent(X, Y, Z)**. For this reason, predicates are only counted as occurrences of the same operator if they have both the same name and the same arity. This is the only proviso attached to any count of operators. All non-predicate operators, i.e., inbuilt operators such as parentheses, commas, etc., are counted as occurrences of the same operator. Thus, five sets of parentheses would add one to η_1 and 5 to N_1. Using these definitions, an example of how they are applied for the program in Figure 9.1 is given in Table 9.1.

It should be noted that Figure 9.1 and Table 9.1 attempt to make clear how to classify some of the more complicated structures in Prolog and are not meant to act as an example of N*hat* being an accurate estimator of N. Such an example could be easily contrived. The fact that N*hat* is not a good estimate of N in this case neither confirms nor denies the validity of the counting strategy proposed. The specific

problem in Table 1 is that η_1 no longer becomes a good predictor of N_1 because there are too many operators with only one occurrence. This unbalances the program length equation, but it remains a matter of empirical investigation as to whether η_1 and η_2 are useful concepts on which to base a model of system size, or whether N*hat* can ever be a good predictor of N.

Figure 9.2 : Example cyclomatic complexity graph (for the program in Figure 9.1)



9.2.2 *Cyclomatic complexity for Prolog*

Cyclomatic complexity is a structure metric related to a measure of program control flow and is meant to relate to the error-proneness of programs. The calculation of cyclomatic complexity remains:

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

Cyclomatic complexity is defined here in terms of the calls from one predicate to another and has the following properties:

- a single node replaces all instantiations of the predicate (such that, a call by any predicate with the same name is represented as a call from a single node with that name)
- a predicate in the body of a clause is represented as an arc from the clause head node to the clause body node (where one arc represents all calls between predicates of the same name)

On the surface, cyclomatic complexity may appear to be the most inappropriate metric to extend to a declarative language such as Prolog, because Prolog does not make explicit the flow of control from one part of the program to another. However, by defining predicates as the point at which decisions are carried out, it seems reasonable to suggest that the nodes of a directed graph are the names of Prolog predicates. In Prolog terms, a predicate is the point at which the conditions of the clause are known to be true or false. The arc of the graph would then be to the predicates which form the body of the clause since Prolog must search for, and attempt to satisfy, the sub-clauses stated. Using these definitions a program can be represented as a directed flow-graph from which a measure of cyclomatic complexity can be taken (see Figure 9.2).

Unlike the definitions of predicates used for Software Science, the number or arity of clauses with the same predicate name does not seem to be important. Principally, the number of predicates does not affect the relationship defined by a set of predicates and its sub-clauses. For instance, the two clause member relation defined above could have been written as:

```
member( X, [ Y | L ]) :-
        ( X=Y ;
          member( X, L)
        ).
```

The only property which needs to be captured in either version of the **member** relation is that there is a single recursive loop being defined. The above version is a more opaque version of the first. Thus, a node within the directed graph is assigned a predicate name as soon as a predicate is defined. The arcs from node to node are from clause head to clause body. Further predicates with the same name do not add further nodes to the graph, but if the sub-clauses are different - i.e., they are directed to different predicates - then a further arc is added to the node to the newly assigned sub-clause predicate. The arity of the predicate is not considered important here because it is the name being used as a location flag that is important, not the structure.

Rules do not necessarily require a predicate, since the rule **:- get_list(X, L).** would fire in Prolog without any pre-conditions. In other words, the above rule would fire as soon as Prolog got to that part of the program. But how can this be represented in a McCabe graph? It was decided to represent null predicates by grouping them all under a node called NULL. This would get around the problem of losing the body of such rules because there is no predicate with which to denote their position in the program. Strictly speaking, however, grouping these clauses together is unsatisfactory since there is virtually no relationship between them other than their lack of a clause head.

Forcing a grouping is necessary, however, since without a strategy of grouping predicates it would be impossible to know what direction to put on the arrows of the McCabe graph. This is acutely problematic for Prolog since, as discussed earlier, it is possible to flow in a number of directions through a clause. Furthermore, it is possible to state that a relationship should not exist between one predicate and another. For example, consider the following program:

```
get_list( _, [ ]).
get_list( X, [ Y | L ]) :-
        parent( mike, Y),
        not grandparent_of( X, Y),
        get_list( X, L).
```

Is there a relationship between **get_list** and **grandparent_of** or not? The view taken here is that there is, since **get_list** cannot be satisfied unless **grandparent_of** is consulted. In other words, the existence of the relationship is not defined by the need to return a **no** rather than a **yes** but by the dependency **get_list** has on **grandparent_of**.

9.2.3 *Data-flow fan-in/out for Prolog*

The data flow metric is a design metric related to a measure of the data being communicated between modules. The calculation of C_P remains:

$$C_P = LOC * (fan\text{-}in * fan\text{-}out)^2$$

Since Prolog is a declarative language and a program file cannot easily be defined as a module or in relation to its neighbouring predicates, a data-flow "module" is

taken here to be a single clause (in that a clause handles a certain set of data conditions) and has the following properties:

- data is equated with variables contained within a clause
- data flow-in is defined as an uninstantiated variable in the head of a clause
- data flow-out is defined as an instantiated variable in the body of a clause

The same supposition used in §9.2.2 about the flow from one part of a Prolog program to another must be used here to define the way in which data flows from one set of predicates to another. An example is given in Table 9.2.

Table 9.2 : Example data-flow table (for the program in Figure 9.1)

No. of clauses	Fan-in	Fan-out
1	2	1
1	1	1
1	1	0
2	0	0

Lines of code = 14

Data-flow, C_P = LOC * (Fan-out * Fan-out)^2

= 14 * (4 * 2)^2

= 896

Of course, this would depend on a correct classification of operands. What is more difficult to justify is that a clause can be equated to a module, since the data flow metric is meant to be a design metric related to a measure of the data being communicated between (conventional) modules. The problem is not that Prolog is a declarative language, since, presumably, even Prolog needs to be modularised. Rather, a Prolog clause so neatly represents a single procedure that even if the representation is of a lower level than Henry & Kafura intended, it seems more reasonable to take into account this feature of Prolog. The assumption being made, then, is that because an individual clause handles a certain set of data conditions, "data-flow" is defined in terms of the type of data contained within the head or body of a clause.

The two types of data flow to be defined are fan-in and fan-out. Data fan-in is here defined as an uninstantiated variable in the head of a clause, while data fan-out is defined as an instantiated variable in the argument of a sub-clause. The suggestion here is that where an uninstantiated variable occurs in the head of a clause, an item of data must be passed to the clause (flow-in) when the predicate is called either from the calling predicate or from the body of the clause; while, an item of data is being passed (flow-out) where an instantiated variable is used in the arguments of a predicate call in the body of a clause. The underscore (null) operand does not count as a flow because no data is required for the clause to be satisfied.

This definition of data-flow also accepts the possibility of data flow originating from the clause body to the calling predicate by instantiating a variable in the clause head. Such a situation occurs in the head of the second **get_list** clause defined earlier, when the operand **Y** is passed by **grandparent_of** and stored in the list **L** only during backtracking after the clause has been satisfied.

This is an example of data tracking backwards within a clause. The definition is assuming, therefore, that any uninstantiated variable in the clause head must flow in from somewhere. Furthermore, operands with the same name in either the head or body are counted separately on the grounds that the data is flowing to more than one position in the clause. For instance, consider the following recursive program:

```
find_name( X, [ X | L ]) :-
        write( 'The name: '),
        write( X),
        write( 'is in the list.').
find_name( X, [ _ | L ]) :-
        find_name( X, L).
```

There are three occurrences of uninstantiated variables in the head of the first **find_name** clause and the data-flow metric would count a fan-in of three even though a call to **find_name** may appear to contain only two items of data, i.e., **find_name(tom, [tom])**. The argument here is that if the first clause matches Prolog would have to instantiate all three variables, and this is the key to the amount of data required by the clause. Whether this simplification affects the ability of data-flow to represent program structure remains to be seen.

9.3 The PSA tool for automatic data collection

The need to develop tools to collect the low-level data has long been recognised, especially within the metrics consortia sponsored by ESPRIT. Out of these projects have come a number of code-measurement tools, such as COSMOS (P2686) and the QUALMS tool from AMI (P5494). These tools apply conventional metrics to conventional languages, however, and could not be used in this study. To solve this problem, a simple Prolog measuring tool was developed which was given the name PSA (*P*rogram *S*tructure *A*nalyst).

The time-scale of this research did not permit a highly sophisticated tool to be developed. In particular, given the problems identified in the previous sections of how Prolog makes no distinction between commands and data, then the PSA tool would - ideally - take all the programs associated with a system, parse the program text and then calculate the required metrics counts. A tool which parsed Prolog code was seen as being beyond the scope of this study and so a more simplistic design was adopted. It was decided to search a Prolog program token-by-token until an operator was recognised that flagged a change in the structure of the program, and only then would the previous tokens be collected together and identified. For instance, given the program line:

member(X, [X | L]) :-

the idea would be for the PSA tool to pick up all the tokens which made up **member**, but only when the "(" operator is found would it be recognised that the tokens collected so far constitute a single item of Prolog code. PSA would then set about classifying the string comprising **m-e-m-b-e-r**. In this case, because of its relationships to the "(" operator, PSA can deduce that **member** is a predicate since no other class of Prolog atom can immediately occur before this operator.

Similarly, when PSA finds the operator "," it knows it must have passed another significant element. In this case, knowing that a predicate has been identified and the other parenthesis ")" has not yet been found, PSA will know that **X** is an operand and the first argument of the predicate **member**. Again, when PSA finds the operator "[" it knows that a second (list) argument has been found. On finding the operators "|" and then "]" PSA deduces that the second argument of **member** has been completed and that the operands **X** and **L** are contained within this argument. The operator ")" ends the number of arguments of **member**, while the operator ":-" flags that the next line contains the body of the clause **member**. Thus, the name of the predicate is deduced in retrospect, and its arity recorded with each comma outside a list between the opening and closing parentheses.

Using this method ensures that PSA only needs to know the in-built operators which separate user-defined functions (such as the names of predicates and variables). The program is not parsed, but specific operators are looked for which denote that the string of tokens just passed was either a predicate or variable. Of course, this method assumes that PSA understands all the possible relationships that an operator can imply. This is difficult for three reasons.

Firstly, the user may change the grammar of the program by defining new sets of operators. This is done using a predicate called **op** which defines the precedence and direction of the relationship (e.g., **op(900, xfy, because**) defines the operator **because** that relates x to y and has a precedence of 900). This freedom of defining operators is problematic in the same way that predicates with no arguments appear to look like instantiated variables. Without the **op** definition in the same program, it would be impossible to detect that **because** was an operator, rather than a predicate or a variable. PSA must therefore assume that where the **op** operator is used it will also be defined. This need not necessarily be the case, however.

Secondly, by taking a token-by-token approach and looking back at the string just past, there is a danger that the string can be incorrectly classified if some look-ahead is not also carried out. Specifically, the full-stop operator "." can denote seven different relationships to the string of tokens before it. These are:

1. within a comment (such as **/*This is a comment.*/**);
2. within a write statement (such as **write('Press F1 for help.')**);
3. at the end of a clause (such as **clause :- !, fail.**);
4. as a decimal point (as in **0.01**);
5. as a list head (such as **.(Head, Tail)**);
6. as part of an inbuilt operator (such as =..);
7. in a user defined operator (as in **Head ... Tail**).

The full-stop operator is unique in having this number of roles, however, and so it was possible for PSA to look-ahead only when the "." operator was found. However, this does assume that the user has not defined combinations of other operators.

Thirdly, the problem of predicates looking like instantiated variables remains. It was decided to handle this by asking the user to tell PSA if a program contained such atoms. Although accepting the vulnerability of PSA to mistakes such as these, it would not be possible to implement a proper solution without devoting a considerable amount of time to the development of a more complete measurement tool. Clearly, even manual counts will be

error-prone and the tool was needed only to be at least as accurate as a manual count. A series of Prolog programs were collected and PSA and manual counts recorded. Where differences occurred it was found to be the fault of the manual count rather than the PSA tool.

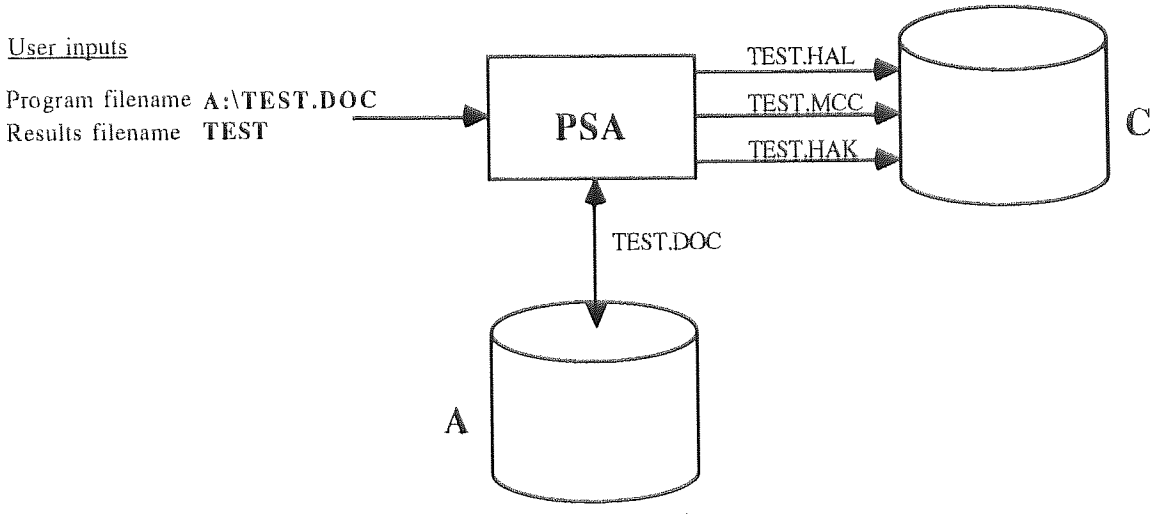
The Program Structure Analyst (PSA) tool was produced using Prolog-2. The program works by reading programs, one ASCII character at a time using **get0**, and storing the characters as a list called **stringsofar**. If one of a number of special ASCII codes is detected which denotes the end of an interesting sequence of characters (such as "(" denoting the end of a functor name and the beginning of one or more arguments), the program analyses the contents of **stringsofar**.

The most basic elements of a Prolog program are taken to be the Halstead operator and operand count. Thus, the results are asserted into the database with the label **pred**, **operator** or **operand**. The next sequence of characters is then read. At the end of the analysis, all the **pred**s, **operator**s and **operand**s are swept up and listed in a table along with the results of applying the Halstead length equation to the data. The table is saved in a separate file under a name given by the user. The predicates in **pred** form the basis of the McCabe count, where each predicate is a node in the directed graph and the number of sub-clauses denote the arc from one predicate to another. The operands noted in the head or body of a clause then form the basis of the data-flow count. PSA stores the results of the analysis in results files with the following extensions:

- .HAL (program length count);
- .MCC (cyclomatic complexity count);
- .HAK (data-flow fan-in/out).

The prefix filename is provided by the user at the same time as PSA asks for the name (and location) of the program to be analysed. PSA has the facility to analyse one program at a time or a series of programs defined within a user-created Prolog file. The results files can be printed and viewed normally through DOS. Overall, the PSA tool reads but does not alter the code of the program being analysed in any way. Using **test.doc** stored on drive **a:** as an example filename with PSA stored on the **c:** drive, the functionality of PSA can be summarised in Figure 9.3. The user is asked to define any predicate with no arguments, i.e., those that may look like instantiated variables. This can be done manually when analysing one program at a time, or by specifying all the relevant information in a file which PSA then reads as each file is located and analysed.

Figure 9.3 : Representation of the PSA tool



The original motivation behind developing the PSA tool was so that a number of commercial companies could be contacted and, instead of having to travel to these sites and laboriously collect data, the PSA tool could be sent directly to interested companies and PSA could be run overnight. This held out the possibility of collecting a large amount of data in a short space of time. However, even with adverts posted in two international Prolog newsletters (Prolog-2 and LPA Prolog), only three companies expressed any interest. The PSA tool was sent to these companies but no results were returned. A number of further attempts were made to re-contact these companies, without success, and the scheme was eventually abandoned.

A fourth, local, company was then contacted and a number of programs were made available for study. These comprised a set of 84 commercially-developed LPA Prolog programs which formed a substantial part of a commercial tool. Of these, 80 were successfully analysed by PSA. The remaining four programs were found to have Prolog structures which had not been programmed into PSA. These 80 programs were principally developed by one experienced programmer, and so there was no need to investigate relevant adjustment factors since programs developed by the same programmer in the same development environment meant any adjustment factors could be set to 1.0 for each program.

With the adjustment factors thus simplified out, the 80 programs were seen as ideal candidates for study and are the programs used to validate the "classic" metrics defined here for Prolog. A summary of the Prolog metric data collected is given in Appendix B.

9.4 Applying structure metrics to LPA Prolog

The aim of the structural analysis here is to deduce a threshold, T, above which Prolog programs tend to be error-prone and to choose the metric which follows subjective ratings of program complexity. For instance, McCabe suggests that programs tend to be error-prone when v(G)≥10. The relationship between "error-proneness" and "subjective complexity" is one assumed by most structure metrics, but not one which has been adequately established. The structure metrics being used are:

- lines of code (defined as non-comment, non-blank lines of code);
- cyclomatic complexity (defined in §9.2.2);
- data-flow fan-in/out (defined in §9.2.3).

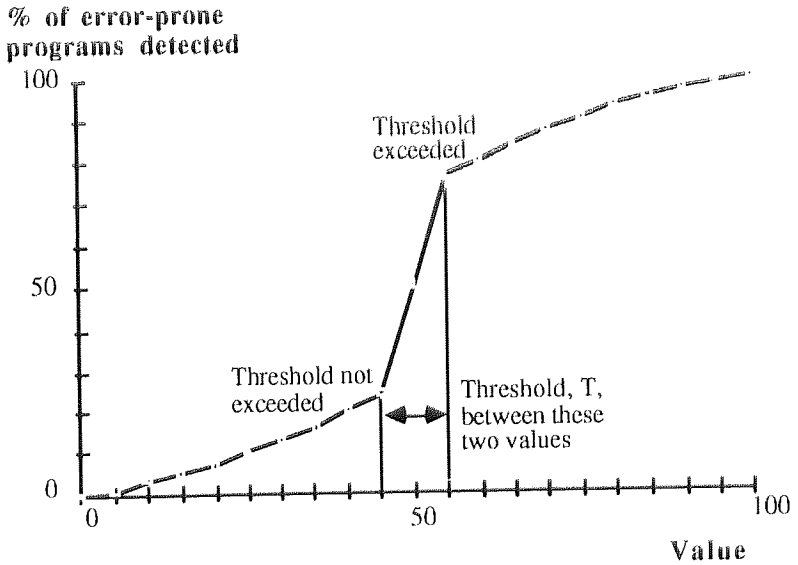
Table 9.3 : Cross-correlation of Prolog structure metrics

	Lines of code	v(G)	C_P	Under-stand	Debug	Test
Lines of code	------	0.910	0.914	0.750	0.719	0.675
v(G)	0.910	-----	0.851	0.719	0.690	0.656
C_P	0.914	0.851	-----	0.749	0.658	0.607
Understand	0.750	0.719	0.749	-----	0.900	0.840
Debug	0.719	0.690	0.658	0.900	-----	0.947
Test	0.675	0.656	0.607	0.840	0.947	-----

Regression was not thought to be a good technique to be used here since neither the metrics nor the rating data correlate well with the number of errors (see Table 9.3). A regression model which attempted to relate two uncorrelated factors would produce a "best fit" that passed through a widely scattered plot of data points. Instead, it was decided to test the relative accuracy of the three structure metrics lines of code (LOC), cyclomatic complexity and data flow. Of the 80 programs analysed, 46 had a total of 115 post-release errors (i.e., 57.5% of the programs had errors). Such errors record faults found by users and β-testers after the product had been released. No reports detailing coding or testing errors were available. Since the task of testing is assigned by program/module, the analysis will focus on detecting an error-prone program regardless of how many errors a particular program might have.

