System architecture metrics: an evaluation

Thesis

System Architecture Metrics: An Evaluation

Martin John Shepperd BA, MSc

In part fufilment of the degree of Doctor of Philosophy

Computer Science

March 1991

To Linda

Abstract

The research described in this dissertation is a study of the application of measurement, or metrics for software engineering. This is not in itself a new idea; the concept of measuring software was first mooted close on twenty years ago. However, examination of what is a considerable body of metrics work, reveals that incorporating measurement into software engineering is rather less straightforward than one might pre-suppose and despite the advancing years, there is still a lack of maturity.

The thesis commences with a dissection of three of the most popular metrics, namely Halstead's software science, McCabe's cyclomatic complexity and Henry and Kafura's information flow - all of which might be regarded as having achieved classic status. Despite their popularity these metrics are all flawed in at least three respects. First and foremost, in each case it is unclear exactly what *is* being measured: instead there being a preponderance of such metaphysical terms as complexity and quality. Second, each metric is theoretically doubtful in that it exhibits anomalous behaviour. Third, much of the claimed empirical support for each metric is spurious arising from poor experimental design, and inappropriate statistical analysis. It is argued that these problems are not misfortune but the inevitable consequence of the *ad hoc* and unstructured approach of much metrics research: in particular the scant regard paid to the role of underlying models.

This research seeks to address these problems by proposing a systematic method for the development and evaluation of software metrics. The method is a goal directed, combination of formal modelling techniques, and empirical evaluation. The method is applied to the problem of developing metrics to evaluate software designs - from the perspective of a software engineer wishing to minimise implementation difficulties, faults and future maintenance problems. It highlights a number of weaknesses within the original model. These are tackled in a second, more sophisticated model which is multidimensional, that is it combines, in this case, two metrics. Both the theoretical and empirical analysis show this model to have utility in its ability to identify hard-to-implement and unreliable aspects of software designs. It is concluded that this method goes some way towards the problem of introducing a little more rigour into the development, evaluation and evolution of metrics for the software engineer.

Acknowledgments

First and foremost I would like to thank my supervisor, Professor Darrel Ince, for all his help and guidance. He has not only been my mentor but also a good friend. I would also like to thank Wolverhampton Polytechnic for a sabbatical year during which the bulk of research described by this thesis was conducted. My colleagues have also offered encouragement, useful criticisms and insightful comments. These have all contributed to the completion of what has often appeared - to me at least - a monumental task! Finally, I owe a deep debt of gratitude to my wife, Linda and children Polly and Adam, who have had to put up with my all too frequent absences. Thankyou.

Published Work

The following is a list of work contained within the thesis that has been published elsewhere.

[Ince88b] Ince, D.C. Shepperd, M.J . 'System design metrics: a review and perspective.' *Proc. IEE / BCS Conf. Software Engineering '88* July 12- 15, Liverpool University, pp23-27. 1988.

[Ince89a] Ince, D.C. Shepperd, M.J. 'An empirical and theoretical analysis of an information flow based design metric'. *Proc. European Software Eng. Conf.*, Warwick, England. Sept. 12-15, 1989.

[Ince89b] Ince, D.C. Shepperd, M.J. 'Quality control of software designs using cluster analysis'. *Proc. EOQC/SQA Conf. Management of quality: key to the nineties.* Vienna,

[Ince90a] Ince, D.C. Shepperd, M.J. 'The use of cluster techniques and system design metrics in software maintenance'. *Proc. IEE/DTI UK IT'90 Conf.*, Southampton, UK, March 1990.

[Shep88a] Shepperd, M.J. 'A critique of cyclomatic complexity as a software metric' *Softw. Eng. J.* 3(2) pp30-36. 1988.

[Shep88b] Shepperd, M.J. 'An evaluation of software product metrics.' *Information & Softw. Tech.* 30(3) pp177-188. 1988.

[Shep89a] Shepperd, M.J. Ince, D.C. 'Metrics, outlier analysis and the software design process'. *Information & Softw. Tech.* 31(2) pp91-98.1989.

[Shep89c] Shepperd, M.J. 'A metrics based tool for software design' *Proc. 2nd Int. Conf. on Softw. Eng. for Real Time Systems*, The Royal Agriculture College, Cirencester, UK. Sept. 18-20, 1989.

[Shep89d] Shepperd, M.J. 'Specification: a new perspective on design metrics'. *The Polytechnic: Wolverhampton, School of Computing and Information Technology, Technical Report 89/01.* (Also accepted for publication *The Computer J.*). .

[Shep90a] Shepperd, M.J. 'An empirical study of design measurement'. *The Softw. Eng. J.* Jan. 1990.

[Shep90b] Shepperd, M.J. Ince, D.C. 'The multi-dimensional modelling and measurement of software designs'. *Proc. Annu. ACM Comp. Sci. Conf.*, Washington DC, Feb.20-22, 1990.

[Shep90c] Shepperd, M.J. 'Early life cycle metrics and software quality models', *Information & Softw. Tech.* 32(4) pp311-316, 1989.

[Shep90e] Shepperd, M.J. Ince, D.C. 'Controlling software maintainability', *Proc. 2nd European Conf. on Softw. Quality Assurance*, Oslo, Norway, 1990.

[Shep90f] Shepperd, M.J. Ince, D.C. 'The use of metrics for the early detection of software design errors'. *Proc. BCS/IEE Software Engineering 1990*, Brighton, July 24-27, 1990.

Contents

Abstract	i
Acknowledgments	iii
Published work	iv
1.INTRODUCTION	1
1.1 Background to the problem area	1
1.2 Aims of the research	3
1.3 Some definitions	4
1.4 Organisation of research	5
2. SOFTWARE METRICS REVIEWED	7
2.1 Metrics: a brief history	7
2.2 Code metrics	9
2.3 Design metrics	24
2.4 Specification metrics	36
2.5 Summary	40
3. PROMISES AND PROBLEMS	42
3.1 Why Software Science, Cyclomatic Complexity and Information Flow?	42
3.2 Software science and the magical number eighteen	44
3.3 Decision count plus one - alias the cyclomatic number	49
3.4 Henry and Kafura's information flow measure	54
3.5 Unfulfilled promises	64
4. A THEORETICAL FRAMEWORK FOR DESIGN MEASUREMENT	67
4.1 Introduction	68
4.2 The Theory of Measurement	70
4.3 Modelling and Measurement	76
4.4 Model Evaluation	82
4.5 A Methodology for the Development of Software Metrics	93
4.6 Summary	103
5. UNI-DIMENSIONAL MODEL OF SOFTWARE DESIGN	105
5.1 Restatement of the Problem	105
5.2 A Formal Model	108
5.3 Theoretical Model Behaviour	124
5.4 An Empirical Analysis	133
5.5 Evaluation of the Uni-dimensional Model	138
6.A MULTIDIMENSIONAL MODEL OF SOFTW^ RE DESIGN	141
6.1 Why multidimensional models?	142
6.2 Measuring module size	142
6.3 Validation of the "work" metric	147

6.4 The Multidimensional Model 162
6.5 Summary 166

7. CONCLUSIONS 167
7.1 Research Coverage 167
7.2 Summary of Findings 168
7.3 Suggestions for Further Work 174
7.4 Postscript 175

References 176

"In truth, a good case could be made that if your knowledge is meagre and unsatisfactory, the last thing in the world you should do is make measurements. The chance is neglible that you will measure the right things accidentally."