A range of data is available to help calculate a threshold value, T. Firstly, ratings out of 10 (1=simple, 10=difficult) were given for each program and obtained from a programmer who was familiar with both the system and implementation language (LPA Prolog). If the structure metrics applied do indeed measure "structure", then there should be a good association between the ratings given to a program and its metric measure. Three sets of ratings were given by the programmer responsible for the development of the system on the basis of the difficulty a Prolog-literate person would have in taking over the maintenance of the system and (1) understanding, (2) debugging, and (3) testing each program file. The raw data collected using the PSA tool is summarised in Appendix B, Table B.1.

Figure 9.4 : Representation of an ideal structure metric



9.4.1 *Comparing three structure metrics*

The question now is how well each of the three structure metrics perform in terms of correctly identifying those programs which contain at least one post-release error. Ideally, a sharp change would be detected about the point at which structured programs became unstructured and, hence, complex and error-prone. A good structure metric, then, would have a narrow threshold below which few (if any) programs have errors and above which most (or all) programs have errors. The behaviour of such a metric is represented in Figure 9.4. In order to define a good model, however, the following questions need to be answered:

- how is the adequacy of a threshold to be defined?
- how are the thresholds to be calculated?
- how can the accuracy of the metrics be compared against the rating data?

The results of the survey in Chapter 6 found that project managers tended to regard the average threshold for an "adequate" estimate at ±18.8%. A reasonable criterion for structure thresholds would therefore seem to be the highest value of T at which programs exceeding T contain 80% of the recorded errors, $S_T(T) \geq 0.80$. This replaces the Pred(0.20) criterion for size or effort models. Along with this condition, however, it also seems necessary to maximise the proportion of programs which exceed T and have recorded errors ($Prop_{Right}$=max) and minimise the proportion of programs which have recorded errors but do not exceed T ($Prop_{Wrong}$=min). These two measures replace MMRE. It would <u>not</u> seem reasonable to apply constraints to $Prop_{Right}$ and $Prop_{Wrong}$ - such as, $Prop_{Right} \geq 0.80$ or $Prop_{Wrong} \leq 0.20$ - since this would depend on the efficacy of the error detection and reporting procedure and not a property of the structure itself. These conditions are therefore defined as:

$S_T(T)$ = the proportion of programs with errors that exceed T

$$= \frac{E}{N_E}$$

$Prop_{Right}$ = the proportion of programs which exceed T that have errors

$$= \frac{E}{N_T}$$

$Prop_{Wrong}$ = the proportion of programs which do not exceed T that have errors

$$= \frac{N_E - E}{N_{total} - N_T}$$

where,

E = number of programs above T which have errors (max=46)

N_E = number of programs with errors (total=46)

N_T = number of programs above T (max=80)

N_{total} = number of programs (total=80)

The value of $S_T(T)$ will vary between 0.00 and 1.00. In comparing the performance of each of the three structure metrics, therefore, the threshold for each is taken be the highest value of T for which the following are true:

$$\mathbf{S_T(T) \geq 0.80} \text{ (and where } \mathbf{Prop_{Right}=max} \text{ and } \mathbf{Prop_{Wrong}=min})$$

Figure 9.5 : Comparison of stepped $Prop_{Right}$ values for LOC, v(G) and C_P

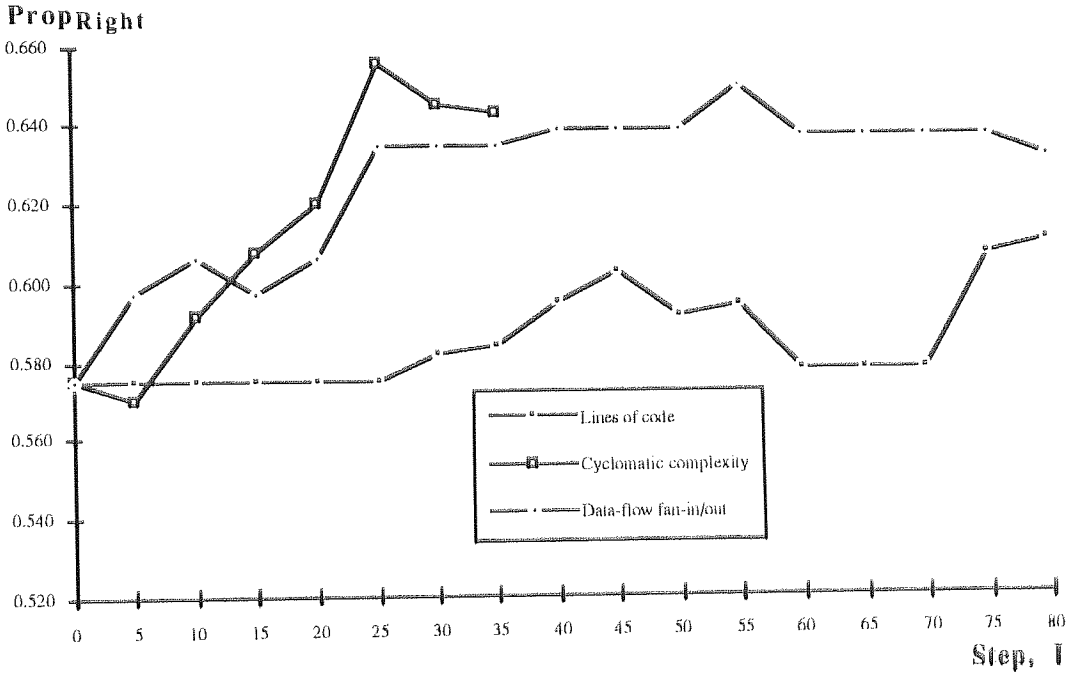
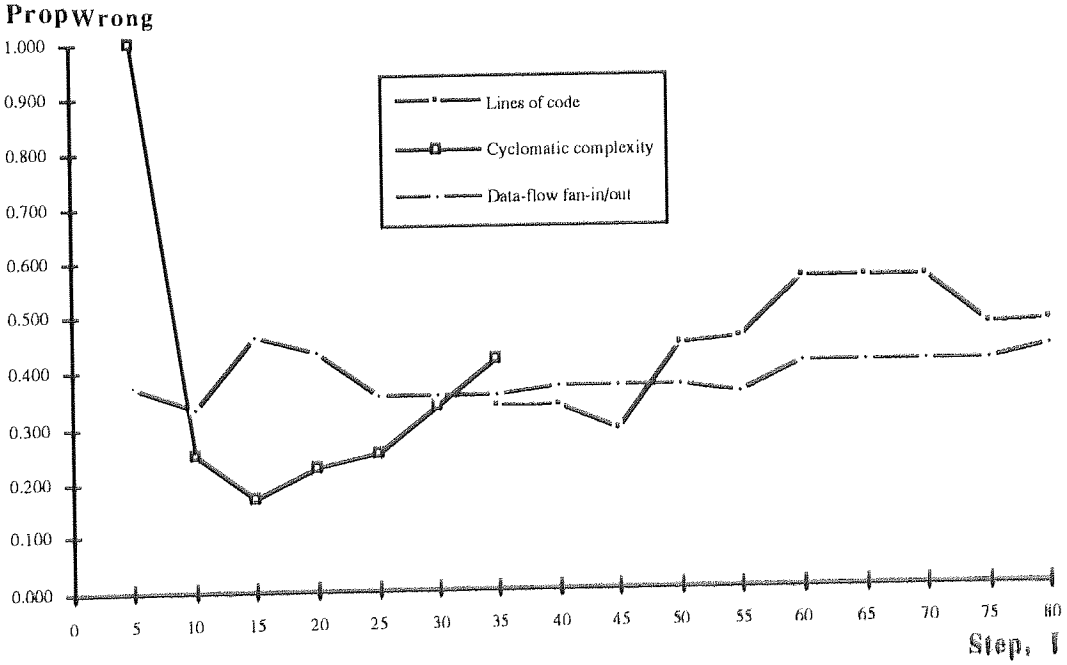


Figure 9.6 : Comparison of stepped $Prop_{Wrong}$ values for LOC, v(G) and C_P



The thresholds are calculated in a step-wise fashion, beginning with a suitably low value of T for which $S_T(T)>0.80$ and increasing in small steps until $S_T(T)<0.80$. The previous value then becomes the threshold value. The method used is as follows:

1. Choose an appropriate value for each step-wise increase, I, (in this case, I=5 is taken for all three metrics).
2. Let the first value of T = 0 (in order to show the rate of change of the values of $S_T(T)$, $Prop_{Right}$ and $Prop_{Wrong}$).
3. Calculate the values of N_T and E for the sample, N.
4. For each step, calculate $S_T(T)$, $Prop_{Right}$ and $Prop_{Wrong}$.
5. If $S_T(T)>0.80$, let I=I+5 and repeat step 3, otherwise let the threshold value T=I-5.

Appendix C, Table C.1 summarises the respective lines of code (LOC), cyclomatic complexity (v(G)) and data fan-in/out (C_P) samples. The $S_T(T)$, $Prop_{Right}$ and $Prop_{Wrong}$ values are listed in Appendix C, Tables C.2 to C.4 and represented in Figures 9.5 and 9.6. Only those steps where there is a change in N_T are listed in Tables C.2-C.4, while the analysis ends when less than 80% of those programs with errors are above the threshold. As can be seen the best metric appears to be cyclomatic complexity, v(G), on the grounds that it:

- requires the fewest number of programs to be tested in order to capture 80% of the programs with errors (56, as opposed to 57 or 59);
- has the highest $Prop_{Right}$ and lowest $Prop_{Wrong}$ at the required 80% cut-off point (0.643 and 0.417, respectively);
- achieves the highest $Prop_{Right}$ of any metric (0.656 when T=25);
- achieves the lowest $Prop_{Wrong}$ of any metric (0.167 when T=15).

The actual threshold chosen for v(G), however, would depend on the limits placed on the number of error-prone programs which are allowed to be released, and the number of programs which can be reasonably tested during the final development phase.

9.4.2 *Finding an optimal combination of structure metrics*

The previous analysis has shown that v(G) is the best of the three structure metrics for detecting error-prone programs. This section attempts to deduce the performance

of the metrics in combination and identify the best combination and threshold values. For each metric, there are three threshold values, T, that could be chosen:

- the point at which more than 80% of the programs with errors exceed T;
- the point at which $Prop_{Right}$ is at a maximum;
- the point at which $Prop_{Wrong}$ is at a minimum.

The performance of the metrics at these points can be deduced from Tables C.2-C.4. However, since LOC, v(G) and C_P do not correlate perfectly against one another, it is possible that they are measuring different aspects of a program's complexity, and so, may perform better when combined. The metrics can be combined in one of a number of ways, such that:

- a program is error-prone if it exceeds T for metric 1 **and/or** metric 2 **and/or** metric3.

Altogether, therefore, there are 14 combinations. Appendix C, Table C.5 shows these 14 combinations when the threshold value, T, for each metric is taken to be when at least 80% of the error-prone programs exceed the value of T. Appendix C, Table C.6 shows the same 14 combinations when the threshold is taken to be when $Prop_{Right}$ is maximised (i.e., the proportion of programs which exceed T and have errors is at a maximum). Appendix C, Table C.7 shows the same 14 combinations when the threshold is taken to be when $Prop_{Wrong}$ is minimised (i.e., when the proportion of missed error-prone programs is at its non-zero minimum).

As can be seen, the combination of metrics in Tables C.5-C.7 consistently show better results (in terms of $Prop_{Right}$ and $Prop_{Wrong}$) than the metrics used in isolation. This not only suggests a better use of the complexity metrics but also that program complexity has more features than are captured by any single metric. The best combination of the LOC, v(G) and C_P metrics can be deduced by applying the following criteria to the performance of any set used:

- more than 80% of the programs with errors (i.e., ≥37 programs) must be captured by the value of T (this criterion is assumed to be a minimum level of accuracy for a metric which proposes to measure error-proneness);
- $Prop_{Right}$ must be maximised (such that the proportion of programs which exceed T and have errors is also maximised);
- $Prop_{Wrong}$ must be minimised (such that the proportion of programs which do not exceed T but have errors is also minimised);

- less than 80% of the programs (i.e., ≤63 programs) must exceed the threshold, T (since the fewer programs triggered, the less effort required to carry out the testing).

These criteria are applied to the previous seven tables of threshold values in order to find the best table entry satisfying all four points (except for Appendix C, Table C.7, which failed the first criterion in all cases). As can be seen (see Table 9.4) the combination of metrics from Appendix C, Table C.6 proves to have the highest $Prop_{Right}$ and the lowest $Prop_{Wrong}$. The best use of the complexity metrics studied here can therefore be deduced to be using cyclomatic complexity, v(G), and data fan-in/out, C_P. In other words, a program is most likely to be error-prone when it exceeds both structure metric thresholds.

Table 9.4 : Best performing combinations of (one or more) metrics (from Appendix C, Tables C.3 to C.8)

Table	Combination	N_T	E	$S_T(T)$	Prop	
					Right	Wrong
C.2	LOC=75	61	37	0.804	0.607	0.474
C.3	v(G)=30	62	40	0.870	0.645	0.333
C.4	Cp=55	60	39	0.848	0.650	0.350
C.5	v(G)=30 & Cp=75	55	37	0.804	0.673	0.360
C.6	v(G)=25 & Cp=55	57	39	0.848	0.684	0.304
C.7	-----	-----	-----	-----	-----	-----

This result suggests that the complexity of the control and data structures are the key to a program's overall error-proneness and not simply a matter of program length. It also suggests that the most effective threshold values are taken from maximising $Prop_{Right}$ rather than from minimising $Prop_{Wrong}$ (which catch more errors but demand most of the programs to be tested). Testing the sensitivity of the result by trying thresholds just below and just above those given (see Appendix C, Table C.8), shows that v(G) could range from T=25 to T=30 without any loss of performance, while the value of C_P needed to be precisely T=55.

In summary, therefore, the search for the highest value of T at which $S_T(T)>0.80$ suggests that the best combination of metrics with which to detect programs that are likely to have documented post-release errors is when:

$$v(G)\geq 30 \quad \underline{\text{and}} \quad Cp\geq 55$$

It should be noted that six of the largest programs exceed the threshold for all three structure metrics and yet have no documented post-release errors (program numbers 1, 35, 61, 62, 77 and 80; see Appendix B, Table B.1). This appears to be because all six programs are core to the functionality of the system and so were extensively tested before release. Hence, it could be argued that the metrics defined here have only captured programs which have been inadequately tested rather than which have overly-complex structures that make them error-prone.

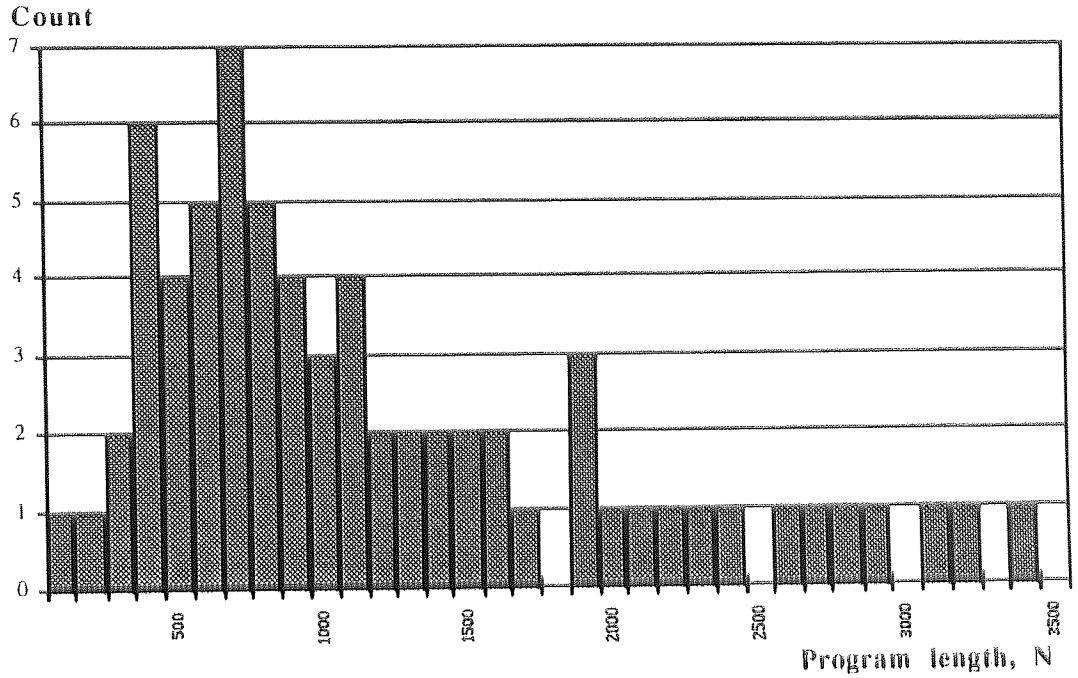
By using documented post-release errors rather than errors detected during testing, this criticism is essentially correct. This point is further reinforced by noting that subjective ratings of the difficulty to understand, test or debug a program correlate better with each other (r=0.840-0.947) than with the metrics LOC, v(G) or C_P (r=0.607-0.750)(see again Table 9.3). Since results of testing were not available, this criticism cannot be overcome.

On the other hand, it could also be argued that since fixing a "bug" during testing if cheaper than post-release "fixes" (e.g., Boehm, 1981), the errors detected within the sample of 80 programs by $v(G)\geq 30$ and $C_P\geq 55$ are the sort of expensive repairs which the standard testing procedure has clearly not captured. Furthermore, since it remains doubtful that structure metrics do measure something called "psychological complexity", there seems little need to follow Kitchenham *et al* (1990) and attempt to correlate the metric to subjective ratings. Rather, what is needed is an analysis of what it is about v(G)30 and $C_P\geq 55$ that relates to the likelihood of a program having a documented, post release error. Answering this question is an area of further research beyond the scope of this current project.

9.5 Applying size metrics to LPA Prolog

The data was collected using PSA and ranged in values of N from 92 to 3391. The sample is skewed with a long upper tail (see Figure 9.7), which is typical of metric data samples. The aim of the analysis here was to deduce a model of Prolog program size which has as input some variable related to the number of unique predicates, P, or the sum of predicates, P_T, and which correlates well with the overall size of the program. The usefulness of a

Figure 9.7 : N Histogram for the LPA Prolog sample



model based on P is that it is a feature of a Prolog program which a Prolog programmer is most likely to have some intuitive feel for. Listing the number of predicates contained within a program is effectively to list the functionality required. Clearly, P would be easier to guess than P_T. With a good model of program size in place, it would then be possible to hypothesise an effort model whereby:

Effort α Program Size

In this case, the final size of the program is taken to be the sum of Prolog operators and operands which was defined by Halstead (1972, 1977) as:

$$N = \eta_1.\log_2\eta_1 + \eta_2.\log_2\eta_2$$

where η_1=number of unique operators and η_2=number of unique operands. Prolog programs are similarly defined in terms of predicates, P, non-predicate operators, O, instantiated variables, I, uninstantiated variables, U and their totals (P_T, O_T, I_T and U_T, respectively). In order to define a good model of Prolog program size the following questions need to be answered:

- what is a definition for an "accurate" model?
- is the $\eta.\log_2\eta$ model a good representation?
- can a model be defined which has P (or P_T) as its key input element?

9.5.1 *Lines of code or program length?*

The measure of accuracy based on Conte *et al* (1986) and refined in Chapter 6 will also be used here. A model of size will be deemed to be accurate, therefore, if:

$$\text{Pred}(0.20) \geq 0.80 \quad \underline{\text{and}} \quad \text{MMRE} \leq 0.20$$

A number of variations of the Halstead model - using the counting strategy defined in §9.2 - was applied to the 80 LPA Prolog programs (see Appendix D, Tables D.1 and D.2). The different versions were applied because high correlation but poor accuracy is typical for Halstead's model (e.g., Conte *et al*, 1986). This raises the question of whether the $\eta.\log_2\eta$ model is a good representation of program size especially since it is well-known that there is no strict basis for Halstead proposing a logarithmic relationship between program length and the number of operators and operands (e.g., Fitzsimmons & Love, 1978; Lister, 1982).

It has also been found that simple linear models (such as $10\eta_2$) have outperformed the length equation in terms of correlations with N (Elshoff, 1976). The fact that the logarithmic N-model is of a dubious form is borne out by the fact that a linear model using η_1 and η_2 has coefficients on a par with the $\eta.\log_2\eta$ model (see again Appendix D, Table D.1). This result suggests there are four competing forms, namely (in the same order as that listed in Table D.1):

1. $f(\eta_1) + f(\eta_2)$
2. $a.(f(\eta_1) + f(\eta_2)) + c$
3. $a.f(\eta_1) + b.f(\eta_2) + c$
4. $a.\eta_1 + b.\eta_2 + c$

where,

$f(\eta_1)$ = $\eta_1.\log_2\eta_1$
$f(\eta_2)$ = $\eta_2.\log_2\eta_2$
a = regression parameter
b = regression parameter
c = regression constant

Figure 9.8 : N versus N*hat* ($=\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2$)

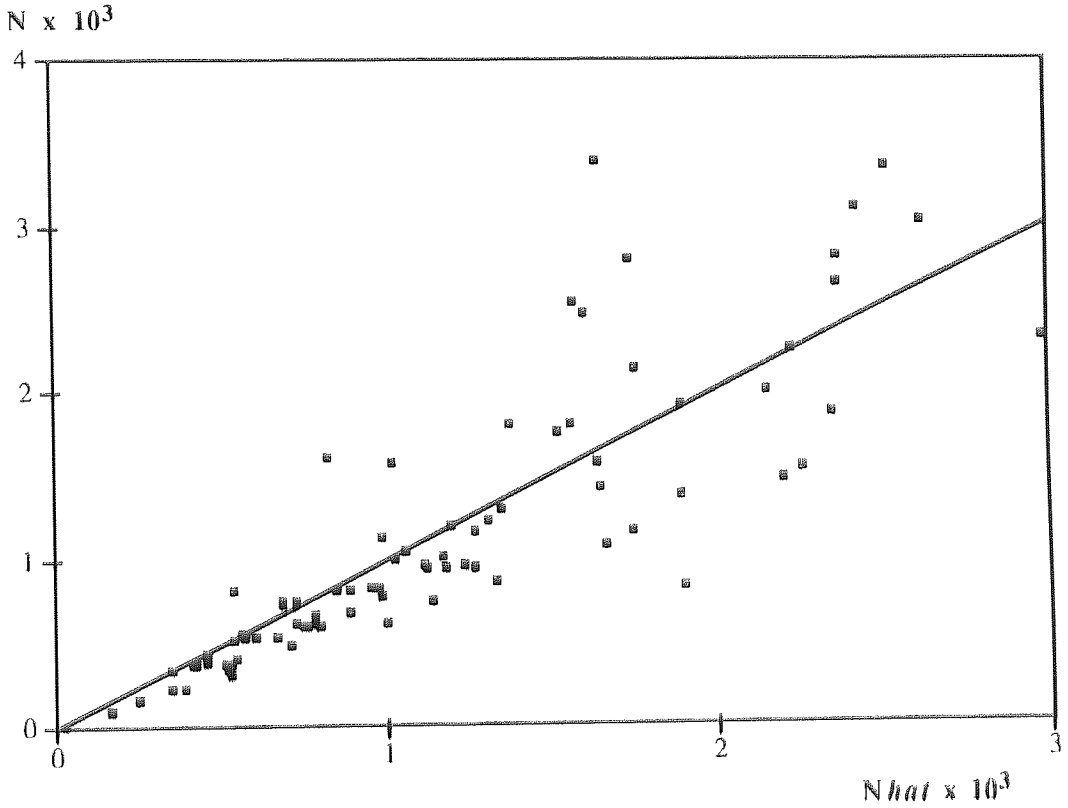


Table 9.5 : Best combinations of logarithmic and linear models of Prolog size (from Appendix D, Tables D.1 to D.4)

Model of	Model=	Coeff. Cor.	Coeff. Det.	MMRE	Pred (0.20)
N*hat*	$\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2$	0.854	0.730	0.25	0.49
N*hat*	$1.05(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)-74.80$	0.856	0.732	0.22	0.59
N*hat*	$1.42(\eta_1.\log_2\eta_1)+0.89(\eta_2.\log_2\eta_2)-92.82$	0.857	0.734	0.21	0.59
N*hat*	$10.53\eta_1+7.55\eta_2-356.22$	0.857	0.735	0.24	0.54
LOC*hat*	$0.13(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)-5.92$	0.930	0.866	0.20	0.66
LOC*hat*	$0.13(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)$	0.930	0.865	0.20	0.66
LOC*hat*	$0.22(\eta_1.\log_2\eta_1)+0.1(\eta_2.\log_2\eta_2)-10.20$	0.936	0.876	0.20	0.66
LOC*hat*	$0.21(\eta_1.\log_2\eta_1)+0.1(\eta_2.\log_2\eta_2)$	0.934	0.873	0.22	0.68
LOC*hat*	$1.68\eta_1+0.80\eta_2-44.92$	0.935	0.874	0.21	0.68
LOC*hat*	$3.09\eta_1-43.43$	0.907	0.822	0.24	0.58
LOC*hat*	$1.54\eta_2-21.50$	0.904	0.818	0.23	0.56

Form 1 is the original Halstead program length equation which only appears to model estimates of N well for small values (see Figure 9.8). The other three forms are deduced by regression and in each case better model performance is achieved (see again Appendix D, Table D.2). More worrying, however, is that η_1 and η_2 used in a regression model for lines of code significantly outperforms the length model. The LOC models are given in Appendix D, Tables D.3-D.4, with a summary of the best N and LOC models given in Table 9.5.

The conclusion seems to be, therefore, that the Halstead logarithmic model is a poor representation of program size, although linear models using η_1 and η_2 as counts of program elements produce results close to the standards of accuracy required by this study. The approach taken here, then, will be to use the concept of operators and operands to decompose a Prolog program into countable elements from which a model can be built which relates to some "size" measure of the program. This size measure can be either the length equation, N, or a model which relates operators and operands to lines of code.

9.5.2 *Deducing a linear size model*

Since models of N and LOC have been found to be relatively accurate using η_1 and η_2, the number of predicates used within a Prolog program can be introduced to the size model by separating the counts of η_1 and η_2 as follows:

P = number of unique predicate names
P_T = total number of predicates
O = number of unique non-predicate operators
O_T = total number of non-predicate operators
I = number of unique instantiated variables
I_T = total number of instantiated variables
U = number of unique uninstantiated variables
U_T = total number of uninstantiated variables
$\eta_1 = P + O$
$\eta_2 = I + U$
$N_1 = P_T + O_T$
$N_2 = I_T + U_T$

The count of η_1 is now separated into a count of predicates, P, with all other operators defined as a count of non-predicate operators, O. The two types of variable defined within Prolog are also used to separate η_2 into counts of instantiated

variables, I (numbers, user-defined atoms and the in-built atoms **true** and **fail**), and uninstantiated variables, U. The count of N remains $N_1 + N_2$.

If P, O, I and U are to be seen as sensible Prolog surrogates for the η_1 and η_2 terms, then the Prolog model must correlate at least as well as a model using η_1 or η_2. Otherwise there would be little point in replacing η_1 and η_2 by P, O, I and U. Three further conditions can be applied in order to reduce the search space of regression models:

1. The most useful model will have the number of unique predicates, P, as the primary input parameter. Models using P_T would be more difficult to use, but are not ruled out in case models using P might fail to reach the necessary levels of accuracy.
2. The values for O and U are not sensible input parameters since these are typically counts of the number of parentheses, use of X's, etc., etc. Models using O_T and I_T will not be ruled out at this stage, however, for the same reasons as P_T in '1'.
3. Only models where the coefficient of correlation is greater than 0.936 will be analysed further. This coefficient is based on the best size model using η_1 and η_2. If a model which replaces η_1 and η_2 with P, P_T, O_T, etc., is meant to be a good representation, then it must perform better than the model using η_1 and η_2.

Using these conditions, a number of regression models were deduced for N and LOC (see Appendix D, Tables D.5 and D.6, respectively). Only when a model exceeded the 0.936 coefficient value and contained no parameters with a negative sign was the same model deduced without a constant (so the line goes through the origin). No negative signs are allowed in the model on the grounds that all the parameters are expected to contribute to program size, and thus, a negative sign suggests the form of the model is wrong (Miyazaki *et al*, 1992).

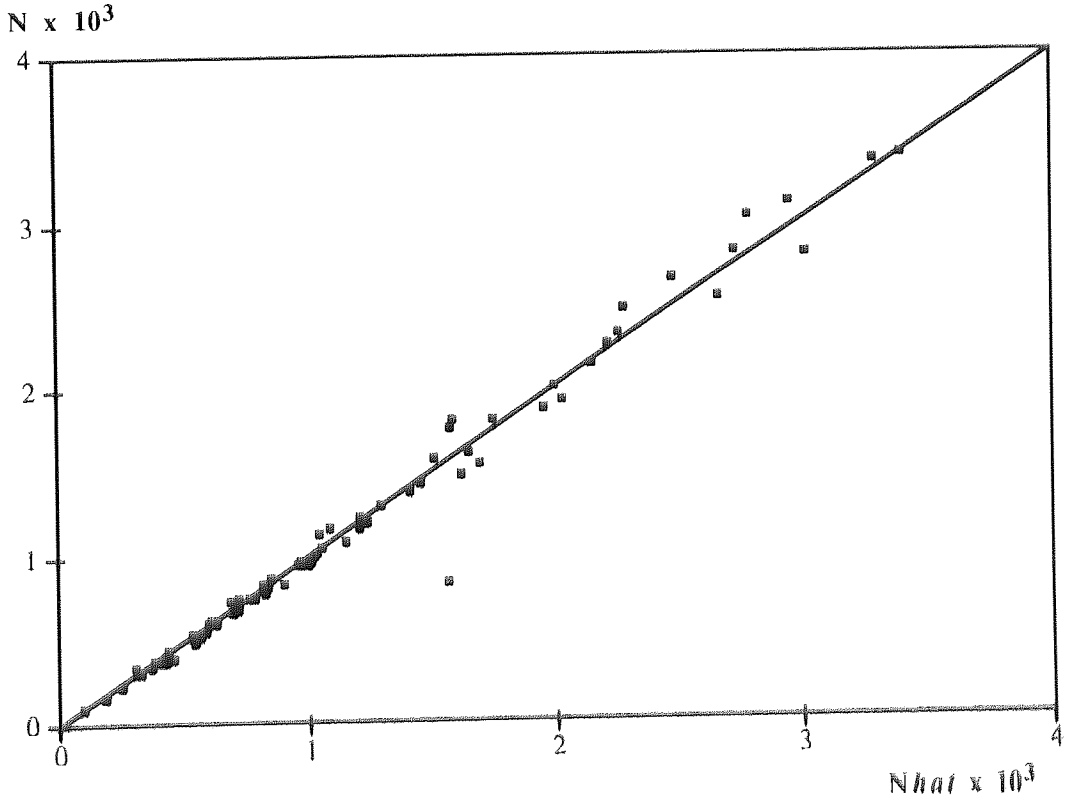
Given the number of models produced it became necessary to increase the third selection criterion such that only those models whose coefficient of determination exceeded 0.900 were selected for further analysis. The secondary analysis tested the performance of the models using the conditions MMRE≤0.20 and Pred(0.20)≥0.80. Values for Pred(0.25) and Pred(0.30) are given for reference. As can be seen (see Table 9.6), out of the six models the best are produced for N. In particular, if values for P, I_T and U_T were available, then a remarkably accurate model would be possible (see Figure 9.9). The accuracy of this model means that models using P_T instead of

P are not required and so no such analysis is presented here. The values of I_T and U_T relate to the amount of data variables used in a program and whether these values can be successfully deduced as input parameters to the program size model is investigated in the following section.

Table 9.6 : Best combinations of models for Prolog size (from Appendix D, Tables D.5 to D.6)

Model of	Model=	MMRE	Pred		
			0.20	0.25	0.30
N*hat*	$0.30P+1.27O_T+0.80I_T+0.84U_T$	0.03	0.98	0.98	0.98
N*hat*	$3.14P+2.59I_T+2.52U_T$	0.06	0.99	0.99	0.99
N*hat*	$9.18P+2.68U_T$	0.15	0.78	0.84	0.91
LOC*hat*	$1.32P+0.10O_T+0.20I_T$	0.17	0.81	0.86	0.88
LOC*hat*	$1.58P+0.34I_T+0.12U_T$	0.17	0.78	0.84	0.86
LOC*hat*	$1.41P+0.13O_T$	0.18	0.78	0.83	0.88

Figure 9.9 : N versus N*hat*($=3.14P+2.59I_T+2.52U_T$)



9.5.3 *A paper-based Prolog sizing tool*

The problem with the model deduced in the previous section is that ideally the cost of a project needs to be determined early, and it is difficult to give reasonable input values for P, I_T or U_T. The best solution to this problem would seem to be to produce a number of banded values of P, I_T and U_T such that as more information is gained a more accurate estimate of the system/program size can be produced.

A set of "bands" is simply produced by choosing a suitable number (say, 5 or 7), ordering the data points in terms of ascending values and dividing the data points into this number of groups. Since there are 80 data points, a model with 5 bands would have 16 data points in each group. Since 7 bands do not divide exactly, it was decided to have 12 data points in the first six groups and the remaining eight in the seventh group. This meant the relatively unrepresentative upper values of P, I_T or U_T would be grouped into a class of similar values. The value assigned to each band is then the average value of the set of data points contained within the group.

The upper and lower thresholds of a band are defined as the limit of values which are meant to be represented by a given band. This is calculated as the average of the highest value in one group and lowest value in the preceding group. For instance, if the first band of P ended with P=24 and the second band of P started with P=26 (i.e., the values of the 16th and 17th data points), then the upper threshold of band 1 would be defined as P<25, while the lower threshold of band 2 would be P≥25. The upper threshold of band 2 would be calculated similarly. The threshold values of I_T and U_T are calculated in the same way.

Using this method and trying different combinations of 5 or 7 bands of P, I_T and U_T, on the basis of the highest Pred(0.20) score the best banded model was found to be:

$$Nhat = 3.14(P)_{5B} + 2.59(I_T)_{7B} + 2.52(U_T)_{7B}$$

where,

$(P)_{5B}$ = values of P (number of unique predicate names) given in 5 bands
$(I_T)_{7B}$ = values of I_T (number of instantiated variables) given in 7 bands
$(U_T)_{7B}$ = values of U_T (number of uninstantiated variables) given in 7 bands

Table 9.7 : A tool for sizing Prolog programs (based on 80 Prolog programs)

Column 1							*Column 2*	*Column 3*
1. How many unique predicate names are to be used?								
	1	2	3	4	5			
	Very Low (<25)	**Low** (25-35)	**Medium** (36-45)	**High** (46-70)	**Very High** (>70)			
P=	**18.3**	**32.4**	**41.9**	**59.4**	**89.4**		____ * 3.14 =	____
2. How data-strong is the program (use of instantiated variables)?								+
	1	2	3	4	5	6	7	
	Extra Low (<50)	**Very Low** (50-80)	**Low** (81-100)	**Medium** (101-140)	**High** (141-180)	**Very High** (181-260)	**Extra High** (>260)	
IT=	**37.0**	**66.9**	**88.5**	**116.7**	**155.7**	**213.7**	**337.0**	____ * 2.59 = ____
3. How processing-strong is the program (use of uninstantiated variables)?								+
	1	2	3	4	5	6	7	
	Extra Low (<80)	**Very Low** (81-110)	**Low** (111-170)	**Medium** (171-240)	**High** (241-310)	**Very High** (311-590)	**Extra High** (>590)	
U_T=	**39.4**	**94.0**	**141.3**	**199.1**	**267.2**	**424.7**	**787.8**	____ * 2.52 = ____
								Nhat = ____

The values in each band can now be represented in a table (see Table 9.7), with Likert ratings from 1 to 5 for P_T, and 1 to 7 for I_T and U_T. This representation of the banded model also stands as a high-level tool for estimating the size of LPA Prolog programs.

In this case, the term I_T is seen as being indicative of a data-strong program on the basis that I_T reflects the number of instantiated variables that must be specified before the clauses can be executed. Hence, the size of I_T represents the size of data which must be programmed into the rule-base of Prolog clauses. On the other hand, U_T is seen as being indicative of a processing-strong program on the basis that U_T reflects the number of intermediary variable names which are used as the clauses are being executed. Hence, the size of U_T is the size of the intermediary processing being carried out by the system. These are regarded as preliminary definitions of I_T and U_T and may change in future developments of the tool.

To use the tool in Table 9.7, the project manager would select the band which best describes the program to be developed. Like most sizing models used early in the development process, selecting the "right" band will be a matter of estimating-by-analogy. To help in this process, the figures in parentheses are the upper and lower threshold values for the band. The surrogate value on the P= (or I_T= or U_T=) line is then entered into the second column which is then multiplied by the appropriate figure given in column 2 to produce a final value in column 3. The estimate is then the sum of the values in this last column.

For instance, if the program was decided to be "Low" on the number of predicate names to be used, "Medium" in terms of being data-strong, and "Low" in terms of being processing strong, then the values 32.4, 116.7 and 141.3 would be entered into the second column. In this case, multiplying by the appropriate number in column 2 and summing the result of column 3 would produce N*hat*=782.7.

By recording the initial estimate and final program size, N, of past projects, a database of analogies can be built up from which to decide on the most appropriate band to choose for subsequent projects and the value of the tool assessed. When this banded tool is applied to the LPA Prolog data-base, the accuracy is found to be MMRE=0.13 and PRED(0.20)=0.90. This extreme accuracy is clearly a property of the fact that the Prolog programs were developed by one person in a steady development environment. The fact remains, however, that the type of simple, local model advocated by Kitchenham (1992) and others can also be developed for Prolog.

9.6 Summary of metrics results

The results of the analysis carried out in the previous sections have shown that Prolog is amenable to analysis using metrics previously designed for conventional languages. In particular, models of size and structure have been deduced from the sample of 80 LPA Prolog programs, where a program is likely contain at least one documented post-release error when,

$$v(G) \geq 30 \quad \underline{\text{and}} \quad C_P \geq 55$$

where,

$v(G)$ = measure of cyclomatic complexity
C_P = measure of data flow-in/out

while the size of a Prolog program follows a linear regression model which has the form:

$$N\mathit{hat} = 3.14P + 2.59I_T + 2.52U_T$$

where,

P = the number of unique predicate names
I_T = the total number of instantiated variable names
U_T = the total number of uninstantiated variable names

The program size model based on Halstead's N out-performs any model based on a relationship between P, O, I, U (or their totals) with lines of code, where MMRE=0.06 and Pred(0.80)=0.99. This performance far exceeds the demands of accuracy sought by project managers in the survey carried out in Chapter 6. The use of v(G) and C_P is also important since, again, it out-performs LOC and where C_P is a measure which can be deduced early in the development process at the design stage. Calculating v(G) at the design stage is a problem which has been investigated recently and shows some promise (e.g., McCabe & Butler, 1989).

The program size model using P - the number of unique predicate names - and number of data variable names shows the most extraordinary accuracy. This may in part be due to the fact that the 80 programs studied were developed primarily by one person within the same development environment (machine, etc.). However, it does seem clear that good models can be developed for Prolog.

The ability to move from the models to the tools - where some method of predicting I_T and U_T would need to be found - is a matter of further investigation and would require substantially more samples of Prolog programs written by different programmers before a reasonable tool can be built.