George Miller

1. INTRODUCTION

Synopsis of chapter

This chapter defines the problem area for the research described within this doctoral thesis, namely the application of quantitative methods to software engineering and the design of system architectures or structures, in particular. It is argued that this is of considerable significance because of the potential impact of poor design decisions upon a wide range of quality factors of the system when it is finally implemented. The software engineering background to this research is then described. This is followed by a more detailed appraisal of the research objectives, and the chapter concludes with some discussion of the research strategy that will be adopted in attempting to carry out the research programme that has been outlined.

1.1 Background to the problem area

This thesis is concerned with quantitative approaches for software engineering, otherwise known as software metrics. Within this rather broad domain we have a particular concern with the design of software system architectures. To date the main thrust of research has been towards the measurement of code, yet it is our belief that by the time the code is available for measurement the majority of a software project's resources have been committed, so that any strategic changes in direction become prohibitively expensive. Therefore the emphasis of this research will be upon metrics that be extracted, and acted upon, at earlier stages of a software project.

Since the software engineering was first proposed as a discipline in its own right in 1968 [Rand68] the primary focus of attention has moved progressively earlier and earlier in the so-called project life cycle. For much of the 1960's and 1970's the major debates concerned code and how to structure the control flow, for example the work of Dijkstra [Dijk68], Dahl *et al* [Dahl72] and Wirth [Wirt76]. More recently, interest began to focus upon the design aspects of software, such as the work of Parnas [Parn72, 79] upon criteria for the modularisation of software and its impact upon the maintenance characteristics of the software. Likewise, Jackson's well known boating lake example [Jack75] demonstrated the impact of design decisions upon the maintainability of the resultant code. The conclusion was that although structured programming techniques were important, they could not compensate for deficiencies in the design. More recently still, attention has turned to those stages of a project before design. These are often termed requirements ıgineering: the aim of which is to transform the informal system requirements of the user or customer into a more structured specification document to be used by the project developers. Clearly, if this

document does not accurately and unambiguously describe the required system then no design methodology or amount of structured programming will prevent the system being developed from failing to satisfy the user and consequently entailing costly rewrites. These rewrites are especially costly since they imply the maximum amount of backtracking because specification, design and coding must re-done. Acknowledgment of the significance of these potential problems has led to the proliferation of structured systems analysis methods [deMa79, NCC86, Ashw90] and also an upsurge of interest in the application of mathematics to the creation and validation of specifications [Hoar78, Jone86, Miln89].

It is evident, then, that the early stages of the software development process, such as requirements analysis and design, are critical to the successful implementation of software systems. It is now widely recognised that errors committed and poor decisions made, during these early stages result in the most costly and intractable problems [Boeh81]. Unfortunately feedback concerning such problems is usually not available until late on in the development process, often during integration testing or later.

As has already been stated the major concern of this research is with design. Requirements engineering is not normally regarded as a design activity because its purpose is to ascertain and describe the user's requirements, because it is aimed at *what* the system should do rather than *how* it should be accomplished. It is the choices concerning the *how* that have the most impact upon such system quality factors as maintainability and implementation effort. In other words for a given specification - namely the one that the user or customer wants - there are a variety of designs or outline solutions. These various solutions cannot be considered to be equivalent in terms of their impact upon system quality factors, despite the fact that they all implement the same set of requirements.

Decisions during design concerning system architecture, have a special significance upon the maintainability, reliability and ease of implementation of the resultant software. Given the absence of immediate feedback, the designer is forced to make decisions concerning alternative architectures with little understanding of their consequences in terms of impact upon the characteristics of the system being developed.

Design methodologies such as Structured Design [Stev74], Jackson Structured Programming [Jack75] and Object Oriented Design [Booc86] attempt to address the problem of very late feedback, by the provision of guide-lines for the generation of system structures and design evaluation criteria. The benefits of such methodologies are considerable; however, they are not mechanistic and the evaluation criteria tend to be qualitative and subjective, for instance module coupling and cohesion [Myer75].

Although the bulk of metrics research has been in the code arena, various design based metrics have been proposed to address the shortcomings of current approaches to

software design. In particular there has been a search for measures that can predict quality factors such as maintainability, whilst the software development is still at the design stage. Another objective has been to provide objective, quantitative evaluation criteria with which to compare competing system architectures. Though, design metrics are potentially very important for the software engineer, there has been limited work on the validation of design metrics and many of the findings have been contradictory[1]. Consequently, there has been almost no take up by the software industry.

The dichotomy between the importance of applying quantitative methods to the design of system architecture and the absence of consistent results or widely accepted metrics typifies the area of software measurement at present. One feels that if software engineering is to truly become an engineering discipline, then quantitative models and the use of measurement are imperatives. Yet at present there appears to be little progress in this direction. This is a theme that the thesis will return to on more than one occasion.

1.2 Aims of the research

And so to the aims of this research. As has already been stated, we wish to develop metrics that provide *useful* quantitative feedback for the designer of software system architectures. This feedback will enable the designer to better understand the consequences of a particular design decision, to compare candidate structures, to be able to identify critical parts of an architecture, predict potential maintenance "hot spots" and guide design inspections to focus on potentially most troublesome areas[2]. Furthermore, we recognise the importance of these design decisions upon a whole range of quality factors, most notably ease of implementation, reliability and maintainability.

However, this first aim leads to a number of further aims. The adjective useful has been emphasised in foregoing discussion. This introduces the need for validation. There is no shortage of metrics [Curt79b]; what are less in evidence though, are metrics that have been satisfactorily evaluated and are consequently widely accepted. The problem of conflicting empirical evidence has already been highlighted. This suggests that although the first concern of this research is with system architecture metrics, it will be necessary to evolve a general framework for the development of

[1]An example of contradictory findings concerns the design metric of Henry and Kafura [Henr81a] known as information flow. Although Henry and Kafura report strong empirical support from their study of the UNIX operating system, other studies have been far more equivocal, for instance the study of the VME operating system by Kitchenham [Kitc88]. Unfortunately, present there does not exist an adequate framework for comparing these studies or reconciling differing claims.

[2]These ideas are further developed in [Shep89a]. In particular it is shown how design metrics may be employed to successively refine a system architecture.

software metrics, evaluation and application. In a sense, it is must be a research objective to bridge the gap that Belady identifies between the "speculators" and the "doers" [Bela79].

It is also important to identify from the outset areas that will not be addressed by this research. Frequently metrics are characterised as either product or process metrics [Shep88b]. As their name implies, the former are measurements derived from software engineering products, for example software designs, test suites and code. These will be the primary concern of this research. However, there is another class of measurement that are extracted from software engineering processes, such as the number of faults uncovered in a design inspection. Although these two classes of metrics are related, this research will not explicitly deal with the process measurements. This is for three reasons. First, the focus of the research is upon a product *viz.* a system architecture and not upon the methods or processes that have been employed to generate that system architecture. Second, to begin to adequately model and measure processes requires a considerable background in cognitive psychology, and sociology. Last, for entirely pragmatic reasons it is necessary to somewhat limit the scope of this research.

1.3 Some definitions

Before we proceed further it is probably appropriate to consider some definitions. First, metrics. Unfortunately there is no generally agreed definition. Some workers use the term interchangeably with measurement. Others distinguish between measurement and metric, where metric indicates a measurement and an underlying model or theory. The latter is intended to emphasise the need for measurements to be placed into a context of a theory. Although laudable in its aim there is such a divergence between it and reality - as will be shown in subsequent chapters - that we do not believe such a definition to be very helpful. Accordingly, within this thesis a metric merely conveys a measurement of a software engineering product or process: no more and no less.