10. Conclusion

"The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man's handiworks."
F.P.Brooks Jr. ('The Mythical Man-month', p177)

Summary: *The contributions to knowledge derived from this research are set out as well as the limitations of the research. The research questions posed in Chapter 1 are answered and areas of further work identified. The final remarks revolve around the potential of TABATHA (in Chapter 8) and the high-level sizing tool (in Chapter 9) to stand as useful models for use in hybrid systems development. The conclusion is that they can.*

Guided by the need to develop metrics and models to support the development of hybrid information systems, this research has posed and sought to answer a number of questions set out in Chapter 1. The theoretical questions asked:

- What problems do project managers face?
- What are "metrics" and "estimating tools"?
- How are they developed?
- How are they used?

The problems of project management and the need for software metrics were defined in Chapters 1 and 2. The literature review in Chapters 3, 4 and 5 focused on the need for metrics to estimate the size and quality of conventional DP systems, the similarity between DP and KBS metrics, and the metric→model→tool approach to developing software cost estimating (SCE) tools. Criticisms of the underlying metrics and models was such that there was a possibility the techniques could not, in fact, support project managers in the control of software development. Extending such technology to hybrid systems would then become a redundant exercise. This led to a number of empirical questions, which asked:

- Are estimating tools being used in industry?
- If not, why not?

- Does this affect the sorts of metrics and tools which a project manager would actually find useful?

The hypothesis that software metrics and models could not be usefully extended to hybrid systems was defined as 'Option Zero' in Chapter 6. It was stated that 'Option Zero' could be dismissed if it could be proved that:

1. Estimation is seen as a problem.
2. There is no prohibition to using SCE tools.
3. No other tools are being used in their place.
4. Conventional metrics can be extended to KBSs.

These points were subsequently investigated by conducting a survey of large UK corporations and computing companies. 'Large' companies were approached on the basis that they were more likely to be developing large projects, demanding stricter management control, and hence, more likely to be using project management tools such as SCE tools.

Point '1' was proved by showing that 91% of those surveyed agreed that estimation was a problem. However, it was also found that only 30% used any kind of SCE tool and suggested that point '2' was, in fact, false. That the lack of use was not a logistical problem was shown in Chapter 6 by setting out a three-point framework which suggested a company would use a SCE tool if:

- they had a structured methodology in place;
- they collected 'time' data at a high rate;
- they used a project management tool.

Point '2' was proved, therefore, by showing in Chapter 6 that 78% of companies satisfied this framework including all those which used SCE tools. A lack of knowledge of the functionality of SCE tools appeared to be a strong factor since only 39%, and probably less, could be said to have any real appreciation of the technology. Point '3' was investigated in Chapter 7 by comparing the group of 16 companies which used SCE tools, and a group of 9 companies which satisfied the three-point framework (above), had a good appreciation of the technology, but did not use the tools. It was deduced that a potential user would become an actual user if:

- they had high requirements volatility and developed large projects, or;
- they had high requirements volatility and charged a client.

Neither of these classifications would seem to adequately define the proper use of SCE tools, however, since SCE models do not specifically model the effect of high requirements volatility. 'Producing large projects' was an attribute of the companies targeted by the Chapter 6 survey.

The issue of charging a client was found to be more significant. On interviewing four of the companies which used SCE tools, it was found that one had stopped charging clients and had also stopped using their (internally developed) SCE tool. The suggestion seemed to be that SCE tools were being used in order to show a client that the process of estimating was being taken seriously, rather than in a belief that SCE tools generated accurate estimates. This "need to show" is further strengthened by recognising that high requirements volatility can also be traced to the client, in this case, changing the stated system requirements. By modelling the interaction between client and developer, the EEPS model suggested there are two points at which an estimate could be produced:

- at the earliest (pre-contract) negotiation stage (step 2), when the only tools available are SCE tools;
- during the detailed planning stage (step 4), when project management tools such as PMW are in almost universal use.

This suggests a conflict over the perception of what an estimate is, and on the use of SCE or project management/planning tools to generate an estimate. A telephone survey of 17 companies known to be familiar with both types of tool showed that both conflicts exist, while only bottom-up estimates at the (step 4) planning stage are perceived as "real."

Point '3' was shown to be true, therefore, by showing that SCE tools are the only tools available when tendering or bidding for a project. The use of a planning tool to estimate remains problematic because planning tools do not provide the crucial values upon which to generate a bottom-up estimate. The low use of SCE tools becomes clear, however, since if "real" estimates are perceived as being bottom-up and task-based, then it appears that the planning tools already in use can support the creation of this sort of estimate. This led to the final question in Chapter 1, which asked:

- "What would a hybrid metric/model/tool look like?"

Point '4' aimed to answer this question and was proved in two parts. Firstly, a description of a SCE tool which matched the required task-based functionality is given in Chapter 8. The tool - called TABATHA - also sought to accommodate the problem of requirements

volatility by defining a method of re-estimation. Only the tasks affected by the change in requirements are re-estimated, but with a non-linear cost associated with changing the original project plan and integrating the new work into the system. Time and cost constraints meant that TABATHA could not be validated empirically, but a metrics programme by which the model could be made into a working tool is given.

Secondly, point '4' was proved by extending conventional DP size and structure metrics to LPA Prolog. A tool called PSA was used to collect data from 80 commercially-developed LPA Prolog programs. "Structure" was defined in terms of non-comment, non-blank lines of code, LOC, McCabe's (1976) cyclomatic complexity metric, v(G), and Henry & Kafura's (1981, 1984) data-flow fan-in/out metric, C_P. Thresholds at which at least 80% of programs with documented post-release errors exceeded the threshold, T, were deduced for each metric and metrics in combination. Initially, if 100% of programs were tested then 57.5% would be found to have at least one error. By maximising the number of programs with errors that exceeded T whilst minimising the number of programs flagged, the best result was found to be when a program exceeded both v(G)=30 and C_P=55. In this case, 71.3% of programs were flagged which accounted for 84.8% of those which had errors.

"Size" was defined in terms of LOC and Halstead's (1972, 1977) program length equation, N. Using linear regression and the performance criteria MMRE≤0.20 and Pred(0.20)≥0.80, the most accurate models were produced by separating N*hat* into counts of predicates, P, total number of instantiated variables, I_T, and the total number of uninstantiated variables, U_T, for which MMRE=0.06 and Pred(0.20)=0.99. Banding the values of P, I_T and U_T and trying different combinations of 5 or 7 bands, a paper-based tool was produced, with MMRE=0.13 and Pred(0.20)=0.90. Such extreme accuracy is likely to be due to the fact that most of the 80 LPA Prolog programs were developed by a single programmer. The main contributions to knowledge contained within this thesis, therefore, can be summarised in the following points:

- Investigating the low use of SCE tools.
- Deducing the conflict between SCE and planning tools.
- Describing a task-based SCE tool which fits the way current UK project managers work.
- Extending conventional software metrics to LPA Prolog to deduce accurate metrics for program size and structure.
- Applying the metric→model→tool approach to deduce a tool to estimate the size of LPA Prolog programs.

10.1 Limitations of the research

Like any research project with a limited time scale, there are short-cuts and limitations within the research which, in most cases, are unavoidable. Some of the more notable problems encountered by this research are stated and explained below:

10.1.1 *Accuracy not an issue in the Chapter 6 survey*

Most studies on the use of SCE tools have focused on the accuracy of these tools (e.g., Mohanty, 1981; Kemerer, 1987; Kusters *et al*, 1991). The survey in Chapter 6, however, did not take accuracy as an issue because until it was established that SCE tools were addressing a recognised problem and were found to be widely used, the accuracy - or otherwise - of such tools would be of little interest to a project manager.

In other words, if estimation was not perceived to be a problem then there would be no need for the tools. Furthermore, if SCE tools did not model factors which were perceived to be meaningful then even if the tools were accurate the suspicion would remain that the accuracy was coincidental; that is to say, a set of inappropriate factors could combine to give an accurate estimate but only in the same way that a clock stuck at 9 'o' clock would still be accurate twice a day. A meaningful SCE model is the issue addressed in Chapter 6, therefore, and precedes any consideration of accuracy.

10.1.2 *Small sample of UK project managers*

The survey in Chapter 6 used a list of over 2000 companies (from ROGET 91) to derive a list of 115 companies, of which, 54 responded to the survey. In subsequent analyses, the sample fell to 25 companies in Chapter 7, with 17 responding to the final telephone survey. The issue being addressed in Chapter 7 was what might prevent a company from using SCE tools. As such, it was important that the sample contained companies who had some knowledge of the technology. The problem is that there are so few companies which have such a knowledge that gaining a larger sample than the 25 companies identified would be a substantial undertaking.

For instance, Heemstra & Kusters (1989) received 597 replies to their survey, with 14% (approximately 83 companies) defined as "actual users". Assuming the proportion of actual and potential users in the Dutch survey is the same as that found in Chapter 6, then there would be a further 46 companies which could be defined as

"potential users". Instead of the sample of 25 companies identified in Chapter 6 then, the Heemstra & Kusters (1989) survey has a potential sample of 129 companies which could be investigated on the same lines as that carried out in Chapter 7. Clearly, this is a sample of sufficient size to carry out the sort of statistical analyses missing from Chapter 7.

However, with a response rate of 22%, this would suggest Heemstra & Kusters began with a target population of 2713 companies. This is well above the list of 115 companies deduced from ROGET 91 for the UK survey. One conclusion might be that the selection criteria used in Chapter 6 is too strict, and that the use of SCE tools is more wide-spread in the UK than the survey suggests. However, since the proportion using SCE tools is more than twice that found by Heemstra & Kusters, it is also likely that the selection criteria used in Chapter 6 correctly defined those companies most likely to be users, i.e., those companies which can be defined as "large". Further responses to the Dutch survey, therefore, were from companies unlikely to be using SCE tools and so would reduce the proportion of users to non-users.

The size of the task to gain a sufficiently large sample of potential/actual SCE tool-using companies can now be appreciated. To receive a sample of 597 replies with a response rate of 22%, the target population would have to be around 2700. If only 115 suitable candidates can be noted from (around) 2200 ROGET 91 entries, then a sample of 2700 would require a ROGET 91-like database of over 50 000 companies. A sample size of this magnitude suggests an international rather than a national survey. Although the sample of companies in Chapter 6 is small, therefore, it can also be argued that this faithfully reflects the proportion of companies who have any real appreciation of software metrics, SCE models and their associated tools. If it were not for the high 47% response rate to the original (Chapter 6) survey, these samples would have been even smaller.

10.1.3 *No validated TABATHA model*

Ideally, the product of this research would have been a properly worked out and validated TABATHA model integrated into the RUSSET tool. However, a final version of the RUSSET tool only became available in the last few months of the IED4/1/1426 project, and so TABATHA did not have the crucial input-output product information which would have guided the collection of data for a particular methodology. There also remains the issue of the cost of a metrics programme to collect the necessary data and this problem would have to be overcome before any

follow-up project could attempt to build the TABATHA SCE tool. It is the cost of such a programme which might be the biggest barrier to the development of TABATHA.

A further problem is the apparent complexity of estimating at the task-based level. However, if an analysis of the project plan is the point at which project managers perceive estimation to exist, then, presumably, they would not have any reservations about using a tool which dealt with exactly this level of analysis.

10.1.4 *Small sample of LPA Prolog programs*

The sample of Prolog programs may appear small with only one system studied and an analysis carried out on 80 programs. This was inspite of adverts placed in two international newsletters published by the vendors of LPA Prolog and Prolog-2. This problem is almost certainly to do with the lack of companies that have an interest in both software metrics and the development of KBSs. Although three companies did respond to the advert, the immature nature of the PSA tool and the untried character of the metrics proposed may have been the root cause of their decision not to supply any data. Since the fourth company was local, it was possible for the tool to be taken to the company and run by the author without any time being spent by the company concerned.

On the other hand, a set of 80 data-points is towards the top end of samples used by the classic software metric studies, for instance: Halstead (1972, 1977) studied only 14 algorithms to support the program length equation; McCabe (1976) only indicates an analysis of 24 'troublesome' programs; Albrecht & Gaffney (1981, 1984) give details of studies with 20-30 systems; Symons (1988) studied 9 systems from two organisations, while; Henry & Kafura (1981, 1984) studied the UNIX operating system which contained 165 modules. The sample size tends to be higher in the development of SCE models, for instance: Farr & Zagorki's (1965) biggest sample was of 25 systems; Boehm (1981) looked back over 13 years in order to collect data on 63 systems, while; Thebaut & Shen (1984) had only 12 data-points. With data on 80 programs it seems reasonable to suggest that the study presented in this thesis is of the same order as other initial metrics studies.

10.1.5 *No extension of hybrid metrics to conventional systems*

Hybrid metrics are defined as the extension of conventional metrics to a KBS development language, such as Prolog. This was primarily because the literature on

conventional metrics is considerably more extensive than for KBS metrics. However, the sizing tool developed in Chapter 9 defined "size" in terms of the number of predicates, and totals of instantiated variables and uninstantiated variables (P, I_T and U_T, respectively). It might then be unclear how this model relates to conventional systems, especially when Software Science is generally regarded as discredited, and SCE models typically require the first input to be in terms of lines of code, not program length.

The analysis here used Software Science to classify elements of Prolog programs in terms of counts of operators and operands. The logarithmic model of N*hat*, however, was dismissed in favour of a simpler, linear model. This simpler model was also found to outperform models of lines of code. On this basis, the best model of system size is not lines of code, but a measure of function names (like the count P in Prolog), and the total number of instantiated and uninstantiated variables (like the count I_T and U_T in Prolog). This count is clearly similar to FPA, although no distinction is made here between internal and external types of data. As such, it would seem possible to apply the same model to conventional systems, although the proof of this would require further data collection.

10.2 Areas of further research

A number of outstanding areas of further research can be identified by the work carried out by this thesis. In particular, there are five key issues which are stated and explained below.

10.2.1 *Establishing the meaningfulness of metric results*

The question remains throughout this thesis of whether the usefulness of size or structure metrics should be established by their psychological validity, or whether the nature of programming is to be uncovered by the application of software metrics. Both scenarios would seem to have some validity since it is clearly important that what is measured is intuitively related to what it is meant to represent. This is what led Kitchenham *et al* (1990) to insist that complexity metrics should relate to the subjective complexity ratings of a programmer. However, the correlation between the LPA Prolog structure metrics and the ratings provided by the programmer mainly responsible for the development of the programs remained poor (see again §9.3.1). On the other hand, the use of psychological concepts to build an understanding of software development has been both supported (e.g., Halstead, 1977) and denied (e.g., Coulter, 1983).

Since systems design and programming are strongly intellectual and creative tasks, it seems reasonable to suggest that a study of software development will be based on or take account of the psychology of the development process. It does not seem reasonable to follow Kitchenham *et al*'s (1990) suggestion, however, that the psychological rating of complexity should be the basis upon which to judge the efficacy of structure metrics. This would be tantamount to suggesting that what is meant by "psychological complexity" has been established and so can be used as a bench-mark against which to judge other more questionable representations of the same feature. But neither philosophy or psychology has been able to properly define the psychology of Man - let alone programmers - to everybody's general satisfaction. The psychology of software development and the results of software metrics (which also includes size metrics) must be linked somehow, what remains an open issue is the extent to which one reflects and judges the usefulness of the other.

10.2.2 *Control of SCE model databases*

When describing the development of SCE models in Chapter 5, it was noted on a number of occasions that the veracity of the model (and consequently, the SCE tool) depended on the quality of the data contained within the underlying database. However, it is not clear by what means the quality of this data can be guaranteed. For instance, "person-hours" is a key factor in estimating cost but it is unclear how this data should be recorded and at what detail. Manually each week? By the computer using log-on time? To the nearest day? To the nearest half-hour?

Although SCE tools tend to give effort in terms of person-months, this level of detail would be too low to detect the effect of new tools or techniques on a particular task. Collecting effort by the hour may also suggest a greater level of accuracy than is warranted. While computer-led data collection has the advantage of a consistent and automatic approach, manual data collection has the advantage of the project manager being able to query the information submitted. If, as has been argued in §10.1.5 that a FPA-like count of functions and data is a better model of system size than LOC, then it can be presumed that person-hours or person-months manually or automatically collected is not necessarily the best measure of development effort. Questioning the type of data collected and the means of collection will undoubtedly lead to better insights as to the best means of ensuring the quality of SCE model databases.

10.2.3 *Better tools to develop SCE models*

One factor which seems to prohibit the development of local SCE models and tools is the cost of collecting and analysing the data required to build such models. The need is clear, therefore, for a series of tools which can quickly and cheaply perform this task. Such tools might be as simple as a spreadsheet (Kokol, 1989) while more sophisticated programmes might include:

- A PSA-like tool which collects the low-level data, but which can sit within the mainframe or network of a development environment and unobtrusively collect program size and effort data.
- A system of collecting the notes of project managers during the project which can then be used to highlight and explain the points at which problems first arise. This would be used before project management/planning tools and would give some clue as to the first point at which accurate estimates can genuinely be produced rather than hoped for.
- A KBS-database which receives, sorts and presents the data collected. The KBS element would be required to judge the quality of the data being received and flagging data-points which appear unusual or unreasonable.

The three tools described above are suggestions of the types of tools a project manager might need in order to support the rapid development of local models, ensure the quality of the data, and question the demand for accurate estimates by senior management of clients at the outset of a project. Although some of the ESPRIT-funded projects have attempted to develop similar tools (e.g., MERMAID, COSMOS, METKIT, etc.), the aim here is for an integrated set of tools for use by the project managers themselves.

10.2.4 *Modelling corrective actions by project managers*

It seems non-sensical that models using results from completed projects should be judged on the basis of estimates made early in the life of a project. Specifically, if high requirements volatility is a standard problem, there seems no reason to believe that the initial estimate will bear any resemblance to what happens between the functionality required at the beginning and the functionality of the completed system. This is not a problem of communication, but a problem of revising the stated requirements as the nature of the required system is better understood.

Under these conditions, the issue is how a project manager can best accommodate changes if budgetary and resource constraints remain tight. Clearly, this can be done well or badly and will make a critical impact on the relationship between developer and client. If the skills and techniques of a project manager who excels in this difficult area could be understood and incorporated into a SCE tool, therefore, the resultant KBS tool would not only be able to produce better estimates (by acknowledging and anticipating the likelihood of change), but the tool would be able to re-estimate and advise on the best course of action when the situation does arise. This would clearly have a significant impact on the training of future project managers as well as an aid to current management.

10.2.5 *Tools which meet the way project managers actually work*

The key point about which this research turns is that it cannot be assumed that a tool which can support a particular task will be used. The difference lies in the context within which the project manager works and the tools and techniques already at hand. In this case, SCE tools were not being used because estimation was not seen as a task in itself, but as part of the process of detailed project planning where planning tools are already in use. To produce a tool which failed to recognise this point is to make a tool which is not perceived as being useful.

Clearly, project managers cannot generate estimates using planning tools, but if SCE tools are portrayed as generating estimates at the earliest stages of a project where it is perceived that no sensible estimates can be produced, then the apparent strength of SCE tools results in its downfall. The solution could be as simple as renaming these types of tool "pre-contract tendering/bidding" tools. In this case, the output from the tool would at least be recognised as different, i.e., more likely to be inaccurate, than the "real" estimate generated during planning. Only tools which estimate alongside the plan could be called "estimating" tools.

The inadequacy of attempting to educate project managers as to what SCE tools mean by an "estimate" is reflected in the belief that the actual practicalities of software development is the best testing ground for ideas on tools to support the process. In other words, the way project managers currently work is likely to be the best synthesis of available tools and techniques that deal with the problems of software development. This is the crucial difference between a tool which is usable, i.e., the requirements for their use are in place, and a tool which is useful, i.e., one which fulfils a required function. Current SCE tools have been found to be usable but not useful.

The fact that project managers are making the best use of currently available tools and techniques raises the further issue of whether the current state of these tools and techniques are satisfactory. Since there is no clear definition of what is meant by 'project management,' it is difficult to know what sorts of tools would best support the tasks being carried out. This problem remains an area of further research.

10.3 Final remarks

As pointed out by Brooks (1975, p177) and Browne & Shaw (1981, p70), the problem with understanding and modelling the nature of project management in software development is that the subject under study is constantly changing. It therefore becomes a matter of debate how long a piece of research remains valid. For instance, the demand for ever more complex systems leads to ever more complex problems. The use of new techniques and tools changes the type of system which can be developed, while the intrinsically intellectual nature of software development makes it difficult to have a clear understanding of what goes on when software is developed. Under these conditions, research in software engineering is in danger of acquiring a transitory importance since it is not unreasonable to suggest that what was true ten years ago in software development is no longer true now.

The advantage of the research carried out in this thesis is that it has focused on the people involved in software development. No matter how much the environment changes, the human factor will remain. "People", in this case, is the project manager attempting to generate and defend an estimate of cost, duration and quality to a client demanding value-for-money. The task-based nature of the SCE tool thus proposed fits into the way UK project managers recognise the usefulness of tools in negotiation with a (probably non-computer literate) client. The estimate at step 2 of EEPS is an "estimate" for the benefit of the client. The meaning of the "real" estimate identified at step 4 of the EEPS model is the point at which the project manager believes there is enough detail to make a first attempt at a reasonable estimate. These two points in the life-cycle would be valid for any project, conventional DP or knowledge based.

The validity of TABATHA, therefore, is that it does not attempt to provide the project manager with new functionality, but attempts to support the point at which estimates are expected, and by the same means by which they are already produced. As long as the database upon which TABATHA is based remains up-to-date (and this remains a crucial area of further research), the task-based approach will remain valid even as other methods of software development change. It is on this basis that the research contained within this thesis is presented as a significant contribution to knowledge.

References

AEI (1989) "Clues to success: Information technology strategies for tomorrow." Report produced by *Amdahl Executive Institute*, Dogmersfield Park, Hartley Wintney, Hants.

ALBRECHT A J and GAFFNEY Jr. J E (1983) "Software function, source lines of code, and development effort prediction: A software science validation." *IEEE Transactions on Software Engineering*, **SE-9**(6), 639-648.

ANDERSON R G (1989) "*Information Systems in Development and Operation.*" Pitman Publishing, London.

ANDREWS B (1989) "Successful expert systems." Report published by *The Financial Times Business Information*, London.

AVISON D E, SHAH H U, POWELL R S and UPPAL P S (1992) "Applying methodologies for information systems development." *Journal of Information Technology*, 7, 127-140.

BADER J L (1988) "Knowledge-based Systems and Software Engineering." *PhD thesis*, IHD Scheme, Aston University.

BAILEY J W and BASILI V R (1981) "A meta-model for software development resource expenditure." In *Proceedings, 5th International Conference on Software Engineering*, IEEE Computer Society Press, pp107-116.

BAKER A L and ZWEBEN S H (1980) "A comparison of measures of control flow complexity." *IEEE Transactions on Software Engineering*, **SE-6**(6), 506-511.

BANKER R D and KEMERER C F (1989) "Scales economies in new software development." *IEEE Transactions on Software Engineering*, **SE-15**(10), 1199-1205.

BARR A and FEIGENBAUM E A (1982) "*The Handbook of Artificial Intelligence, Volume II* (Eds.)." Pitman Books, London.

BASDEN A (1984) "On the application of expert systems." In M.J.Coombs (Ed.), *op cit.*, pp59-75.

BASILI V R and HUTCHENS D H (1983) "An empirical study of a syntactic complexity family." *IEEE Transactions on Software Engineering*, **SE-9**(6), 652-663.

BASILI V R and PERRICONE B T (1984) "Software errors and complexity: An empirical investigation." *Communications of the ACM*, **27**(1), 42-52.

BASILI V R and ROMBACH H D (1988) "The TAME project: Towards improvement-oriented environments." *IEEE Transactions on Software Engineering*, **SE-14**(6), 758-773.

BASILI V R, SELBY R W and PHILLIPS T (1983) "Metric analysis and data validation across Fortran projects." *IEEE Transactions on Software Engineering*, **SE-9**(6), 652-663.

BAUMERT J, CRITCHFIELD A and LEAVITT K (1988) "The need for a comprehensive expert system development methodology." *Telematics and Informatics*, **5**(3), 325-334.

BEHRENDT W, LAMBERT S C, RINGLAND G A, HUGHES P and POULTER K (1991) "GATEWAY: Metrics for knowledge based systems." In *Proceedings, First World Congress on Expert Systems*, Miami, 4-7 December, 1991, pp1056-1067.

BETTERIDGE R (1992) "Successful experience of using function points to estimate project costs early in the life-cycle." *Information and Software Technology*, **34**(10), 655-658.

BOEHM B W (1976) "Software engineering." *IEEE Transactions on Computers*, **C-25**(12), 1226-1241.

BOEHM B W (1981) "*Software Engineering Economics.*" Prentice-Hall, Englewood Cliffs, NJ.

BOEHM B W (1984) "Software engineering economics." *IEEE Transactions on Software Engineering*, **SE-10**(1), 4-21.

BOEHM B W (1988a) "A spiral model of software development." *IEEE Computer*, **May**, 61-72.

BOEHM B W (1988b) "Overview of ADA-COCOMO." In *Proceedings, 4th COCOMO User Group Meeting*, Pittsburgh, November, 1988.

BOEHM B W (1991) "Software risk management: Principles and practice." *IEEE Software*, **January**, 32-41.

BOEHM B W, BROWN J R and LIPOW M (1976) "Quantitative evaluation of software quality." In *Proceedings, 2nd International Conference on Software Engineering*, IEEE Computer Society Press, pp592-605.

BOEHM B.W and PAPACCIO P N (1988) "Understanding and controlling software costs." *IEEE Transactions on Software Engineering*, **SE-14**(10), 1462-1477.

BOEHM B W and ROSS R (1989) "Theory-W software project management: Principles and examples." *IEEE Transactions on Software Engineering*, **SE-15**(7), 902-916.

BOLLINGER T B and McGOWAN C (1991) "A critical look at software capability evaluations." *IEEE Software*, **July**, 25-41.

BOLTER J D (1984) "*Turing's Man.*" Penguin Books, Harmondsworth.

BOOSE J and GAINES B (1988) "*Knowledge Acquisition Tools for Expert Systems* (Eds.)." Academic Press, London.

BORNING A (1987) "Computer system reliability and nuclear war." *Communications of the ACM*, **30**(2), 112-131.

BRAMER M A and MILNE R W (1993) "*Research and Development in Expert Systems IX* (Eds.)." Cambridge University Press, Cambridge.

BRATKO I (1986) "*Prolog Programming for Artificial Intelligence.*" Addison-Wesley Publishing Co., Wokingham.

BREUCKER J and WIELINGA B (1989) "Models of expertise in knowledge acquisition." In G.Guido and C.Tasso (Eds.), *op cit.*, pp265-295.

BROCKLEHURST S and LITTLEWOOD B (1992) "New ways to get accurate reliability measures." *IEEE Software*, **July**, 34-42.

BROOKS F P Jr. (1975) "*The Mythical Man-Month: Essays on Software Engineering.*" Addison-Wesley, Reading, MA.

BROOKS R (1977) "Towards a theory of the cognitive processes in computer programming." *International Journal of Man-Machine Studies*, **9**, 737-751.

BROWNE J C and SHAW M (1981) "Towards a scientific basis for software evaluation." In A.J.Perlis *et al* (Eds.), *op cit.*, pp19-41.

CARD D N (1990) "Software quality engineering." *Information & Software Technology*, **32**(1), 310.

CARD D N and AGRESTI W W (1988) "Measuring software design complexity." *Journal of Systems and Software*, **8**, 185-197.

CARD D N, CHURCH V E and AGRESTI W W (1986) "An empirical study of software design practices." *IEEE Transactions on Software Engineering*, **SE-12**(2), 264-271.

CHAPIN N (1979) "A measure of software complexity." In *AFIPS Conference Proceedings of the 1979 National Computer Conference*, 4-7 June, New York, pp995-1002.

CHEN E T (1978) "Program complexity and programmer productivity." *IEEE Transactions on Software Engineering*, **SE-4**(3), 187-194.

CHEN Y S (1992) "Zipf-Halstead's theory of software metrication." *International Journal of Computer Mathematics*, **41**, 125-138.

CHERNIAVSKY J C and SMITH C H (1991) "On Weyuker's axioms for software complexity measures." *IEEE Transactions on Software Engineering*, **SE-17**(6), 636-638.

CHRISTENSEN K, FITSOS G P and SMITH C P (1981) "A perspective on software science." *IBM Systems Journal*, **20**(4), 372-387.

CLOCKSIN F W and MELLISH C S (1981) "*Programming in Prolog.*" Springer-Verlag, New York, NY.

CONTE S D, DUNSMORE H E and SHEN V Y (1986) "*Software Engineering Metrics and Models.*" The Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA.

COOMBS M J (1984) "*Developments in Expert Systems* (Ed.)." Academic Press, London.

CORBETT M and KIRAKOWSKI J (1992) "An analogical approach to cost estimation," In *Proceedings, European Software Cost Modelling Meeting*, Munich, 27-29 May, 1992.

COTE V and St.-DENIS R (1991) "A dynamic measurement for software cost estimation." In *Proceedings, Fourth International Conference on Software Engineering & its Applications*, Toulouse, 9-13 December, 1991.

COULTER N S (1983) "Software science and cognitive psychology." *IEEE Transactions on Software Engineering*, **SE-9**(2), 166-171.

COUPAL D and ROBILLARD P N (1990) "Factor analysis of source code metrics." *Journal of Systems and Software*, **12**, 263-269.

COURTNEY R E and GUSTAFSON D A (1993) "Shotgun correlations in software measures." *Software Engineering Journal*, **January**, 5-13.

CURTIS B, SHEPPARD S, MILLIMAN P, BORST M and LOVE T (1979) "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics." *IEEE Transactions on Software Engineering*, **SE-5**(2), 96-104.

D'AGAPAYEFF A (1987) "*Report to the Alvey Directorate on the Second Survey of Expert Systems in UK Business.*" IEE Computer Society Press, London.

DAVIS J S and LeBLANC R J (1988) "A study of the applicability of complexity measures." *IEEE Transactions on Software Engineering*, **SE-14**(9), 1366-1372.

DeMARCO T (1982) "*Controlling Software Projects: Management, Measurement and Estimation.*" Yourdon Press, New York, NY.

DODD T (1990) "*Prolog: A logical approach.*" Oxford University Press, Oxford.

DTI (1992) "Knowledge based systems: Survey of UK applications." Report produced for the *Department of Trade and Industry* by Touche Ross Management Consultants.

DUDA R, GASCHNIG J and HART P E (1979) "Model design in the PROSPECTOR consultant system for mineral exploration." In D.Michie (Ed.), *op cit.*, pp153-167.

DUNSMORE H E and GANNON J D (1980) "An anlysis of the effects of programming factors on programming effort." *Journal of Systems and Software*, **1**(2), 144-153.

EDWARDS J S (1991) "*Building Knowledge-Based Systems: Towards a Methodology.*" Pitman Publishing, London.

EJIGOU L O (1985) "A simple measure of software complexity." *ACM SIGPLAN Notices*, **20**(3), 16-31.

EJIGOU L O (1991) "TM: A systematic methodology of software metrics." *ACM SIGPLAN Notices*, **26**(1), 124-132.

ELSHOFF J L (1976) "Measuring commercial PL/1 programs using Halstead's criteria." *ACM SIGPLAN Notices*, **11**(5), 38-46.

ELSHOFF J L (1978) "An investigation into the effects of the counting method used on software science measurements." *ACM SIGPLAN Notices*, **13**(2), 30-45.

EVANGELIST W M (1982) "Software complexity metric sensitivity to program structuring rules." *Journal of Systems & Software*, **3**, 231-243.

EVANGELIST W M (1983) "Relationships among computational, software, and intuitive complexity." *ACM SIGPLAN Notices*, **18**(12), 57-59.

EYSENCK M W (1984) "*A Handbook of Cognitive Psychology.*" Lawrence Erlbaum Associates, London.

FARR L and ZAGORSKI H J (1965) "Quantitative analysis of programming cost factors: A progress report." In *Proceedings, ICC Symposium on Economics of Automatic Data Processing*, North-Holland, Amsterdam.

FEIGENBAUM E A, BUCHANAN B G and LEDERBERG J (1971) "On generality and problem solving: A case study using the DENDRAL program." In B.Meltzer and D.Michie (Eds.), *op cit.*, pp165-190.

FELICAN L and ZALATEU G (1989) "Validating Halstead's theory for Pascal programs." *IEEE Transactions on Software Engineering*, **15**(12), 1630-1632.

FENTON N E (1991) "*Software Metrics: A Rigorous Approach.*" Chapman & Hall, London.

FENTON N E and WHITTY R W (1986) "Axiomatic approach to software metrication through program decomposition." *The Computer Journal*, **29**(4), 329-399.

FISHER D and GORMAN J M (1990) "An evaluation of software cost estimating tools." *Report TAS071*, Inland Revenue, Matheson House, Telford, Shropshire TF3 4ER.

FITZSIMMONS A and LOVE T (1978) "A review and evaluation of software science." *ACM Computing Surveys*, **10**(1), 3-18.

FORGY C L (1982) "RETE: A fast algorithm for the many pattern/many object pattern match problem." *Artificial Intelligence*, **19**(1), 17-37.

FREIMAN F R and PARK R E (1979) "Price software model - version 3. An overview." In *Proceedings, IEEE-Pliny Workshop on Quantatative Software Models*, pp32-41.

FUNAMI Y and HALSTEAD M H (1976) "A software physics analysis of Akiyama's debugging data." In *Proceedings, MRI XXIV Symposium on Computer Software Engineering*, Polytechnic Press, New York, pp133-138.

GLADDEN G R (1982) "Stop the life cycle - I want to get off." *ACM Software Engineering Notes*, **7**(2), 35-39.

GOODMAN P A (1992) "Application of cost-estimation techniques: Industrial perspective." *Information and Software Technology*, **34**(6), 379-382.

GORDON R D (1979) "Measuring improvements in program clarity." *IEEE Transactions on Software Engineering*, **SE-5**(2), 79-90.

GORDON R D and HALSTEAD M H (1976) "An experiment comparing Fortran programming times with the software physics hypothesis." In *AFIPS Conference Proceedings of the 1976 National Computer Conference*, pp935-937.

GRADY R B and CASWELL D L (1987) "*Software Metrics: Establishing a Company-wide Program.*" Prentice-Hall, Inc., Englewood Cliffs, NJ.

GUIDO G and TASSO C (1989) "*Topics in Expert System Design* (Eds.)." Elsevier/North-Holland, Amsterdam.

GUNTON T (1989) "*Infrastructure: Building a Framework for Corporate Information Handling.*" Prentice-Hall, New York, NY.

GUPTA A and FORGY C L (1983) "Measurements on production systems." *Report CMU-CS-83-167*, Department of Computer Science, Carnegie-Mellon University.

GUPTA A and FORGY C L (1989) "Static and run-time characteristics of OPS5 production systems." *Journal of Parallel and Distributed Computing*, **7**, 64-95.

HAGER J A (1989) "Software cost reduction methods in practice." *IEEE Transactions on Software Engineering*, **SE-15**(12), 1638-1644.

HALSTEAD M H (1972) "Natural laws controlling algorithm structure." *ACM SIGPLAN Notices*, 7(2), 19-26.

HALSTEAD M H (1977) "*Elements in Software Science.*" Elsevier North-Holland, New York, NY.

HAMER P G (1991) "Looking at data." In *Proceedings, European Software Cost Modelling Meeting*, Noordwijk, 11-13 June, 1991.

HAMER P G and FREWIN G D (1982) "M.H.Halstead's software science: A critical examination." In *Proceedings, 6th International Conference on Software Engineering*, IEEE Computer Society Press, pp197-206.

HANSEN W J (1978) "Measurement of program complexity by the pair (cyclomatic number, operator count)." *ACM SIGPLAN Notices*, **13**(3), 29-33.

HARRIS-JONES C, BARRETT T, WALKER T, MOORES T and EDWARDS J (1992) "A framework for KBS/conventional methods integration." In *Proceedings, IEE Software Engineering and AI Colloquium*, London, 10 April, 1992.

HARRIS-JONES C, BARRETT T, WALKER T, MOORES T and EDWARDS J (1993) "A methods model for the integration of KBS and conventional information technology." In M.A.Bramer and R.W.Milne (Eds.), *op cit.*, pp25-43.

HARRISON W and MAGEL K (1981) "A complexity measure based on nesting level." *ACM SIGPLAN Notices*, **16**(3), 63-74.

HAYES-ROTH F (1989) "Towards benchmarks for knowledge systems and their implications for data engineering." *IEEE Transactions on Knowledge and Data Engineering*, **1**(1), 101-100.

HAYES-ROTH F, WATERMAN D A and LENAT D B (1983) "*Building Expert Systems* (Eds.)." Addison-Wesley, Reading, MA.

HAYWARD S A, WIELINGA B J and BREUCKER J A (1988) "Structured analysis of knowledge." In J.Boose and B.Gaines (Eds.), *op cit.*, pp149-160.

HEEMSTRA F J (1992) "Software cost estimation." *Information and Software Technology*, **34**(10), 627-639.

HEEMSTRA F J and KUSTERS R J (1989) "Controlling software development costs: A field study." In *Proceedings, International Conference on Organization and Information Systems*, Bled, Yugoslavia, 13-15 September, 1989, pp652-664.

HEEMSTRA F J and KUSTERS R J (1991) "Function point analysis: Evaluation of a software cost estimation model." *European Journal of Information Systems*, **1**(4), 229-237.

HENRY S and KAFURA D (1981) "Software structure metrics based on information flow." *IEEE Transactions on Software Engineering*, **SE-7**(5), 510-518.

HENRY S and KAFURA D (1984) "The evaluation of software systems' structure using quantitative software metrics." *Software - Practice and Experience*, **14**(6), 561-571.

HENRY S and SELIG C (1990) "Predicting source-code complexity at the design stage." *IEEE Software*, **March**, 36-44.

HERD J R, POSTAK J N, RUSSELL W E and STEWART K R (1977) "Software cost estimation study - study results. Final Technical Report." *Report RADC-TR-77-220* (June 1977), Doty Associates, Inc., Rockville, MD.

HUMPHREY W S (1989) "*Managing the Software Process.*" Addison-Wesley, Reading, MA.

HUMPHREY W S, SNYDER T R and WILLIS R R (1991) "Software process improvement at Hughes Aircraft." *IEEE Software*, **July**, 11-23.

INCE D C (1984) "Module interconnection language and Prolog." *ACM SIGPLAN Notices*, **19**(8), 89-93.

INCE D (1990) "An annotated bibliography of software metrics." *ACM SIGPLAN Notices*, **25**(8), 15-23.

INCE D and ANDREWS S (1990) "*The Software Life Cycle* (Eds.)." Butterworths, London.

INCE D and HEKMATPOUR S H (1988) "*Software Prototyping: Formal Methods and VDM.*" Addison-Wesley, MA.

IPL (1989) "Software Quality Survey." Confidential report produced by *Information Processing Limited*, Eveleigh House, Grove Street, Bath.

IYENGAR S S, PARAMESWARAN N and FULLER J (1982) "A measure of logical complexity of programs." *Computer Languages*, **7**, 147-160.

JEFFREY D R and LOW G (1990) "Calibrating estimation tools for software development." *Software Engineering Journal*, **July**, 215-221.

JENKINS A M, NAUMANN J D and WETHERBE J C (1984) "Empirical investigation of systems development practices and results." *Information Management*, **7**, 73-82.

JENSEN R W (1983) "An improved macrolevel software development resource estimation model." In *Proceedings, 5th International Society of Parametric Analysis*, pp87-92.

JENSEN R W (1984) "A comparison of the Jensen and COCOMO schedule and cost estimation models." In *Proceedings, 6th International Society of Parametric Analysis*, pp96-106.

JOHNSTON D B and LISTER A M (1981) "A note on the software science length equation." *Software - Practice and Experience*, **11**, 875-879.

JONES T C (1978) "Measuring programming quality and productivity." *IBM Systems Journal*, **17**(1), 39-63.