It might also be useful to consider what is meant by design. In a very abstract sense design encompasses any activity where there is an element of choice or decision making. Without choice the activity is entirely deterministic and therefore, at least potentially, may be fully automated. Evidently, this encompasses transforming a requirements specification into an outline solution. It is customary, within software engineering, to distinguish between high level and detailed design. The former is concerned with architecture - for example the choice of modules, their interfaces and data structures - whilst the latter is concerned with algorithms. This research is concerned with the former because it is availat earlier in the software life cycle, and because it is believed to have a greater impact upon implementation effort, reliability and maintainability [Your79, Stev80]. One major restriction is imposed.

This research will focus upon the design of "function strong" systems (for instance a reactor control system), as opposed to "data strong" systems (for instance a library catalogue system) [deMa82]. Again the reason for this is primarily a pragmatic one so as to restrict the area of research[3].

1.4 Organisation of research

The research falls into three stages. First, existing software metrics and metrics research will be reviewed. Although, we have a particular interest in design metrics all metrics derived from any software engineering product will be scrutinised in a search for any underlying trends or general patterns. Second, the research will need to devise a general framework for the development or selection of software metrics, followed by evaluation and application. In the devising of such a framework it may be necessary to borrow from other disciplines such as classical measurement theory, experimental design and statistics. Third and last, will be the development and evaluation of a measure, or set of measures, using the framework described above in order to meet the original goal of aiding the software designer choose between alternative system architectures in order to improve the ease of implementation; the reliability of implementation, and the ease of maintenance. It is expected that applying the method to an actual measurement problem - that of developing design metrics - will yield information concerning the utility of the method itself. It is also anticpated that the method will facilitate the process of developing design metrics, especially the problem of metric evaluation.

This research will be of little value if all that emerges is yet more untested and unvalidated metrics. There is evidence that there is no shortage, at present, of such metrics [Curt79a, Shep88b]. Instead the themes of method and evaluation will recur throughout this dissertation.

The next chapter, chapter two, reviews the current state of metrics research with respect to specification, design and code metrics. Particular attention will be given to the evaluation of the various metrics described. Chapter three offers a detailed dissection of three of the most popular and influential metrics, where it is argued that all suffer from a similar malaise, of weak underlying models and unconvincing empirical validations. From this are drawn lessons that then form the basis of a method for the development and evaluation of software metrics, which will be outlined in chapter four. Chapter five will illustrate the method by the development and evaluation of a design metric. However, the evaluation highlights various

[3]Subsequent to the research described within this d ıoral dissertation, we have commenced an investigation into the possibility of providing data designers with quantitative feedback. Early results, described in [Ince90b, c] suggest that at least some of principles for functional design are appropriate for data modelling.

deficiencies which will be remedied in a more sophisticated model given in chapter six. The research findings are summarised in chapter seven.

2. SOFTWARE METRICS REVIEWED

Synopsis of chapter

This chapter reviews developments in the field of software metrics from its inception in the early seventies to the present time. Metrics can be categorised according to the stage of software development they associated with. Thus we have code metrics, design metrics and specification metrics. Although our primary interest is with software designs, code measures will also be examined and in particular the question will be asked - can any techniques be extended back through the software development process? Similarly with specification metrics, the question will be asked - can measures extracted from specifications provide insights into the software design process?

It is shown that the most substantial work is still being performed in the code metric arena, despite increasing interest in earlier life cycle measures. The latter probably reflects a more general trend within software engineering, namely the relatively recent discovery of, and emphasis upon, non-coding aspects of software systems. A number of design metrics have been proposed but few validated. Specification metrics are at an even more speculative stage, particularly given the absence of any standard approach by the software industry to the issue of requirements specification[1], particularly in terms of notation. Moreover, they would seem to be restricted to addressing size related issues since specifications deal only with, (or ought to), the nature of the problem, not its solution. Thus this chapter summarises the current state of the art for software metrics. This then provides the necessary groundwork for an in depth analysis of three of the most prominent metrics in the following chapter.

2.1 Metrics: a brief history

In the early days of fifties and sixties problems of computer hardware resources tended to be the most pressing. As a result measurement was almost universally targeted at the issues of computational time and the memory requirements of algorithms. However, as has often been remarked, this is no longer the case. From the seventies onwards, developments in computer hardware have outstripped progress in software technology. Most costs and bottle-necks are now associated with the software parts of a computer

[1]For example, even such well publicised techniques as data flow diagraming and entity-relationship analysis remain minority practises.

system [Boeh81]. There has been a parallel movement in software metrics so that most effort is now directed at measuring properties of the software that lead to the consumption of human resources (e.g. programming effort, reliability etc.) rather than the hardware resources, since these tend to be less costly.

This interest in human resources expended on the development and operation of software systems has manifested itself as an attempt to quantify software complexity. Complexity is perceived as the "root of all evil"[2] and if only it could be reduced this would bring about attendant reductions in all manner of software evils such as excessive development and testing effort, unreliability, and unmaintainability. Despite the obvious appeal of such a proposition, software complexity has proved to be a rather intractable proposition notwithstanding the large amount of attention focused in its direction.

From the early seventies onwards software engineers have attempted to measure software complexity, chiefly by concentrating upon a few syntactical properties of program code; for instance the number of tokens or the number of program decisions. Towards the end of the seventies a growing realisation began to emerge that many of the most costly and least desirable software attributes were the consequence of problems during the design stage [Parn72, Jack75, deMa78, Stev80]. It was thus a natural progression to consider measuring properties of designs in order to identify complexity earlier in the software life cycle. This has proved less straightforward than one might imagine due to the lack of standardisation in either the practice of design or even use of notation. Nevertheless a number of design metrics have been proposed.

Given the trend of increasing concern for the early stages of the software life cycle, it is hardly surprising that in the eighties there have been some attempts to measure specifications, although these have been targeted at project size and effort estimation rather than complexity as such. Specifications ought only describe the problem and therefore it is not possible to obtain insights into its solution (i.e. design and implementation) or solution complexity. Specifications tend also to be less standardised or structured than designs, leading to problems of comparability of measurements. Even so, the management benefits of obtaining early predictions of system development are such that there have been a number of metrics developed in this area.

The remainder of this chapter, surveys what may then loosely be called software complexity metrics[3], commencing with code metrics and then progressing back through

---

[2]To paraphrase St. Paul more accurately, "love of con plexity is the root of all evil" (1 Timothy ch. 6 v 10)!

[3]We will return to the matter of software complexity at a later stage, and argue that it is nebulous to the extent that it defies measurement, and is the source of the fruitlessness of much current work and should

the software life cycle. It concludes with a summary of the current state of affairs and future prospects for software metrics.

2.2 Code metrics

2.2.1 Lines of code

The simplest software complexity metric is lines of code (LOC). The basis for LOC is that program length can be used as a predictor of program characteristics such as reliability and ease of maintenance. Despite, or possibly even because of, the simplicity of this metric, it has been almost universally reviled [McCa76, deMa82, Ejio85]. Certainly there are serious difficulties with defining what actually constitutes a line of code, consequently modifications such as the number of source statements or machine code instructions generated, have been advanced. None of these modifications could exactly be described as being in vogue.

The suggestion of Basili and Hutchens [BasH83] that the LOC metric be regarded as a baseline metric to which all other metrics be compared is appropriate. It would be reasonable to expect an effective code metric to perform better than LOC, and so as a minimum, LOC offers a "null hypothesis" for empirical evaluations of software metrics[4].

2.2.2 Software science metrics

One of the earliest attempts to provide a code metric based on a coherent model of software complexity was provided by the late Maurice Halstead. His influences appear to have been extraordinarily diverse ranging from thermodynamics, Shannon's Information Theory, cognitive psychology to reverse compilation [Hals72]. This led to the postulating of a set of general laws that Halstead saw as being analogous to natural laws [Hals72,Hals77]. Initially software algorithms were the object of interest but this rapidly burgeoned to "linguistics, psychology, or any field dealing with 'Man the Symbol Manipulator'" [Hals72]. Indeed, it was suggested that software science might usefully be employed for such diverse fields as semantic partitioning, child development psychology and Shakespearean analysis [Hals79b]! Not to be outdone, a

---

be dropped as a measurement objective forthwith. Nevertheless we will stick with conventional nomenclature for the meantime so as to minimise confusion.

[4] In passing it is also worth noting that much empirical work [Kitc81, Henr81a, BasP84] has shown the metric to correlate strongly with other metrics, most notably McCabe's cyclomatic complexity [McCa76] as demonstrated by Shepperd [Shep88a].

recent study has even attempted to apply software science to the problem of modelling compiler performance [Shaw89].

Halstead originally described the work as software physics [Hals72, Funa76, Love76, Knif78] but this was subsequently discarded in favour of the soubriquet, software science [Hals77, Fitz78b, Hals79a]. The underlying concept was that software comprehension is a process of mental manipulation of program tokens. These tokens can be characterised as either operators (executable program verbs such as IF, DIV and READ) or operands (variables and constants). Thus a program can be thought of as a continuous sequence of operators and their associated operands.

To derive the various software science metrics the following counts are required.

n_1 = count of unique operators

n_2 = count of unique operands

N_1 = total no. of operators

N_2 = total no. of operands

The program vocabulary, n is given by:

$$n = n_1 + n_2 \qquad (1)$$

and the program length, N in tokens, by:

$$N = N_1 + N_2 \qquad (2)$$

Halstead suggested that manipulation of each token requires its retrieval from a sort of mental dictionary comprising the entire program vocabulary n, and that this was by means of a binary search mechanism. Therefore, the number of mental comparisons, or dictionary accesses, required to understand the piece of software can easily be calculated from the size of the vocabulary and the total number tokens that are used. Halstead referred to this as the program volume, V which is:

$$V = N * \log_2.n \qquad (3)$$

Since the same program may be implemented in a number of different ways, it is useful to have a measure of a particular implementation's volume relative to some theoretical optimum solution with the minimum possible volume, V^*. Halstead termed this the program level L:

$$L = V^* / V \qquad (4)$$

Since an increasing difficulty metric, D seems to be intuitively more satisfying than a diminishing level metric, many investigators have added a difficulty metric D which is the inverse of program level:

$$D = 1 / L \qquad (5)$$

In practice it is virtually impossible to derive the potential volume V*, so an estimate of the program level, ^L is used:

$$\hat{L} = (2 / n_1) * (n_2 / N_2) \qquad (6)$$

Similarly, the estimated difficulty metric, ^D is:

$$\hat{D} = (n_1 / 2) * (N_2 / n_2) \qquad (7)$$

The rationale for equation 7 is as follows. The term $(n_1/2)$ will increase with the use of additional operators thereby adding to the complexity of the code. The divisor is 2 on the basis that this is the minimum possible number of operators to implement a particular algorithm (i.e. a function call and a function argument grouping operator)[5].

[5]This justification for estimated difficulty, ^D is restricted to those programming languages which support procedure invocation and parameter passing. As consequence software science has a more limited domain than some of its proponents claim (e.g. [Knij78]).

The other term (N_2/n_2) represents the average use of an operand. The more frequently an operand is referenced the greater the complexity.

Since the program difficulty gives the number of elementary mental discriminations (EMD) per comparison, and the volume gives the total number of comparisons, it is possible to derive the effort, E which is required to manipulate the program in terms of EMD's.

$$E = D * V \qquad (8)$$

Halstead's model assumes that programmers make a constant number of these EMD's per second. By adapting work by the psychologist Stroud [Stro66] he suggested that the time, T required to generate the program could be calculated by using the Stroud number, S which is the number of EMD's the brain is able to make per second[6]. Stroud estimated S to lie within the range 5 to 20. Halstead by using a value of S=18 was able to predict T in seconds as:

$$T = E / 18 \qquad (9)$$

The model also provides an equation for estimating program length, N^ using only the counts n_1 and n_2 which might be available prior to completion of coding. Thus we have:

$$N^{\wedge} = n_1.\log_2.n_1 + n_2.\log_2.n_2 \qquad (10)$$

Halstead further hypothesised that for a given programming language, that as the potential volume V* increases, the program level L will decrease in such a way that V*.L remains invariant. Using this invariant, which he termed λ or the language level, values were obtained of 1.53 for PL/1, 1.21 for ALGOL, 1.14 for Fortran and 0.88 for CDC assembly language [Hals77]. Subsequent attempts to establish language levels for other programming languages have been wide ranging: for instance ESS [Bail81], Cobol [Shen81, Zweb79], PL/S and BAL [Smit80], RPG [Hart80] and APL [Zweb79, Laur82]. A

[6]Unfortunately, Halstead took Stroud's work rather out of context and in addition adopted a value that lies very much at the top of the range of values suggested by Stroud. This theme will be further developed in the critique of Halstead's work in the next chapter.

feature apparent in all the above investigations is that, with the exception of APL which produced a surprisingly low lambda value, the results are approximately in accord with one's intuitive expectations. However, more careful analysis reveals large variances and a strong inverse dependence on length [Hame82, Shen83]. Christensen *et al* [Chris81] suggest that what is being captured is not the language level so much as the use of the language by the program. These doubts have not discouraged continuing efforts as exemplified by Kokol on spreadsheet software [Koko89], Konstam and Wood for APL [Kons85] and Wang [Wang84b] for Pascal and Ada.

Early empirical validations of software science produced seemingly[7] very high correlations between predicted and actual results. Studies related software science measures to development time [Gord76, Hals77], incidence of software bugs [Funa76], program recall [Love77] and program quality [Love76, Elsh76, Fitz78a]. Ottenstein [Otte76] even reported the successful application of software science counts to the problem of detecting student plagiarism. Extending the educational application possibilities still further Shen investigated the relationship between software science and student grades [Shen79]. Gordon [Gord79] suggested that the effort metric, E could be used as an indicator of good programming practice, on the basis that E was reduced 40 times out of 46 examples of 'improved' programs culled from Kernighan and Plauger's classic work on programming style [Kern78]. Elshoff [Elsh76] analysed some commercial PL/1 programs, and although encountered some difficulties with the counting rules, particularly with regard to constants, again reported positive results.

Unfortunately, subsequent work has been rather more equivocal. The first intimation of problems with the software science came in the late 1970's when researchers [Fitz78b] attempted to apply the metrics to two sets of independently published experimental data due to Gould of program debugging [Goul75] and Weissman of program comprehension [Weis74]. In both cases correlations were weaker than previously reported or not statistically significant. Another study by Bowen [Bowe78] found only a modest correlation between the level metric (the inverse of difficulty) and number of errors detected in 75 modules. Particularly disconcerting was the fact that it was out performed by the much scorned LOC metric.

A pattern of confusion began to emerge as it became apparent that some researchers had been correlating errors with effort [Funa76, Fitz78b] and others with difficulty or level [Bowe78, Feue79, Smit80]. It was also suggested that the cause of the less than stratospheric correlations between E and bugs reported for three large software systems

[7]Careful review of the methods and analytical techniques employed, for example by [Hame82], suggest that these correlations are less significant than was first believed - a point that will be returned to in the next chapter.

[Fitz78b] were the consequence of poor data collection[8], in that the researchers did not have access to data on faults detected during development. It is not clear why this should have such a large bearing upon the application of the software science model.