JONES T C (1986) "*Programming Productivity.*" McGraw-Hill Book Co., New York, NY.

JONES T C (1991) "*Applied Software Measurement: Assuring Productivity and Quality.*" McGraw-Hill, Inc., New York, NY.

KAFURA D and REDDY G R (1987) "The use of software complexity metrics in software maintenance." *IEEE Transactions on Software Engineering*, **SE-13**(3), 335-343.

KAISLER S H (1986) "Expert system metrics." In *Proceedings, International Conference on Neural Networks*, Atlanta, 14-17 October, 1986, pp114-120.

KEMERER C F (1987) "An empirical validation of software cost models." *Communications of the ACM*, **30**(5), 416-429.

KEMERER C F (1992) "Function point measurement reliability: A field experiment." In *Proceedings, European Software Cost Modelling Meeting*, Munich, 27-29 May, 1992.

KERSCHBERG L (1990) "Expert database systems: Knowledge/data management environments for intelligent information systems." *Information Systems*, **15**(1), 151-160.

KITCHENHAM B A (1981) "Measures of programming complexity." *ICL Technical Journal*, **May**, 298-316.

KITCHENHAM B A (1991) "Software metrics." In *Proceedings, European Software Cost Modelling Meeting*, Noordwijk, 11-13 June, 1991.

KITCHENHAM B A (1992) "Empirical studies of assumptions that underlie software cost-estimation models." *Information and Software Technology*, **34**(4), 211-218.

KITCHENHAM B A, PICKARD L M and LINKMAN S J (1990) "An evaluation of some design metrics." *Software Engineering Journal*, **5**(1), 50-58.

KOKOL P (1989) "Using spreadsheet software to support metrics life-cycle activities." *ACM SIGPLAN Notices*, **24**(5), 27-37.

KONIG W and BEHRENDT R (1989) "The production of expert systems." *Angewandte Informatik*, **31**(3), 95-102. (In German.)

KUSTERS R J, van GENUCHTEN M and HEEMSTRA F J (1991) "Are software cost-estimating models accurate?" In R.Veryard (Ed.), *op cit*, pp155-161. Also published in *Information and Software Technology*, **32**(3), 187-190.

LARANJEIRA L A (1990) "Software size estimation of object-oriented systems." *IEEE Transactions on Software Engineering*, **SE-16**(5), 510-522.

LASSEZ J-L, van der KNIJFF, SHEPHERD J and LASSEZ C (1981) "A critical examination of software science." *Journal of Systems and Software*, **2**, 105-112.

LEDERER A L and PRASAD J (1992) "Nine management guidelines for better cost estimating." *Communications of the ACM*, **35**(2), 51-59.

LINKMAN S G and WALKER J G (1991) "Controlling programs through measurement." *Information and Software Technology*, **33**(1), 93-102.

LIPOW M (1982) "Number of faults per line of code." *IEEE Transactions on Software Engineering*, **SE-8**(4), 437-439.

LISTER A M (1982) "Software science - The Emporer's new clothes?" *The Australian Computer Journal*, **14**(2), 66-70.

LOW G C and JEFFREY D R (1990a) "Function points in the estimation and evaluation of the software process." *IEEE Transactions on Software Engineering*, **SE-16**(1), 64-71.

LOW G C and JEFFREY D R (1990b) "Calibrating estimation tools for software development." *Software Engineering Journal*, **5**(4), 215-221.

MacDONELL S G (1991) "Rigour in software complexity measurement experimentation." *Journal of Software and Systems*, **16**, 141-149.

MACLEISH K J and VENNERGRUND D A (1986) "An expert system development life-cycle model and its relevance to traditional software systems." In *Proceedings, 5th Annual International Phoenix Conference on Computers and Communications*, Phoenix, 26-28 March, 1986, pp592-596.

MARKUSZ Z and KAPOSI A A (1985) "Complexity control in logic-based programming." *The Computer Journal*, **28**(5), 487-495.

MATSON J E and MELLICAMP J M (1993) "An object-oriented tool for function point analysis." *IEEE Expert Systems*, **10**(1), 3-14.

McCABE T J (1976) "A complexity measure." *IEEE Transactions on Software Engineering*, **SE-2**(4), 308-320.

McCABE T J and BUTLER C W (1989) "Design complexity measurement and testing." *Communications of the ACM*, **32**(12), 1415-1425.

McCRACKEN D D and JACKSON M A (1982) "Life cycle concept considered harmful." *ACM Software Engineering Notes*, **7**(2), 28-32.

McDERMOTT J (1980) "R1: An expert in the computer systems domain." In *Proceedings, 1st National Conference of the American Association for Artificial Intelligence*, Stanford, pp269-271.

MELTZER B and MICHIE D (1971) "*Machine Intelligence, Volume 6* (Eds.)." Edinburgh University Press (Wiley), New York, NY.

MICHIE D (1979) "*Expert Systems in the Micro-Electronic Age* (Ed.)." Edinburgh University Press, Edinburgh.

MICHIE D and JOHNSTON R (1984) "*The Creative Computer.*" Viking Press, London.

MILLER G A (1956) "The magical number seven, plus or minus two: Some limits on our capacity for processing information." *Psychological Review*, **63**, 81-97.

MILLER L A (1990) "Dynamic testing of knowledge bases using the heuristic testing approach." *Expert Systems with Applications*, **1**, 249-269.

MIYAZAKI Y (1991) "COCOMO tailoring and its support tool." In *Proceedings, European Software Cost Modelling Meeting*, Noordwijk, 11-13 June, 1991.

MIYAZAKI Y and MORI K (1985) "COCOMO evaluation and tailoring." In *Proceedings, 8th International Conference on Software Engineering*, IEEE Computer Society Press, pp292-299.

MIYAZAKI Y, TAKANOU A, NOZAKI H, NAKAGAWA N and OKADA K (1991) "Method to estimate parameter values in software prediction models." *Information and Software Technology*, **33**(3), 239-243.

MOHANTY S N (1981) "Software cost estimation: Present and future." *Software - Practice and Experience*, **11**, 103-121.

MÖLLER K H and PAULISH D J (1993) "*Software metrics: A practitioner's guide to improved product development.*" Chapman & Hall, London.

MORGAN D G, SHORTER D N and TAINSH M (1988) "Systems engineering: A strategy for the improved design and construction of complex IT systems." Report published by the *Information Engineering Directorate*, Kingsgate House, 66-74 Victoria Street, London.

MURDOCH S T (1987) "A review of the intersection of expert systems and database systems: Expert/database systems." *International Journal of Systems Research and Information Science*, **2**, 111-119.

MUSA J D (1979) "Software reliability measures applied to system engineering." In *AFIPS Conference Proceedings of the 1979 National Computer Conference*, New York, 4-7 June, 1979, pp941-946.

MUSA J D, IANNINO A and OKUMOTO K (1987) "*Software Reliability: Measurement, Predication, Application.*" McGraw-Hill, New York.

MYERS G J (1977) "An extension to the cyclomatic measure of program complexity." *ACM SIGPLAN Notices*, **12**(10), 61-64.

MYERS M and KAPOSI A A (1991) "Modelling and measurement of Prolog data." *Software Engineering Journal*, **November**, 413-434.

NELSON E A (1966) "Management handbook for the estimation of computer programming costs." *Report AD-A648750*, Systems Development Corporation (31 October, 1966).

OLLE T W, HAGELSTEIN J, MacDONALD I G, ROLLAND C, SOL H G, van ASSCHE F J and VERRIJN-STUART A A (1988) "*Information Systems Methodologies: A Framework for Understanding.*" Addison-Wesley, Wokingham.

OULSNAM G (1979) "Cyclomatic numbers do not measure complexity of unstructured programs." *Information Processing Letters*, **9**(5), 207-211.

OVIEDO E I (1980) "Control flow, data flow and program complexity." In *Proceedings, IEEE Computer Software and Applications Conference*, November, 1980, pp146-152.

PARTRIDGE D (1990) "Artificial intelligence and software engineering: A survey of possibilities." In D.Ince and D.Andrews (Eds.), *op cit.*, pp375-385.

PARTRIDGE D and WILKS Y (1987) "Does AI have a methodology which is different from software engineering?" *AI Review*, **1**(2), 111-120.

PERLIS A J, SAYWARD F G and SHAW M (1981) "*Software Metrics* (Eds.)." The MIT Press, Cambridge, MA.

PFLEEGER S L (1991) "Model of software effort and productivity." *Information and Software Technology*, **33**(3), 224-231.

PFLEEGER S L (1993) "Lessons learned in building a corporate metrics program." *IEEE Software*, **May**, 67-74.

PHAN D, VOGEL D and NUNAMAKER J (1988) "The search for perfect project management." *Computerworld*, **September**, 95-100.

PIOWARSKI P (1982) "A nesting level complexity measure." *ACM SIGPLAN Notices*, **17**(9), 40-50.

PLANT R T (1991) "Factors in software quality for knowledge-based systems." *Information & Software Technology*, **33**(7), 527-536.

PORTER A A and SELBY R W (1990) "Empirically guided software development using metric-based classification trees." *IEEE Software*, 7(2), 46-54.

PRATHER R E (1984) "An axiomatic theory of software complexity measure." *The Computer Journal*, **27**(4), 340-347.

PRATHER R E (1987) "On hierarchical software metrics." *Software Engineering Journal*, **2**(2), 42-45.

PRATHER R E (1988) "Comparison and extension of theories of Zipf and Halstead." *The Computer Journal*, **31**(3), 248-252.

PREECE A D (1990) "Towards a methodology for evaluating expert systems." *Expert Systems*, 7(4), 215-223.

PRERAU D S (1985) "Selection of an appropriate domain for an expert system." *The AI Magazine*, **6**(2), 26-30.

PUTNAM L H (1978) "A general empirical solution to the macro software and estimating problem." *IEEE Transactions on Software Engineering*, **SE-4**(4), 345-361.

PUTNAM L H (1991) "Trends in measurement, estimation, and control." *IEEE Software*, **March**, 105-107.

PUTNAM L H and PUTNAM D T (1984) "A data verification of the software fourth power trade-off law." In *Proceedings, 6th International Society of Parametric Analysis*, pp443-471.

PUTNAM L H, PUTNAM D T and THAYER L P (1983) "A method to measure the 'effective productivity' in building software systems." In *Proceedings, 5th International Society of Parametric Analysis*, pp95-143.

RAMSEY C L and BASILI V R (1989) "An evaluation of expert systems for software engineering management." *IEEE Transactions on Software Engineering*, **SE-15**(6), 747-759.

READDIE M, STRENG K-H and WERMSER D (1989) "KADS metrication." ESPRIT Project 1098, *Report SD-G10-001*. Produced by SD(Europe) Ltd. & NTE NeuTech Entwicklungsgesellschaft + Co. KG.

ROGET 91. *Register of Graduate Employment and Training*. Published for Higher Education careers offices by Central Services Unit, Armstrong House, Oxford Road, Manchester.

ROYCE W W (1970) "Managing the development of large software systems." In *Proceedings, WESCON70*, Los Angeles, 25-28 August, 1970, pp1-9.

RUBIN H A (1983) "Macro and micro-estimation of software development parameters: The ESTIMACS system." In *Proceedings, SOFTFAIR: A conference on software development tools, techniques and alternatives*, pp109-118.

RUBIN H A (1985) "A comparison of cost estimation tools (A panel session)." In *Proceedings, 8th International Conference on Software Engineering*, IEEE Computer Society Press, pp174-180.

SALT N F (1982) "Defining software science counting strategies." *ACM SIGPLAN Notices*, **17**(3), 58-67.

SANSOM W B, NEVILL D G and DOGARD P I (1987) "Predictive software metrics based on a formal specification." *Information and Software Technology*, **29**(5), 242-248.

SCHNEIDEWIND N F (1979) "Software metrics for aiding program development and debugging." In *AFIPS Conference Proceedings of the 1979 National Computer Conference*, New York, 4-7 June, 1979, pp989-994.

SCHNEIDEWIND N F (1992) "Methodology for validating software metrics." *IEEE Transactions on Software Engineering*, **SE-18**(5), 410-421.

SHEN V Y, CONTE S D and DUNSMORE H E (1983) "Software science revisited: A critical analysis of the theory and its empirical support." *IEEE Transactions on Software Engineering*, **SE-9**(2), 155-165.

SHEPPERD M J (1988) "A critique of cyclomatic complexity as a software metric." *Software Engineering Journal*, **3**(2), 30-36.

SHEPPERD M J (1991) "System architecture metrics: An evaluation." *PhD thesis*, The Open University.

SHEPPERD M J and INCE D C (1991) "Software metrics in software engineering and artificial intelligence." *Technical Report No. 91/11*, Faculty of Mathematics, The Open University. Also published in *International Journal of Software Engineering and Knowledge Engineering*, **1**(4), 463-476.

SHORTLIFFE E H (1976) "*Computer-Based Medical Consultations: MYCIN.*" Elsevier North-Holland, New York, NY.

SIMON H A (1974) "How big is a chunk?" *Science*, **183**, 482-488.

SNEED H M (1989) "*Software Engineering Management.*" Ellis-Horwood, Chichester.

SOONG N L (1977) "A program stability measure." In *AFIPS Conference Proceedings of the 1977 National Computer Conference*, pp163-173.

SOORGARD P (1991) "Evaluating expert system prototypes." *AI & Society*, **5**, 3-17.

SPENCE I T A and CAREY B N (1991) "Customers do not want frozen specifications." *Software Engineering Journal*, **6**(3), 175-180.

STETTER F (1984) "A measure of program complexity." *Computer Languages*, **9**, 203-210.

SYMONS C R (1988) "Function point analysis: Difficulties and improvements." *IEEE Transactions on Software Engineering*, **SE-14**(1), 2-11.

SYMONS C R (1991) "*Software Sizing and Estimating. Mk.II FPA (Function Point Analysis).*" John Wiley & Sons, Chichester.

TAUSWORTHE R C (1981) "Deep space network software cost estimation model." *Publication 81-7*, Jet Propulsion Laboratory, Pasadena, CA.

THEBAUT S M and SHEN V Y (1984) "An analytical resource model for large-scale software development." *Information Processing & Management*, **20**(1-2), 293-315.

TRACZ W J (1979) "Computer programming and the human thought process." *Software - Practice and Experience*, **9**, 127-137.

UNDERHILL L H (1963) "The growth of complexity of a general-purpose program." *The Computer Journal*, **6**(April 1963-January 1964), 37-38.

van de BRUG A, BACHANT J and McDERMOTT J (1986) "The taming of R1." *IEEE Expert*, **Fall**, 33-39.

van GENUCHTEN M and KOOLEN H (1991) "On the use of software cost models." *Information & Management*, **21**, 37-44.

VERYARD R (1991) "*The Economics of Information Systems and Software* (Ed.)." Butterworth-Heinemann Ltd., Oxford.

VERNER J and TATE G (1992) "A software size model." *IEEE Transactions on Software Engineering*, **SE-18**(4), 265-278.

VESSEY I (1987) "On matching programmers' chunks with program structures: An empirical investigation." *International Journal of Man-Machine Studies*, **27**, 65-89.

WALSTON C E and FELIX C P (1977) "A method of programming measurement and estimation." *IBM Systems Journal*, **16**(1), 54-73.

WATERMAN D A (1986) "*A Guide to Expert Systems.*" Addison-Wesley, Reading, MA.

WEINBERG G M and SCHULMAN E L (1974) "Goals and performance in computer programming." *Human Factors*, **16**(1), 70-77.

WEITZEL J R and KERSCHBERG L (1989) "Developing knowledge based systems: Re-organising the system development cycle." *Communications of the ACM*, **32**(4), 482-488.

WEYUKER E J (1988) "Evaluating software complexity measures." *IEEE Transactions on Software Engineering*, **SE-14**(9), 1357-1365.

WOODWARD M R (1984) "The application of Halstead's software science theory to Algol 68 programs." *Software - Practice and Experience*, **14**, 263-276.

WOODWARD M R, HENNELL M A and HEDLEY D A (1979) "A measure of control flow complexity in program test." *IEEE Transactions on Software Engineering*, **SE-5**(1), 45-50.

WOLVERTON R W (1974) "The cost of developing large-scale software." *IEEE Transactions on Computers*, **C-23**(6), 615-636.

YAU S S and COLLOFELLO J S (1980) "Some stability measures for software maintenance." *IEEE Transactions on Software Engineering*, **SE-6**(6), 545-552.

YOULL D P (1990) "*Making Software Development Visible: Effective Project Control.*" John Wiley & Sons, Chichester.

ZUSE H and BOLLMANN P (1989) "Software metrics - Using measurement theory to describe the properties and scales of static software complexity metrics." *ACM SIGPLAN Notices*, **24**(8), 23-33.

APPENDIX A

1991 Survey Form

The original survey form used three colours of paper. The front and back cover was in white. The Part A section was in yellow. The Part B section was in green. The reasoning behind this colour scheming is that if darker paper produces less glare, colouring the survey form would be less hostile to the eyes of prospective respondents. Because of the poor quality of microfisching darker paper, however, the survey form is presented here using only white paper.

ASTON UNIVERSITY

METRICS AND SYSTEM DESCRIPTION SURVEY

ALL INFORMATION SUPPLIED IS CONFIDENTIAL

There are two parts to the survey: PART 1 deals with the size of your company (so similar companies can be grouped) and the experience of estimating and using support tools or their equivalent; PART 2 is more speculative and an explanation of the purpose of this section is given on page 10 of the survey. Please read this page carefully before answering PART 2.

PART 1: The Use of Metrics and Other Support Tools in Large Companies

Q1. *Company name*

Q2. *Which structured (documented) methodologies are used within the department?*

Q3. *How many people are there in your department involved in software development?*

Number of people

1-50	51-100	101-150	151-200	201-250	251-300	301+
☐	☐	☐	☐	☐	☐	☐

Q4. *What percentage of development man-years are currently spent on knowledge-based (or expert) system development?*

% man-years on KBS

0	1-10	11-25	26-50	51-75	76-90	91+
☐	☐	☐	☐	☐	☐	☐

Q5. *Is this figure increasing, decreasing or about the same as previous years?*

Increasing	Decreasing	About the same
☐	☐	☐

Q6. *Do you see project estimation as a problem?*

Yes	No
☐	☐

Q7. *Have you developed any Software Cost Estimating tools in-house?*

Yes	No
☐	☐

Q8. *Have you heard of any of the following Software Cost Estimating tools?*

	No	Yes →	*Have you evaluated this tool?* No	Yes →	*Do you use this tool?* No	Yes
Before You Leap	☐	☐	☐	☐	☐	☐
GECOMO PLUS	☐	☐	☐	☐	☐	☐
SOFTCOST - R	☐	☐	☐	☐	☐	☐
SEER	☐	☐	☐	☐	☐	☐
BIS / ESTIMATOR	☐	☐	☐	☐	☐	☐
CHECKMARK	☐	☐	☐	☐	☐	☐
PRICE - S	☐	☐	☐	☐	☐	☐
SLIM	☐	☐	☐	☐	☐	☐
ESTIMATICS	☐	☐	☐	☐	☐	☐
PC - COMO	☐	☐	☐	☐	☐	☐
COSTAR	☐	☐	☐	☐	☐	☐
SYSTEM - 3	☐	☐	☐	☐	☐	☐
SIMPACT ESTIMATOR	☐	☐	☐	☐	☐	☐

Q9. *Are there any other Cost Estimating tools you know of or use?* (Please underline those you use.)