Three large controlled experiments carried out by Curtis *et al* [Curt79a, Curt79c] and Sylvia Shepherd *et al* [Shep79] attempted to relate the software science E measure to the time taken to carry out a simple maintenance task. Initial results were extremely discouraging in two ways. First correlations were low or not statistically significant. Second LOC out performed the E measure in almost all cases. Variations in subject ability was held to be a factor for the poor performance of the software science model coupled with an insufficient range of metric values[9]. These were rectified in the third experiment and more significant relationships observed between E and maintenance effort. However, it was also clear that E was more strongly correlated with LOC and than with maintenance time. Again LOC out performed E when the analysis was carried out at the module rather than the program level. Thus Curtis et al concluded that software science performs better as program size increases. They also suggested that the relationship was probably curvilinear. This was in marked contrast to both earlier work which ignored modularisation and Halstead's model which is based upon linear relationships.

Much of the experimental difficulty that Curtis *et al* [Curt79a,c] experienced was due to their awareness that many other factors significantly affected the maintenance process. In their discussion they mention programmer ability, type of maintenance change and explicitly adopt Thayer's classification of software bugs [Thay78], the context of the bug and the control structure of the program. It would not be difficult to considerably extend this list. Unfortunately none of these factors are incorporated into the software science model. At the very least, one is drawn to the conclusion that the analogy that Halstead draws between his work and the laws of thermodynamics [Hals72] is an inexact one.

In contradiction to the findings of Curtis *et al*, Woodfield [Wood80] found that the E metric had a marked tendency to over-predict for large software systems. These findings were corroborated by a study of 197 PL/1 programs where in 80% of the cases the length equation over-estimated size [Feue80]. However, Woodfield found that decomposing larger programs into small "logical chunks" improved the performance of E, although still only marginally better than LOC [Wood81a]. Another large study of over 400 modules by Basili and Phillips [BasP81] found that E did not predict development time significantly better than LOC. Yet another study of program

[8]Needless to add, data collection problems do not significantly effect high correlations!

[9]One might argue that a metric that fails to adequately discriminate between programs with clearly distinct characteristics is of reduced utility.

comprehension [Wood81b] found that for small scale software, there was a correlation between E and program recall. A more recent study still [BasS83], reports that LOC *out* performs E for effort and error prediction.

However, to add to this already confused situation many of the early studies have been questioned, both on statistical and experimental grounds [List82, Hame82]. Card and Agresti [Card87] also indicate that correlations based on the length equation (10) cannot be accepted because this equation is mathematically dependent upon actual length[10].

What may be concluded from the above empirical work? Disarray seems to be the most predominant feature. Nevertheless, certain patterns emerge: researchers have attempted to apply the software science model to a very large range of software products and processes; there seems to be little standardisation even in such fundamental areas as counting n_1 and n_2 and this alone must account for some of the variations in results [Elsh78, Fits80, Chri81, Salt82, Cont82]. The measure that has attracted most attention *is the effort metric, E (equation 8). This is often used as a* predictor of such product characteristics as maintainability and reliability. The curious feature concerning E is its consistently high correlation with the more traditional metrics such as ELOC and LOC.

Software science has attracted much interest because it was the first attempt to provide a coherent framework within which software could be measured. It has the advantage that since it deals with tokens it is fairly language independent. Moreover, the basic inputs n_1, n_2, N_1 and N_2 may *all be easily extracted automatically*[11].

Serious criticisms that have been levelled at software science, which will be examined in more detail in the next chapter, however, its role would at present appear to be very limited, especially as a universal model of program complexity. Despite these difficulties there remains widespread and uncritical reference to it, even in the recent literature and text books (e.g. [Arth85 ,Prat88, Schn88]. Nevertheless, given the absence of a clear pattern in the empirical analyses described above, one must have misgivings concerning the model and or its application. Possibly the most important legacy of software science is in the way that it attempts to provide a coherent and explicit model of program complexity, as a framework within which to measure and make interpretations. Conte *et al* [Cont86] provide a fitting epitaph for Halstead's work:

---

[10]The way that Halstead defines estimated length N' ensures that there must exist a positive correlation between size and N^.

[11]*Vide* the 200 line Pascal program given by deMarco [deMa82] to obtain the basic software science counts. The cynic might suggest that this might be a major contributor to the extraordinary level of interest in empirical investigations of software science.

"Halstead's work was instrumental in making metric studies an issue with computer scientists. Although the model of the programming process he proposed has limited empirical support, his work suggested that it was possible to apply a rigourous scientific approach to the programming process which for years had been considered an art."

2.2.3 Graph theoretic measures

An alternative but equally influential code metric is McCabe's cyclomatic complexity [McCa76]. His objectives were twofold: first to predict the effort of testing the software and thereby identify appropriate decompositions of the software into modules; second to predict complexity related characteristics of the resultant software.

Given the increasing costs of software development, McCabe stated that was required was a:

"mathematical technique that will provide a quantitative basis for modularisation and allow us to identify software modules that will be difficult to test or maintain". [McCa76]

Use of a lines of code (LOC) metric was deemed to be an inadequate approach, since McCabe could see no obvious relationship between length and module complexity. Instead, he suggested that the number of control paths, through a module, would be a better indicator, particularly as he perceived a strong relationship with testing effort. Furthermore, much of the work on "structured programming" in early 1970's emphasised program control flow structures [Dijk72, Dahl72].

The model that McCabe adopted was to view software as a directed graph with edges representing the flow of control and nodes as statements. In effect this is a simplified flow chart. Complexity was hypothesised as being related to control flow complexity, that is the more loops, jumps and selections the program contains the more complex it is to understand.

The control flow of any procedural piece of software can be depicted as a directed graph, by treating each node as an executable statement (or group of statements where the flow of control is sequential) and the edges as flow of control between them. The cyclomatic complexity of a graph indicates the number of basic paths within a graph,

which when taken in combination can be used to generate all possible paths[12] through the graph or module.

```
BEGIN
  REPEAT
     Writeln ('Enter a number or zero to stop');
     readln (num);
  IF num  0 > THEN
     writeln (num,'is positive')
  ELSE
     If num   0 < THEN
     writeln (num,'is negative');
  UNTIL num = 0 END
END
```

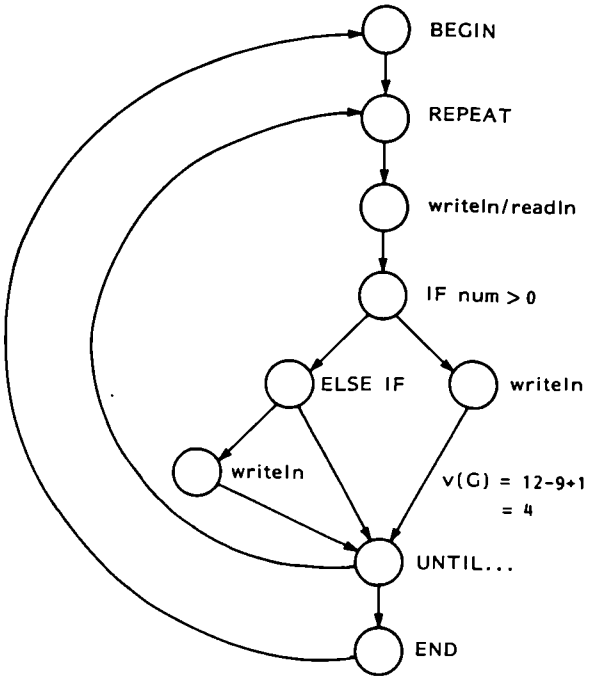


Figure 2.1: Derivation of v(G) from a program flow graph

The cyclomatic complexity of a strongly connected graph is equivalent to the number of its basic paths (i.e. linearly independent circuits). These in combination can be used to generate all possible paths through the graph or module. Thus McCabe decided to use cyclomatic complexity as a complexity metric.

The cyclomatic complexity, v of a graph G is:

$$v(G) = e - n + 1 \qquad (11)$$

where e is the number of edges and n is the number of nodes.

The graph is strongly connected if any node can be reached from any other node. Figure 2.1 shows an example derivation of cyclomatic complexity from a simple program and its related control graph. In this example, an additional edge (linking the END to the BEGIN node) has been added in order to make each node reachable. Where a program is made up of a number of modules this is modelled by a graph comprising a set of components, one for each module. For such a graph, S the cyclomatic complexity is:

[12]The cyclomatic number of a program flow graph is utilised instead of a simple count of unique paths because the latter is non-denumerable whenever the flowgraph contains a backwards branch.

$$v(S) = e - n + 2p \qquad (12)$$

where: p = number of connected components.

As McCabe observed the calculation reduces to a simple count of Boolean conditions plus one. He argued that since a compound condition, for example:

```
IF X < 1 AND Y < 2 THEN
```

was a thinly disguised nested IF, each condition should contribute to module complexity, rather than merely counting predicates. Likewise, a case statement is viewed as a multiple if statement (i.e. it contributes n-1 to v(G) where n is the number of cases).

McCabe saw a practical application of the metric in using it to provide an upper limit to module complexity, beyond which a module should be sub-divided into simpler components. A value of v(G) = 10 was suggested although it was accepted that in certain situations, notably large case structures, the limit might be relaxed.

As *per* the software science metrics, McCabe's ideas attracted a great deal of interest, and again due to the comparative simplicity of calculating the metric many empirical validations have been performed. His original validation [McCa76] was based upon conformance of the metric to intuitive judgments of program complexity, and in fact this approach was continued by many of the other early validations and extensions (e.g. [Myer77, Hans78]). Others adopted a more objective approach to empirical validation.

Results ranged widely[13]. Henry and Kafura, who whilst studying the UNIX[14] operating system, found a correlation in excess of 0.95 between v(G) and reported changes[15]. This is in complete contrast to other studies that report correlations as low as -0.09 between the metric and bug location [Bowe78]. Studies adopted varying interpretations as to exactly what v(G) was measuring or predicting. Basili and Perricone [BasP83] dealt with error density whereas Kitchenham [Kitc81] and Shen *et al* [Shen85] examined absolute error counts. Basili [BasS83], in yet another study, attempted to relate v(G) to programming effort, as did Gaffney [Gaff79]. Sylvia Sheppard [Shep79] correlated v(G)

[13]Table 3.2 summarises these empirical results in the next chapter.

[14]UNIX is a trademark of somebody somewhere or other!

[15]This result must be treated with some caution as Henry and Kafura appear to have ignored all error free, and the four largest, modules.

to the ability to recall a program and Sunohara *et al* [Suno81] observed the relationship with design effort. Curiously, there do not seem to have been any attempts to investigate the links with testing effort or number of test cases used.

The counting rules for different control statements have been the subject of some controversy. Myers [Myer77], by using certain examples, argued that complexity ought to be viewed as an interval using the number of predicates as a lower bound and the number of conditions as an upper bound. The basis of his argument seems to be that compound decision statements potentially allow fewer ELSE branches to be introduced than the equivalent predicate implemented as nested individual statements. For example:

```
IF X=0 AND Y>1 THEN ...                    v(G) = 3
               ELSE ...;                  Myers = (2:3)

IF X=0 THEN
         IF Y>1 THEN ...                   v(G) = 3
                ELSE ...                  Myers = (3:3)
ELSE ...;
```

Myers suggests that it is "intuitively obvious" that the second example is more complex than the first, a distinction not made by cyclomatic complexity. However, one could present a counter example:

```
IF X=0 AND Y>1 THEN ...                    v(G) = 3
               ELSE ...;                  Myers = (2:3)

IF X=0 THEN
         IF Y>1 THEN ...                   v(G) = 3
                ELSE ...;                 Myers = (3:3)
```

Here it is not all clear that the second example is more complex than the first despite the higher lower bound of the Myers interval. It would seem that counting ELSE branches would be a more effective alternative. More serious, and indicative of much of the confusion surrounding cyclomatic complexity, is the dilemma concerning what we actually wish to measure. From the point of view of number of branches and testing difficulty we may not wish to disagree with McCabe. Psychological complexity might be another matter.

The treatment of case statements has also been subject to disagreement. Hansen [Hans78] suggested that since they were easier to understand than the equivalent nested

ifs they should only contribute one to the module complexity. Other researchers [BasR79] have suggested a $\log_2(n)$ relationship where n is the number of cases.

Assuming that psychological or cognitive complexity is the target of our measurement process, several developments have been proposed that take into account the structuredness of the software. It is argued that to consider each decision in isolation is an inadequate model of program complexity and that the "cognitive load" is a product of the way in which decisions are combined. In other words a contextual view is required. Thus one development of McCabe's metric has been to incorporate a notion of nesting depth (e.g. [Duns80, Harr81, Mage81, Piow82, Prat84]).

Other researchers have modified the cyclomatic complexity metric in order to capture the degree to which the control flow of a piece of software is structured[16]. Woodward *et al* [Wood79] present their knot metric which uses the number of arc intersections of a program flow graph as an index of structuredness. A fully structured program will have zero knots. The authors report that in a library of Fortran subroutines, approximately one third had zero knots, one third less than 10 knots and less than one sixth over 20 knots. Brown and Fisher [Brow78] describe a software tool to automate the analysis of program flow graphs in order to identify non-structured control flow constructs. These ideas tend, however, to make the implicit assumption that the target program is at least Fortran-esque and so they have limited applicability.

Control flow complexity has continued to exercise the minds and imaginations of many metrics researchers. Zolnowski [Zoln81] reports that in his study of software complexity factors, software developers indicated that 8 out of 9 control flow measures were rated as very important. It is therefore hardly surprising that suggestions for control flow measures, most derived from McCabe's original work, continued apace. Schniedewind and Hoffman [Schn79] have defined control flow metrics based on the minimum number of paths and node reachability[17]. They reported significant correlations between these measures and number of errors and time to find errors. They also noted that these metrics out-performed both McCabe's metric and LOC. Subsequent variations on a theme include [Iyen82, Negr83, Stet84, Sinh86]. No empirical validations are offered. More novel is the attempt by Hall and Preiser [Hall84] to apply cyclomatic complexity to software designs. Again no empirical support is provided. Possibly the most imaginative of all, is the suggestion of Samson *et al* [Sams87] that the number of axioms in an algebraic specification will be equivalent to the v(G) of the resulting

---

[16]Structured is intended in the sense described by Dijkstra [Dijk68] and Dahl *et al* [Dahl72] amongst others.

[17]These metrics only feasible with the restriction that paths traversing a backward branch more than once are excluded.

imnplementation. We will return to this last suggestion when reviewing specification metrics.

A contrasting approach to program control complexity is due to Chen [Chen78]. An information theoretic viewpoint[18] is adopted whereby the source comprises only two distinct characters, a sequence and a selection, but may be infinitely long, thus most[19] program structures may be described. The entropy of a source is used to capture the relationship between control flow complexity and programmer productivity. Chen presents results that are suggestive of some empirical relationship, although he hypothesizes that it may be quantised rather than continuous. Davis and LeBlanc describe similar work [Davi88].