No Yes → *Which ones?*

☐ ☐

Q10. *What is your opinion of Software Cost Estimating tools (e.g., are they useful, relevant, etc.)?*

Q11. *What percentage error (+ or - %) would you assign to the following estimates?*

An **EXCELLENT** estimate would be within + / - ☐ %

A **GOOD** estimate would be within + / - ☐ %

An **ADEQUATE** estimate would be within + / - ☐ %

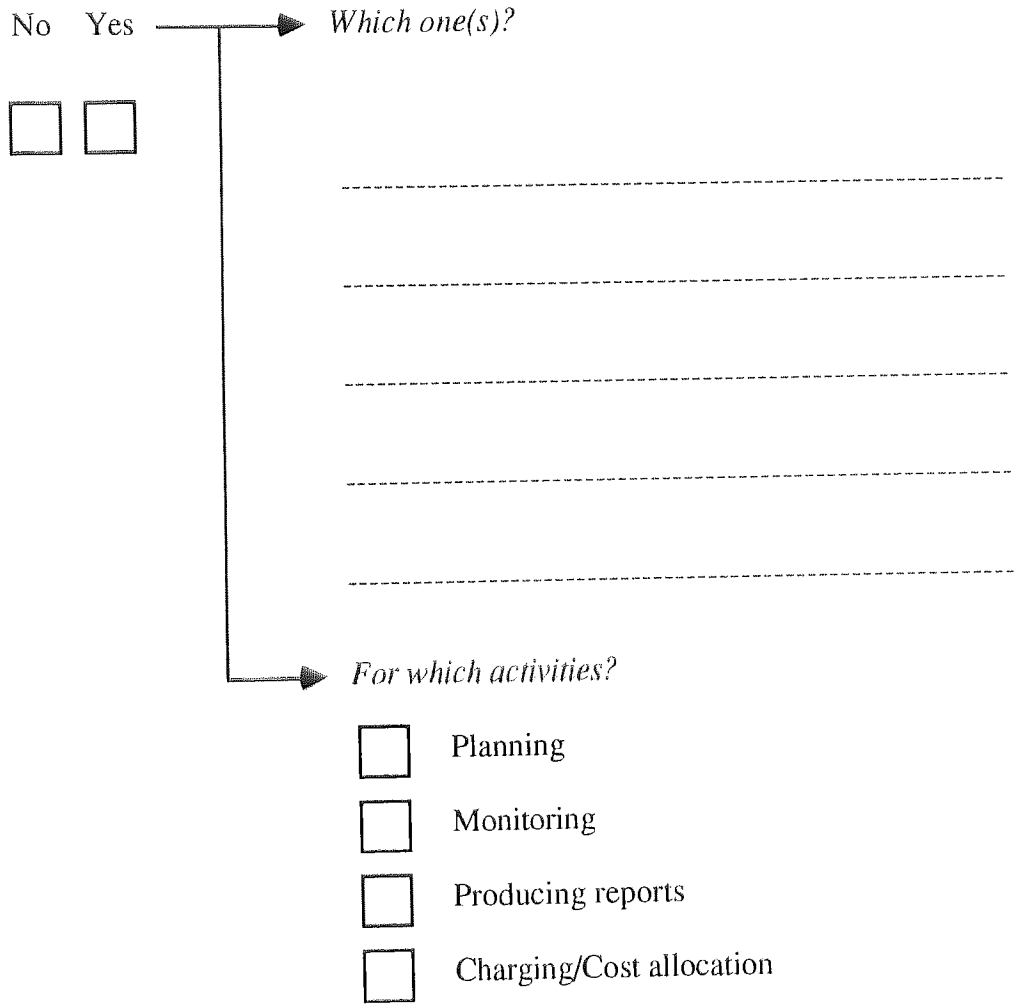
A **POOR** estimate would be within + / - ☐ %

A **BAD** estimate would be within + / - ☐ %

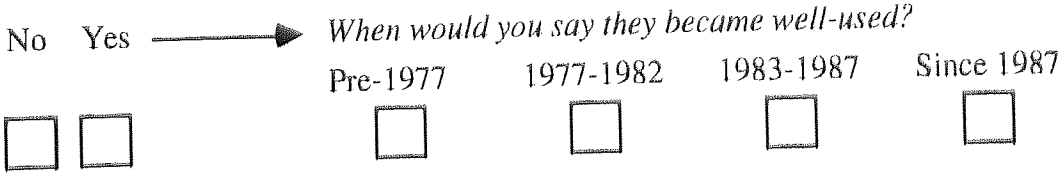
Q12. *What accuracy of estimate for project cost and/or duration would be expected from a manager if the project was:*

	Excellent	Good	Adequate	Poor	Bad
Entirely new and complex	☐	☐	☐	☐	☐
New, but not complicated	☐	☐	☐	☐	☐
Maintenance or re-use of system code	☐	☐	☐	☐	☐

Q.13 *Do you use a Project Management tool (e.g., PMW, Instaplan, etc)?*



Q14. *Would you say Project Management tools are commonly used?*



Q15. *How often do you collect any of the following types of data during the development of a project?*

	All of the time	Most of the time	Occas-sionaly	Rarely	Never
Number of errors/bugs discovered in the system	☐	☐	☐	☐	☐
Type of error (system crash, incorrect output, incorrect functionality, etc)	☐	☐	☐	☐	☐
Man-hours	☐	☐	☐	☐	☐
Type of time expenditure (i.e., distinguishing between meetings, interviews, programming, etc)	☐	☐	☐	☐	☐
Productivity rates, in terms of:					
- lines of code	☐	☐	☐	☐	☐
- number of rules generated (KBS)	☐	☐	☐	☐	☐
- number of functions completed	☐	☐	☐	☐	☐

Q16. What other types of data do you normally collect?

Q17. *Once the initial systems requirements specification has been agreed, how often are there changes to these requirements during development?*

Never	Rarely	Occas-sionaly	Most of the time	All of the time
☐	☐	☐	☐	☐

Q18. *What are the major reasons for projects failing?* In terms of:

a) Time/money

---

b) Meeting user requirements

Q19. *Is there anything else which you think is important about Software Cost Estimation and support tools which has not been addressed by the above series of questions?*

PART 2: Recognising the Type of System from the System Description

This section speculates that the ability to estimate accurately depends on an understanding of the system proposed.

A number of descriptive words applicable to either KBS or conventional systems have been collected. If the words come from a description of system requirements at the analysis stage (e.g., submitted by a systems analyst), is the nature of the system to be developed recognisable at this early stage? If all specific information is omitted, we are left with the action word alone to represent what the system or process is meant to do. Rate the words in a range from 1 - 7, according to whether you believe the word was taken from the system specification of:

1 - probably a KBS project
3 - possibly a KBS project
4 - can be implemented as either
5 - possibly a conventional project
7 - probably a conventional project

2, 6 - intermediate values

As can be the case with system requirements, some of the words are more obscure than others. **Please try and rate all the words even if you have no experience of systems with these type of requirements.** Thank you.

Q.1 *System character*

The character of a system expresses what its overall purpose is meant to be (i.e., what function it has) and forms sentences like, "The following system will be used to ..... ". If the user completed the sentence with one of the following words, would you say the system sounded like a conventional or KBS project?

Please try and rate all the words even if you have no experience of systems with these type of requirements.

1 - probably KBS 4 - can be implemented as either 7 - probably conventional

	1	2	3	4	5	6	7
Advise	☐	☐	☐	☐	☐	☐	☐
Assist	☐	☐	☐	☐	☐	☐	☐
Teach	☐	☐	☐	☐	☐	☐	☐
Help	☐	☐	☐	☐	☐	☐	☐
Support	☐	☐	☐	☐	☐	☐	☐
Design	☐	☐	☐	☐	☐	☐	☐
Develop	☐	☐	☐	☐	☐	☐	☐
Build	☐	☐	☐	☐	☐	☐	☐
Plan	☐	☐	☐	☐	☐	☐	☐
Diagnose	☐	☐	☐	☐	☐	☐	☐
Classify	☐	☐	☐	☐	☐	☐	☐
Detect	☐	☐	☐	☐	☐	☐	☐
Control	☐	☐	☐	☐	☐	☐	☐
Monitor	☐	☐	☐	☐	☐	☐	☐
Maintain	☐	☐	☐	☐	☐	☐	☐

Q2. *System processes*

The processes of a system details a job of work (i.e., how the system will work) and forms sentences like, "The system will need to ..... (something or someone)". If the user completed the sentence with one of the following words, would you say the system sounded like a conventional or KBS project?

Please try and rate all the words even if you have no experience of systems with these type of requirements.

1 - probably KBS 4 - can be implemented as either 7 - probably conventional

	1	2	3	4	5	6	7
Invent	☐	☐	☐	☐	☐	☐	☐
Search	☐	☐	☐	☐	☐	☐	☐
Cooperate	☐	☐	☐	☐	☐	☐	☐
Transform	☐	☐	☐	☐	☐	☐	☐
Convert	☐	☐	☐	☐	☐	☐	☐
Specify	☐	☐	☐	☐	☐	☐	☐
Instruct	☐	☐	☐	☐	☐	☐	☐
Integrate	☐	☐	☐	☐	☐	☐	☐
Justify	☐	☐	☐	☐	☐	☐	☐
Tell	☐	☐	☐	☐	☐	☐	☐
Present	☐	☐	☐	☐	☐	☐	☐
Recognise	☐	☐	☐	☐	☐	☐	☐
Filter	☐	☐	☐	☐	☐	☐	☐
Differentiate	☐	☐	☐	☐	☐	☐	☐
Acquire	☐	☐	☐	☐	☐	☐	☐
Interpret	☐	☐	☐	☐	☐	☐	☐
Demonstrate	☐	☐	☐	☐	☐	☐	☐

Q3. *Is there anything else which you think is important about system requirements which needs to be taken into account?*

Q4. *Would you like to receive a copy of the summary of the results of this survey?*

No ☐ Yes ☐ → Contact name:

Postal address:

THANK YOU VERY MUCH FOR YOUR ASSISTANCE
IT IS GREATLY APPRECIATED

APPENDIX B

Summary of the PSA results files

Table	Title
B.1	LPA Prolog program structure metric data
B.2	LPA Prolog program size metric data

Table B.1 : LPA Prolog program structure metric data

(No.) Filename	v(G)	C_P	Errors	Programmer rating			LOC
				Understand	Debug	Test	
(1) USERS	91	15631	0	5	4	4	287
(2) UPMOD	53	289	0	7	8	6	122
(3) TOKENS	85	231573	1	7	7	6	164
(4) TESTSTGE	20	3	2	4	6	5	59
(5) SUPPROJ	38	466	1	3	2	2	57
(6) STGEOPTS	52	487	3	7	8	8	139
(7) STATUS	34	712	0	3	2	2	104
(8) STAGES	76	11784	4	8	8	8	266
(9) SPJINT	20	21	0	3	2	2	55
(10) REVQUEST	36	208	1	4	4	4	93
(11) RESTART	7	0	0	1	2	2	25
(12) REPSTGE	34	133	3	4	3	2	110
(13) REPORT	53	408	3	4	4	4	118
(14) READLST	16	1037	0	5	4	3	72
(15) READKEY	21	23	0	4	3	2	72
(16) READID	35	17727	0	6	5	4	172
(17) QUESTREP	60	2126	2	5	4	3	120
(18) QCAT	53	7250	3	6	7	7	258

(No.) Filename	v(G)	C_P	Errors	Programmer rating			LOC
				Understand	Debug	Test	
(19) PROJOPTS	58	57	1	4	5	4	120
(20) PROJDATA	68	352	0	5	6	5	145
(21) PRIM	1	0	1	1	2	1	35
(22) PORTPROJ	31	40	2	3	2	2	75
(23) PORTMOD	7	0	0	1	1	1	31
(24) POPUPS	75	112425	0	9	8	6	237
(25) PMWV3	35	456	4	3	3	2	57
(26) PMWV2	31	78	3	2	2	2	47
(27) PHASEPER	25	12	3	3	2	2	57
(28) NOTEPAD	41	55	0	3	3	3	96
(29) MAIN	38	56	2	2	3	2	103
(30) LOTUS	27	12	1	2	1	1	33
(31) LOGOUT	14	1	0	1	2	2	44
(32) IPFMOD	85	3546	1	4	3	4	213
(33) INTFC	31	308	2	3	4	3	110
(34) INSTA	20	11	1	2	1	1	47
(35) INPUT	83	124119	0	6	6	6	277
(36) IMPPROJ	94	17139	2	6	8	7	201
(37) IMPMOD	95	3517	3	4	4	4	180
(38) HEURS	32	5	0	3	3	3	88
(39) HELP	44	242	1	4	4	3	133

(No.) Filename	v(G)	C_P	Errors	Programmer rating			LOC
				Understand	Debug	Test	
(40) GETPATH	21	25	0	2	3	2	70
(41) GENFORM	14	10	0	4	3	2	38
(42) FPAMOD	77	1583	1	4	4	4	196
(43) FKEYS	22	175	0	2	3	3	108
(44) FCAT	54	6329	3	6	7	7	255
(45) EXPREP	73	4353	1	3	3	2	173
(46) EXPPROJ	59	3553	2	6	8	9	135
(47) EXPMOD	62	778	2	4	8	8	152
(48) EXPLO	39	16	0	2	2	1	82
(49) EXPHIST	36	36	0	3	3	1	82
(50) EXPFPA	39	4986	0	3	3	1	179
(51) EVAL	133	435538	3	4	4	3	444
(52) ESTREP	52	613	2	3	2	1	88
(53) ERROR	37	115	6	3	6	8	92
(54) EMODDEL	19	2	1	2	3	3	51
(55) EDITFPA	48	36368	0	-----	-----	-----	206
(56) EDITFORM	85	83644	3	7	7	7	218
(57) DATA	89	7937	5	8	10	10	215
(58) COPYMOD	44	255	2	4	5	6	111
(59) DBLMENU	39	9257	0	7	6	6	138
(60) COMPFORM	49	12114	2	7	7	6	175

(No.) Filename	v(G)	C_P	Errors	Programmer rating			LOC
				Understand	Debug	Test	
(61) CATS	179	171485	0	8	8	6	363
(62) BUILDMOD	141	26466	0	6	5	4	267
(63) FRMWRK	46	3209	0	5	5	5	130
(64) ESTIMATE	150	132716	5	10	10	10	300
(65) ADMIN	9	2	0	1	1	1	51
(66) UTILS	85	146641	4	5	5	5	272
(67) FPA	110	1474503	1	7	8	8	385
(68) ESTMOD	95	138953	3	7	7	7	330
(69) HISTORY	42	2015	0	5	4	4	131
(70) KEYBOARD	61	20190	0	3	3	2	206
(71) MAINMENU	54	4854	1	9	8	7	160
(72) MODULE	51	2054	2	7	8	8	163
(73) PROJREC	155	210488	9	9	10	8	326
(74) READNUM	60	17421	0	6	5	4	162
(75) RECORD	44	1887	4	4	4	6	131
(76) REPFILE	108	50758	3	6	5	4	280
(77) REPMOD	109	160312	0	7	7	7	311
(78) REPRTS	19	270	0	4	4	3	79
(79) RUNMENU	82	212831	0	9	8	5	295
(80) TPARSER	20	11	0	3	2	2	36

Table B.2 : LPA Prolog program size metric data

(No.) Filename	Software science						LPA Prolog elements								LOC
	η_1	η_2	N_1	N_2	N	N*hat*	P	P_T	O	O_T	I	I_T	U	U_T	
(1) USERS	93	144	1078	507	1585	1641	71	181	22	897	95	171	49	336	287
(2) UPMOD	57	97	545	301	846	973	44	90	13	455	53	78	44	223	122
(3) TOKENS	76	151	1545	990	2535	1568	54	216	22	1329	95	150	56	840	164
(4) TESTSTGE	31	61	222	120	342	515	21	36	10	186	51	94	10	26	59
(5) SUPPROJ	43	72	512	233	745	678	31	85	12	512	37	63	35	170	57
(6) STGEOPTS	52	108	643	359	1002	1025	38	94	14	549	49	76	59	283	139
(7) STATUS	40	80	430	200	630	719	30	81	10	349	59	131	21	69	104
(8) STAGES	90	226	1204	680	1884	2352	73	186	17	1018	167	321	59	359	266
(9) SPJINT	38	58	274	137	411	539	23	41	15	233	36	59	22	78	55
(10) REVQUEST	51	75	414	200	614	756	39	68	12	346	44	64	31	136	93
(11) RESTART	20	19	64	28	92	167	8	9	12	83	17	25	2	3	25
(12) REPSTGE	44	71	502	247	749	677	29	92	15	410	52	91	19	156	110
(13) REPORT	57	80	550	269	819	838	45	99	15	451	43	80	37	189	118
(14) READLST	33	61	508	309	817	528	17	53	16	455	27	68	34	241	72
(15) READKEY	33	50	292	148	440	449	19	45	14	247	26	44	24	104	72
(16) READID	48	86	994	616	1610	820	28	129	20	765	47	176	39	440	172
(17) QUESTREP	69	110	681	342	1023	1167	53	124	16	557	62	92	48	250	120
(18) QCAT	99	209	979	562	1541	2267	85	170	14	809	162	292	47	270	258

(No.) Filename	Software science						LPA Prolog elements								LOC
	η_1	η_2	N_1	N_2	N	N*hat*	P	P_T	O	O_T	I	I_T	U	U_T	
(19) PROJOPTS	50	79	438	192	630	780	37	92	13	346	48	77	29	115	120
(20) PROJDATA	72	115	672	307	979	1231	58	137	14	535	81	142	34	165	145
(21) PRIM	24	100	410	267	677	774	4	52	20	410	77	243	23	24	35
(22) PORTPROJ	45	50	347	177	524	529	37	67	8	280	40	56	10	121	75
(23) PORTMOD	21	31	115	55	170	246	14	20	7	95	27	38	4	17	31
(24) POPUPS	86	162	1677	1118	2795	1742	63	175	23	1502	37	114	125	1004	237
(25) PMWV3	50	71	496	258	754	719	34	83	16	413	40	81	31	177	57
(26) PMWV2	46	53	364	192	556	558	31	62	15	302	34	57	19	135	47
(27) PHASEPER	37	46	260	140	400	447	33	40	4	220	26	38	20	102	57
(28) NOTEPAD	51	63	364	171	535	666	33	71	18	283	46	82	17	89	96
(29) MAIN	57	100	426	193	619	999	40	78	17	348	73	105	27	88	103
(30) LOTUS	38	39	263	117	380	406	27	52	11	211	24	35	15	82	33
(31) LOGOUT	29	39	156	76	232	347	18	22	11	134	29	44	10	32	44
(32) IPFMOD	89	160	785	384	1169	1748	73	145	16	640	120	191	40	193	213
(33) INTFC	56	118	502	255	757	1137	41	77	15	425	96	148	22	107	110
(34) INSTA	39	55	245	134	379	524	26	39	13	206	32	48	23	86	47
(35) INPUT	91	159	1374	761	2135	1755	72	186	19	1188	97	205	62	556	277
(36) IMPPROJ	105	162	940	445	1385	1894	91	186	14	754	76	117	86	328	201
(37) IMPMOD	85	153	940	495	1435	1655	67	174	18	766	65	92	88	403	180
(38) HEURS	46	73	320	168	488	706	36	60	10	260	55	89	18	79	88
(39) HELP	59	96	516	274	790	979	43	85	16	431	73	103	23	171	133

(No.) Filename	Software science						LPA Prolog elements								LOC
	η_1	η_2	N_1	N_2	N	N*hat*	P	P_T	O	O_T	I	I_T	U	U_T	
(40) GETPATH	38	54	252	133	385	510	24	38	14	214	30	48	24	85	70
(41) GENFORM	30	48	238	139	377	415	15	25	15	213	21	31	27	108	38
(42) FPAMOD	88	152	728	361	1089	1670	72	134	16	594	123	217	29	144	196
(43) FKEYS	37	86	401	204	605	745	24	52	13	349	72	150	14	54	108
(44) FCAT	100	202	956	531	1487	2211	86	159	14	797	159	279	43	252	255
(45) EXPREP	76	125	880	428	1308	1346	66	161	10	719	74	118	51	310	173
(46) EXPPROJ	77	115	650	302	952	1270	67	131	10	519	54	80	61	222	135
(47) EXPMOD	62	103	703	344	1047	1058	57	139	5	564	59	102	44	242	152
(48) EXPLO	42	62	367	170	537	596	31	74	11	293	38	60	24	110	82
(49) EXPHIST	42	58	355	185	540	566	28	59	14	296	35	71	23	114	82
(50) EXPFPA	47	141	775	387	1162	1268	33	155	14	620	96	230	45	157	179
(51) EVAL	157	194	2103	934	3037	2620	134	387	23	1716	102	174	72	760	444
(52) ESTREP	60	91	571	264	835	947	46	124	14	447	63	110	28	154	88
(53) ERROR	64	134	587	284	871	1331	40	107	24	480	104	144	30	140	92
(54) EMODDEL	32	42	159	73	232	386	21	28	11	131	31	37	11	36	51
(55) EDITFPA	60	142	1176	639	1815	1370	39	205	21	971	98	375	44	264	206
(56) EDITFORM	100	167	1228	702	1930	1897	80	185	20	1043	79	180	88	522	218
(57) DATA	80	147	1262	543	1805	1564	64	248	16	1014	98	278	49	265	215
(58) COPYMOD	56	87	455	233	688	886	42	79	14	376	43	74	44	159	111
(59) DBLMENU	51	129	765	441	1206	1194	38	97	13	668	85	154	44	287	138
(60) COMPFORM	59	136	820	421	1241	1311	42	137	17	683	88	169	48	252	175