To summarise, there has been - and still is - much interest in capturing software complexity in terms of the complexity of control flow. There have been various approaches to measuring control flow complexity. These range from McCabe's simple model [McCa76] which can be characterised by the number of simple decisions plus one, to more sophisticated models that account for nesting depth (e.g. [Piow82]) or are based upon considerations of entropy [Chen78]. Of these metrics cyclomatic complexity is the most thoroughly validated with studies to correlate the metric with error-proneness, maintainability, understandability and development effort have produced erratic results. The most startling observations are the consistently high correlations with LOC and the out-performing of v(G) by LOC in a significant number of cases [BasH83, Curt79, Kitc82, Paig80, Wang84a]. *Few of the other metrics described have been* subjected to anything other than the most cursory empirical scrutiny, a point emphasised by the review of graph theoretic metrics in [Shep88a].

More serious still is the confusion concerning what is being measured. McCabe in his original paper [McCa76] seemed to commute between the idea of measuring testing difficulty, providing guide-lines for the modularisation of software and capturing cognitive or psychological complexity of the software. A subsequent report [McCa82] concentrated developing "programs that are not inherently untestable". However, by this time the seeds of confusion had already been sown. McCabe's underlying model is examined in more detail in the following chapter.

---

[18]Entropy is regarded as the degree of uncertainty or potential variety of the information source that the channel, in this case the programmer, must handle [Shan49].

[19]We say most advisedly since one might anticipate certain difficulties with exception handling in a language such as Ada.

2.2.4 Hybrid metrics

As a consequence of the shortcomings of the more straightforward code based product metrics attention has been given to combining the best aspects of existing metrics. Such metrics are frequently termed hybrid metrics.

Harrison and Magel [Harr81] attempt to combine Halstead's metric with a development of McCabe's metric based on nesting level. They argue that neither metric is individually sufficient. However, when used in combination a metric results that is "more intuitively satisfying". No further validation is offered.

A similar approach was adopted by Oviedo [Ovie80] who combined control flow and data flow into a single program complexity metric. The metric was validated by applying it to a number of 'good' and 'bad' programs published in the literature. Although, a start, this hardly represents a serious effort at empirical validation.

Hansen [Hans78] proposed a 2-tuple of cyclomatic complexity and operand count, (defined to be arithmetical operators, function and subroutine calls, assignments, input and output statements and array subscription). However, the value of 2-tuples as useful metrics, has been questioned [Bake80,Cont86]. This is because comparisons are difficult to make between differing measurements, for example <a,b> and <c,d> where a < c and b > d[20].

Potier *et al* [Poti81] describe an intriguing method of by-passing the problem of n-tuples by constructing a decision tree. In their study of error data, both software science and cyclomatic measures are combined, in order to identify error prone modules. Using non-parametric discriminant analysis they identified various threshold values for the different metrics that were then entered into a decision tree. Curiously, program vocabulary, n, was found to be the metric most effective at discriminating between reliable and error prone modules and was thus placed at the top of the tree[21]. Even when using decision trees to combine metrics into a composite approach one metric tends to predominate, namely the one applied at the root node of the decision tree. Consequently the approach may not always be applicable.

---

[20]More formally we do not have closure of > relation and thus we cannot generate even a weak order. These problems are addressed in the section describing classical measurement theory in Chapter Four.

[21]It is not reported whether LOC was as examined as a potential discriminant although given the widely discovered association between LOC and the software science measures one might suspect that it would perform well. This also highlights a problem of using statistically driven metrics in that extremely bizarre models may emerge - as in this case where it is difficult to see the impact of n upon the number of errors other than as a proxy for module size.

Arguably, the most extreme variant of the hybrid approach is the one put forward by Munson and Koshgoftaar [Muns90] in the form of their relative complexity metric. They state that:

> *"unlike other metrics, the relative complexity metric combines, simultaneously, all attribute dimensions of all complexity metrics"*[22]

The approach is entirely statistical in that it is based upon the factor analysis of an arbitrary set of code metrics, without regard for the meaning of the base set metrics. Consequently as they state there is no limit to the number of metrics that might be combined. What the resultant relative complexity metric means, is a quite a different proposition.

Despite the very real difficulties of integrating metrics, Kafura and Canning [KafC85] argue that:

> *"The interplay between and among the resources and factors is too subtle and fluid to be observed accurately by a single metric or a single resource."*

Basili and Rombach make a similar argument [BasR88] in that a single metric is seldom adequate to capture software properties of interest. This creates something of a dilemma. The vector approach to software measurement leads, at best, to semi-orders. On the other hand single metrics in isolation are too simplistic to provide adequate explanations for software engineering phenomena. This would seem to suggest that either metrics and models should be restricted to simpler areas and facets of software engineering. Alternatively we must start to regard software more as a system, by which many factors and resources are integrated. Should the latter be accomplished this would provide the basis for at least weak ordering[23].

2.2.5 Code metrics summary

In short, despite attracting a considerable level of attention in the research community, none of the code metrics described above can be regarded as particularly promising. The recurring pattern is either one of researchers correlating metrics (applying different counting rules) to differing software quality factors and obtaining divergent results. Strong associations with program size measures appear to be the only invariant results.

---

[22] The emphasis is mine.

[23] The reason why measurement vectors do not permit the generation of at least weak orders is that there is no underlying model with which to link the individual vector elements. The relationship between measurement and metric is explored in chapter four.

Some metrics might be useful when tailored to specific environments and problems, but as a general means of obtaining insights into, and as a means of combating software complexity, they have little to commend them. An important reason for this state of affairs, is the highly simplistic uni-dimensional models of single resources that underlie the metrics so far described. This, coupled with their late availability suggests that attention is better directed towards design and specification metrics.

2.3 Design metrics

Unlike code metrics, design metrics can be obtained at a much earlier stage in the software development process. If necessary, the design can be reworked in order to avoid anticipated problems with the final product, such as high maintenance costs, without the need to abandon code and thus waste a great deal of development effort. Early feedback has been the main motivation for work in the field of design metrics.

Most interest has centred around structural or architectural aspects of a design, what is sometimes termed high level design. The architecture *describes the way in which the* system is divided into components, and how the components are interrelated. Some measures also require information extracted from low level design (i.e. the internal logic of design components, often expressed as a program design language). It is perhaps surprising that there has been little consideration of database systems where there is little functionality and most of the design effort is directed towards the data model. An exception is deMarco's Bang metric [deMa82] that is derived from an entity relationship diagram. This is described more fully in a later section.

There are two general problems that all design metrics encounter: namely the lack of sufficiently formal notations and validation difficulties. Ideally a metric should be extracted automatically; certainly all the relevant information must be available. However, software engineers tend to use a wide variety of notations, many of them informal, for instance by placing excessive reliance upon natural language descriptions. This makes it very difficult to extract meaningful measurements. To counter this, a number of special purpose notations have been proposed [Bowl83, Bean84, Ince84] or conformance to suitable existing ones such as module hierarchy charts [Yin78, Beny79, Chap79]. Another alternative has been to infer design or structural properties from the resultant code [Henr81a]. Such an approach must be considered a last resort since the advantages of early feedback are squandered.

Validation difficulties, in particular separating characteristics arising from the design from characteristics induced at a later stage, such as coding errors and poor testing, are in part responsible for paucity of empirical validations of design metrics. It is perhaps

unfortunate, as Belady [Bela79] remarks that there exist two sub-cultures within the software engineering community: the "speculators" and the "doers". As far as design metrics are concerned the "speculators" are in a substantial majority.