(No.) Filename	Software science						LPA Prolog elements								LOC
	η_1	η_2	N_1	N_2	N	N*hat*	P	P_T	O	O_T	I	I_T	U	U_T	
(61) CATS	120	201	1868	951	2819	2367	103	209	17	1659	124	234	77	717	363
(62) BUILDMOD	113	184	1340	669	2009	2155	96	242	17	1098	108	205	76	464	267
(63) FRMWRK	65	107	627	339	966	1113	50	96	15	531	60	102	47	237	130
(64) ESTIMATE	123	199	1792	851	2643	2374	105	347	18	1445	111	217	88	634	300
(65) ADMIN	23	69	205	111	316	526	12	23	11	182	65	94	4	17	51
(66) UTILS	111	194	1487	764	2251	2229	90	229	21	1258	82	146	112	618	272
(67) FPA	100	235	2166	1195	3361	2515	76	352	24	1814	148	565	87	630	385
(68) ESTMOD	101	284	1534	785	2319	2987	85	235	16	1299	219	371	65	414	330
(69) HISTORY	58	114	616	344	960	1119	40	81	18	535	69	133	45	211	131
(70) KEYBOARD	84	182	309	538	847	1903	64	133	23	840	120	220	62	318	206
(71) MAINMENU	62	118	639	318	957	1181	47	109	15	530	74	140	44	178	160
(72) MODULE	64	92	780	357	1137	984	46	132	18	648	55	93	37	264	163
(73) PROJREC	125	203	2086	1029	3115	2427	110	382	15	1704	94	201	109	828	326
(74) READNUM	64	96	1027	549	1576	1016	40	154	24	873	51	144	45	405	162
(75) RECORD	62	81	553	271	824	883	45	98	17	455	47	94	34	177	131
(76) REPFILE	83	139	1213	541	1754	1518	62	228	21	985	106	274	33	267	280
(77) REPMOD	71	159	1634	829	2463	1599	56	294	15	1340	107	278	52	551	311
(78) REPRTS	40	90	395	213	608	797	29	63	11	332	73	125	17	88	79
(79) RUNMENU	70	165	2111	1280	3391	1644	51	312	19	1799	49	221	116	1059	295
(80) TPARSER	34	34	252	96	348	346	20	43	14	209	17	19	17	77	36

APPENDIX C

LPA Prolog structure analysis tables

Table C.1 : Summaries of the three metrics

Metric	Minimum value	Maximum value	Step, I
Lines of code, LOC	25	444	5
Cyclomatic complexity, v(G)	1	179	5
Data-flow fan-in/out, C_P	0	14745	5

Table C.2 : Step-wise analysis of LOC thresholds

Threshold	N_T	E	$S_T(T)$	$Prop_{Right}$	$Prop_{Wrong}$
0	80	46	1.000	0.575	-----
30	79	46	1.000	0.582	-----
35	77	45	0.978	0.584	0.333
40	74	44	0.957	0.595	0.333
45	73	44	0.957	0.603	0.286
50	71	42	0.913	0.592	0.444
55	69	41	0.891	0.594	0.455
60	64	37	0.804	0.578	0.563
75	61	37	0.804	0.607	0.474
80	59	36	0.783	0.610	0.476

Table C.3 : Step-wise analysis of v(G) thresholds

Threshold	N_T	E	$S_T(T)$	$Prop_{Right}$	$Prop_{Wrong}$
0	80	46	1.000	0.575	-----
5	79	45	0.978	0.570	1.000
10	76	45	0.978	0.592	0.250
15	74	45	0.978	0.608	0.167
20	71	44	0.957	0.620	0.222
25	64	42	0.913	0.656	0.250
30	62	40	0.870	0.645	0.333
35	56	36	0.783	0.643	0.417

Table C.4 : Step-wise analysis of C_p thresholds

Threshold	N_T	E	$S_T(T)$	$Prop_{Right}$	$Prop_{Wrong}$
0	80	46	1.000	0.575	-----
5	72	43	0.935	0.597	0.375
10	71	43	0.935	0.606	0.333
15	67	40	0.870	0.597	0.462
20	66	40	0.870	0.606	0.429
25	63	40	0.870	0.635	0.353
40	61	39	0.848	0.639	0.368
55	60	39	0.848	0.650	0.350
60	58	37	0.804	0.638	0.409
80	57	36	0.783	0.632	0.435

Table C.5 : Combination of metrics using threshold values at 80% of programs with errors detected (LOC=75; v(G)=30; C_P=75)

Combination	N_T	E	$S_T(T)$	$Prop_{Right}$	$Prop_{Wrong}$
LOC and v(G) and C_P	52	34	0.739	0.654	0.429
LOC and v(G)	58	36	0.783	0.621	0.455
LOC and C_P	54	34	0.739	0.630	0.462
v(G) and C_P	55	37	0.804	0.673	0.360
LOC or v(G) or C_P	65	40	0.870	0.615	0.400
LOC or v(G)	64	40	0.870	0.625	0.375
LOC or C_P	64	39	0.848	0.609	0.438
v(G) or C_P	65	40	0.870	0.615	0.400
LOC and (v(G) or C_P)	60	36	0.783	0.600	0.500
(LOC and v(G)) or C_P	62	39	0.848	0.629	0.389
LOC or (v(G) and C_P)	63	39	0.848	0.619	0.412
(LOC or v(G)) and C_P	57	37	0.804	0.649	0.391
(LOC and C_P) or v(G)	62	39	0.848	0.629	0.389
(LOC or C_P) and v(G)	61	39	0.848	0.639	0.368

Table C.6 : Combination of metrics using threshold values at the highest $Prop_{Right}$ for each metric (LOC=80; v(G)=25; C_P=55)

Combination	N_T	E	$S_T(T)$	$Prop_{Right}$	$Prop_{Wrong}$
LOC and v(G) and C_P	54	36	0.783	0.667	0.385
LOC and v(G)	58	36	0.783	0.621	0.455
LOC and C_P	55	36	0.783	0.655	0.400
v(G) and C_P	57	39	0.848	0.684	0.304
LOC or v(G) or C_P	66	41	0.891	0.621	0.357
LOC or v(G)	64	41	0.891	0.641	0.313
LOC or C_P	64	39	0.848	0.609	0.438
v(G) or C_P	66	41	0.891	0.621	0.357
LOC and (v(G) or C_P)	59	36	0.783	0.610	0.476
(LOC and v(G)) or C_P	63	39	0.848	0.619	0.412
LOC or (v(G) and C_P)	62	39	0.848	0.629	0.389
(LOC or v(G)) and C_P	58	39	0.848	0.672	0.318
(LOC and C_P) or v(G)	64	41	0.891	0.641	0.313
(LOC or C_P) and v(G)	61	39	0.848	0.639	0.368

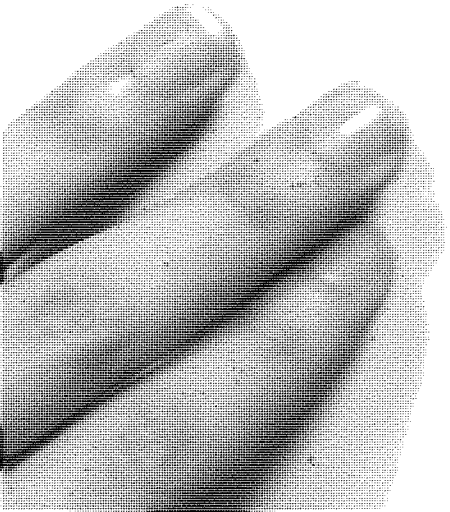


Table C.7 : Combination of metrics using threshold values at the lowest $Prop_{Wrong}$ for each metric (LOC=45; v(G)=15; C_P=10)

Combination	N_T	E	$S_T(T)$	$Prop_{Right}$	$Prop_{Wrong}$
LOC and v(G) and C_P	69	42	0.913	0.609	0.364
LOC and v(G)	72	44	0.957	0.611	0.250
LOC and C_P	69	42	0.913	0.609	0.364
v(G) and C_P	71	43	0.935	0.606	0.333
LOC or v(G) or C_P	75	45	0.978	0.600	0.200
LOC or v(G)	75	45	0.978	0.600	0.200
LOC or C_P	75	45	0.978	0.600	0.200
v(G) or C_P	74	45	0.978	0.608	0.167
LOC and (v(G) or C_P)	72	44	0.957	0.611	0.250
(LOC and v(G)) or C_P	74	45	0.978	0.608	0.167
LOC or (v(G) and C_P)	75	45	0.978	0.600	0.200
(LOC or v(G)) and C_P	71	43	0.935	0.606	0.333
(LOC and C_P) or v(G)	74	45	0.978	0.608	0.167
(LOC or C_P) and v(G)	74	45	0.978	0.608	0.167

Table C.8 : Sensitivity analysis of the best threshold values for v(G) and C_P

v(G) threshold	C_P threshold	N_T	E	$S_T(T)$	$Prop_{Right}$	$Prop_{Wrong}$
20	50	59	39	0.848	0.661	0.333
25	50	58	39	0.848	0.672	0.318
30	50	58	39	0.848	0.672	0.318
20	55	58	39	0.848	0.672	0.318
25	55	57	39	0.848	0.684	0.304
30	55	57	39	0.848	0.684	0.304
20	60	56	37	0.804	0.661	0.375
25	60	55	37	0.804	0.673	0.360
30	60	55	37	0.804	0.673	0.360
25	55	57	39	0.848	0.684	0.304
26	55	57	39	0.848	0.684	0.304
27	55	57	39	0.848	0.684	0.304
28	55	57	39	0.848	0.684	0.304
29	55	57	39	0.848	0.684	0.304
30	55	57	39	0.848	0.684	0.304

APPENDIX D

LPA Prolog size analysis tables

Table	Title
D.1	Logarithmic and linear models of program length (N) using η_1 and η_2
D.2	Performance of the N models using η_1 and η_2
D.3	Models of lines of code (LOC) using η_1 and η_2
D.4	Performance of the LOC models using η_1 and η_2
D.5	Regression models for N based on P, O_T, I, I_T and U_T
D.6	Regression models for LOC based on P, O_T, I, I_T and U_T
D.7	Values of P, I_T and U_T bands
D.8	Performance of the 5 and 7 banded models using P, I_T and U_T

Table D.1 : Logarithmic and linear models of program length (N) using η_1 and η_2

Form	Model	Coeff.Corr.	Coeff.Det.
1	$\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2$	0.854	0.730
2	$1.05*(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)-74.80$	0.856	0.732
2	$1.00*(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)$	0.854	0.730
3	$1.42(\eta_1.\log_2\eta_1)+0.89(\eta_2.\log_2\eta_2)-92.82$	0.857	0.734
3	$1.27(\eta_1.\log_2\eta_1)+0.88(\eta_2.\log_2\eta_2)$	0.855	0.731
4	$10.53\eta_1+7.55\eta_2-356.22$	0.857	0.735
4	$6.08\eta_1+7.40\eta_2$	0.839	0.703

Table D.2 : Performance of the N models using η_1 and η_2

Model	N*hat*=	MMRE	Pred		
			0.20	0.25	0.30
1	$\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2$	0.25	0.49	0.60	0.65
2	$1.05*(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)-74.80$	0.22	0.59	0.69	0.73
3	$1.00*(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)$	0.25	0.49	0.60	0.65
4	$1.42(\eta_1.\log_2\eta_1)+0.89(\eta_2.\log_2\eta_2)-92.82$	0.21	0.59	0.65	0.71
5	$1.27(\eta_1.\log_2\eta_1)+0.88(\eta_2.\log_2\eta_2)$	0.27	0.49	0.55	0.64
6	$10.53\eta_1+7.55\eta_2-356.22$	0.24	0.54	0.64	0.75
7	$6.08\eta_1+7.40\eta_2$	0.37	0.29	0.41	0.46

Table D.3 : Models of lines of code (LOC) using η_1 and η_2

LOC*hat*=	Coeff.Corr.	Coeff.Det.
$0.13*(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)-5.92$	0.930	0.866
$0.13*(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)$	0.930	0.865
$0.22(\eta_1.\log_2\eta_1)+0.1(\eta_2.\log_2\eta_2)-10.20$	0.936	0.876
$0.21(\eta_1.\log_2\eta_1)+0.1(\eta_2.\log_2\eta_2)$	0.934	0.873
$1.68\eta_1+0.80\eta_2-44.92$	0.935	0.874
$1.12\eta_1+0.785\eta_2$	0.915	0.838
$3.09\eta_1-43.43$	0.907	0.822
$2.51\eta_1$	0.888	0.788
$1.54\eta_2-21.50$	0.904	0.818
$1.39\eta_2$	0.899	0.807

Table D.4 : Performance of the LOC models using η_1 and η_2

Model	LOC*hat*=	MMRE	Pred		
			0.20	0.25	0.30
1	$0.13*(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)-5.92$	0.20	0.66	0.76	0.85
2	$0.13*(\eta_1.\log_2\eta_1+\eta_2.\log_2\eta_2)$	0.20	0.66	0.75	0.83
3	$0.22(\eta_1.\log_2\eta_1)+0.1(\eta_2.\log_2\eta_2)-10.20$	0.20	0.66	0.75	0.83
4	$0.21(\eta_1.\log_2\eta_1)+0.1(\eta_2.\log_2\eta_2)$	0.22	0.68	0.71	0.76
5	$1.68\eta_1+0.80\eta_2-44.92$	0.21	0.68	0.75	0.80
6	$1.12\eta_1+0.785\eta_2$	0.29	0.54	0.64	0.70
7	$3.09\eta_1-43.43$	0.24	0.58	0.69	0.74
8	$2.51\eta_1$	0.34	0.48	0.51	0.63
9	$1.54\eta_2-21.50$	0.23	0.56	0.70	0.81
10	$1.39\eta_2$	0.26	0.60	0.69	0.74

Table D.5 : Regression models for N based on P, O_T, I, I_T and U_T

Based on	N*hat*=	Coeff.Corr	Coeff.Det
P	24.26P-5.10	0.781	0.610
P+I	21.69P+2.53I-59.54	0.786	0.618
P+O_T+I_T+U_T	0.42P+1.26O_T+0.83I_T+0.85U_T-6.99	0.995	0.989
	0.30P+1.27O_T+0.80I_T+0.84U_T	0.995	0.989
P+O_T+I_T	-0.84P+1.87O_T-0.03I_T+5.50	0.994	0.988
P+O_T+U_T	-0.52P+1.72O_T+0.25U_T+11.94	0.994	0.988
P+I_T+U_T	3.68P+2.65I_T+2.50U_T-38.29	0.991	0.982
	3.14P+2.59I_T+2.52U_T	0.991	0.981
P+O_T	-0.83P+1.86O_T+4.38	0.994	0.988
P+U_T	7.72P+2.70U_T+83.25	0.957	0.916
	9.18P+2.68U_T	0.956	0.913
P+I_T	17.38P+3.47I_T-155.46	0.852	0.726

Table D.6 : Regression models for LOC based on P, O_T, I, I_T and U_T

Based on	LOC*hat*=	Coeff.Corr	Coeff.Det
P	3.27P-3.15	0.884	0.782
P+I	2.56P+0.66I-17.37	0.906	0.820
P+O_T+I_T+U_T	1.33P+0.18O_T+0.10I_T-0.12U_T-10.67	0.973	0.946
P+O_T+I_T	1.51P+0.09O_T+0.23I_T-12.50	0.971	0.943
	1.32P+0.10O_T+0.20I_T	0.969	0.940
P+O_T+U_T	1.20P+0.24O_T-0.20U_T-8.37	0.972	0.945
P+I_T+U_T	1.80P+0.37I_T+0.12U_T-15.20	0.967	0.934
	1.58P+0.34I_T+0.12U_T	0.964	0.929
P+O_T	1.45P+0.13O_T-2.49	0.958	0.918
	1.41P+0.13O_T	0.958	0.918
P+U_T	2.35P+0.14U_T+1.52	0.919	0.844
P+I_T	2.43P+0.407I_T-20.61	0.946	0.895
	2.18P+0.37I_T	0.941	0.885

Table D.7 : Values of P, I_T and U_T bands

Parameter	5-Band	Border	Value	7-Band	Border	Value
P	1(upto11)	22.0	15.36	1(upto16)	26.5	18.31
	2(upto23)	31.0	27.50	2(upto32)	37.5	32.38
	3(upto35)	39.5	35.67	3(upto48)	46.0	41.88
	4(upto47)	46.0	42.33	4(upto64)	69.0	59.38
	5(upto59)	64.0	55.08	5(upto80)	>69.0	89.44
	6(upto71)	85.0	72.17			
	7(upto80)	>85.0	100.00			
I_T	1(upto11)	52.0	37.00	1(upto16)	61.5	43.88
	2(upto23)	79.0	66.92	2(upto32)	92.5	80.50
	3(upto35)	98.0	88.50	3(upto48)	141.0	114.00
	4(upto47)	141.0	116.67	4(upto64)	205.0	169.25
	5(upto59)	178.0	155.67	5(upto80)	>205.0	288.44
	6(upto71)	258.5	213.67			
	7(upto80)	>258.5	337.00			
U_T	1(upto11)	78.5	39.36	1(upto16)	87.0	53.31
	2(upto23)	112.0	94.00	2(upto32)	155.0	120.19
	3(upto35)	167.5	141.33	3(upto48)	241.5	194.50
	4(upto47)	241.5	199.08	4(upto64)	401.0	294.25
	5(upto59)	314.0	267.17	5(upto80)	>401.0	652.63
	6(upto71)	587.0	424.67			
	7(upto80)	>587.0	787.78			

Table D.8 : Performance of the 5 and 7 banded models using P, I_T and U_T

Model	MMRE	Pred		
		0.20	0.25	0.30
$3.14P+2.59I_T+2.52U_T$	0.06	0.99	0.99	0.99
$3.14(P)_{5B}+2.59(I_T)_{5B}+2.52(U_T)_{5B}$	0.19	0.71	0.83	0.86
$3.14(P)_{5B}+2.59(I_T)_{7B}+2.52(U_T)_{5B}$	0.17	0.73	0.86	0.91
$3.14(P)_{5B}+2.59(I_T)_{5B}+2.52(U_T)_{7B}$	0.15	0.85	0.90	0.91
$3.14(P)_{5B}+2.59(I_T)_{7B}+2.52(U_T)_{7B}$	0.13	0.90	0.95	0.96
$3.14(P)_{7B}+2.59(I_T)_{5B}+2.52(U_T)_{5B}$	0.18	0.75	0.85	0.88
$3.14(P)_{7B}+2.59(I_T)_{7B}+2.52(U_T)_{5B}$	0.16	0.79	0.86	0.90
$3.14(P)_{7B}+2.59(I_T)_{5B}+2.52(U_T)_{7B}$	0.15	0.84	0.91	0.94
$3.14(P)_{7B}+2.59(I_T)_{7B}+2.52(U_T)_{7B}$	0.12	0.88	0.94	0.95