Initial work [Hane72, Chan74, Myer75, Soon77] although of undoubted value, suffered from the disadvantage that they were not fully objective. Estimates of one kind or another are required. A crucial aspect of a metric is that the measurement is objective, quantifiable, decidable and repeatable. Since software systems are frequently very large it is desirable that the metric can be obtained automatically by a software tool. None of the above are candidates for objective, automatable product metrics and will not be considered further by this thesis.

Other more recent approaches are potentially automatable. The almost universal model adopted is based upon the idea of system complexity formulated by the architect Alexander [Alex64]. This was adapted for software development in the functional design methodology of Stevens *et al* [Stev74], in particular their design evaluation criteria of maximising module cohesion and minimising module coupling. Cohesion may be regarded as the singleness of purpose or function of a module. A module that plays the national anthem and solves crosswords has a low cohesion because it performs two functions that are completely unrelated. This is generally considered to be an undesirable property of a module, since, it will result in it being harder to understand and more difficult to modify. In an informal sense we can predict that if a design comprises of modules with low cohesion this will result in various undesirable properties in the final product. Coupling is in many ways the corollary of cohesion. It is the degree of independence of one module form another. Minimising connections between modules makes them easier to understand and update.

Within this general framework, a number of different design metrics have been proposed. They *differ mainly in the detail of how best to capture coupling and cohesion* and from what notation they are best measured from. Metrics either capture aspects of a design that are internal to individual modules and these we term intra-modular design metrics, or they deal with the relationships between modules, termed inter-modular design metrics or they deal with both. The following discussion deals with each family of design metrics in turn.

2.3.1 Intra-modular design metrics

The first family of metrics are those that purely deal with aspects of *intra* modular complexity and contain two exceptions to the general model described in that they are both extensions to Halstead's software science [Szul81, Reyn84]. These allow the

designer to estimate the various measures such as n_1 and n_2 prior to the completion of code in order to calculate the software science metrics. Neither have been extensively tested and both suffer from inherent weaknesses of Halstead's model discussed earlier in this chapter.

Another metric in the intra modular measurement family is Emerson's cohesion metric [Emer84]. This is based on module flow graphs and variable reference sets. The aim is to discriminate between the different types of module cohesion that Stevens *et al* [Stev74] describe. However, the metric is unvalidated apart from the author's observation that for 25 published modules [Kern78] the metric indicates high levels of cohesion. This is justified on the basis that the modules are intended as examples of "good design" and therefore could reasonably be expected to be highly cohesive.

All the metrics in this family are severely disadvantaged by requiring knowledge of the internal details of each module in question. Unfortunately, this is unlikely to be available before coding is well under way, if not complete. Although such metrics may have a role, they cannot be considered true design metrics.

2.3.2 Inter and intra modular design metrics

The second family of design metrics are those based upon a combination of *inter* and *intra* modular measurements. The general rationale for this approach is that the total complexity of a design is a function of the sum of the individual complexities of each design component and the manner in which these components interrelate.

Probably the most widely known design metric in this family is the information flow measure [Henr79, Henr81a, Henr81b, KafH81, Henr84] which attempts to capture the structural complexity of a system and to provide a specific, quantitative basis for design decision making. Henry and Kafura considered that the prime factor determining structural complexity was the connectivity of a module[24] to its environment. Connections are defined as channels or information flows whereby one module can influence another.

The following types of information flows are defined, where in each case there is a flow from module A to B:

---

[24]Strictly speaking we should use the term procedure since in Henry and Kafura's terminology "module" is applied to the set of all procedures that reference a particular data structure. We will not do so in order to avoid suggestions of specific programming language dependence and to keep to generally accepted nomenclature.

i) *local flows* which may either be *direct* when module A passes parameters to B (see Fig. 2.2); or *indirect* when A returns a value to B (see Fig. 2.3); or module C calls A and B and passes the result value from A to B (see Fig. 2.4);

ii) global flows where module A writes to a data structure DS and B reads from DS (see Fig. 2.5).

A module's connections to its environment are a function of its fan-in and fan-out. The fan-in of a procedure is the number of local flows that terminate at that procedure plus the number of data structures from which information is retrieved. The fan-out is the number of local flows that originate from a procedure plus the number of data structures updated. The total number of input to output path combinations per procedure is given by:

```
( fan-in * fan-out )                                    (13)
```

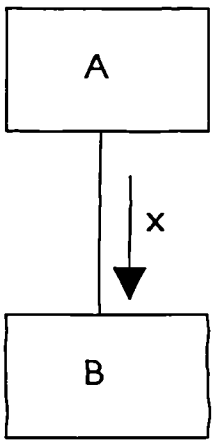


Figure 2.2: Local information flow from module A to B

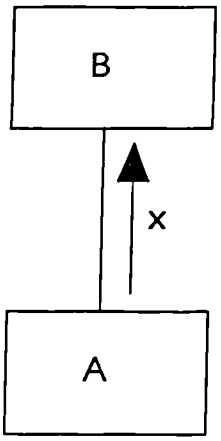


Figure 2.3: Local information flow from module A to B

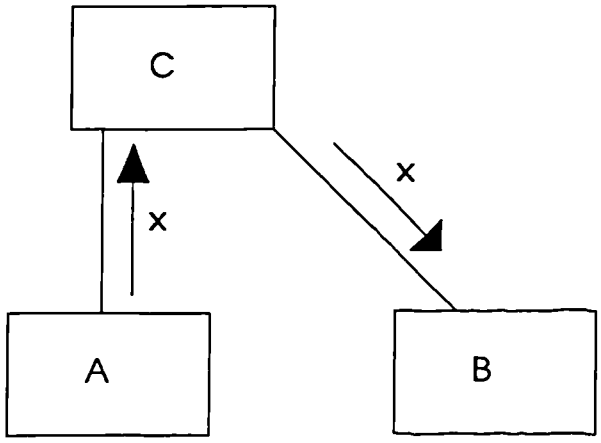


Figure 2.4: Indirect local information flow from module A to B, via C

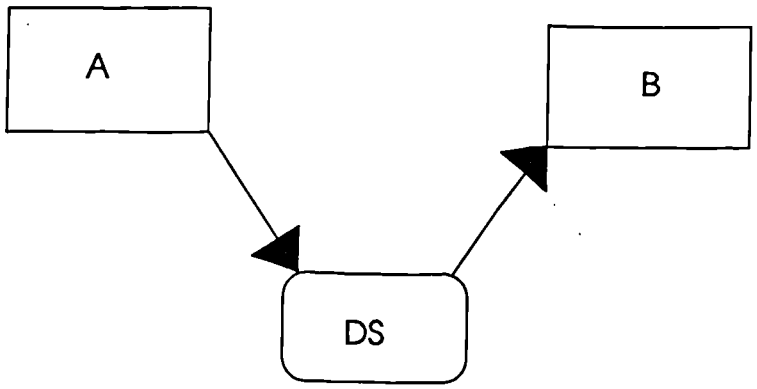


Figure 2.5: Global information flow from module A to B via data structure D

This is given a weighting of raising by the power of two in order to reflect their belief that connective complexity is a non-linear function.

$$(\text{fan-in} * \text{fan-out})^2 \qquad (14)$$

This complexity is combined with the internal complexity, measured by LOC, of the procedure to give a measure of procedure complexity as:

$$\text{length} * (\text{FAN-IN} * \text{FAN-OUT})^2 \qquad (15)$$

A multiplicative relationship in Equation 15 was adopted between internal and connective complexity as the two were considered to be orthogonal and because it indicated the number of potential information flow paths through the module.

A number of applications are suggested for the information flow metric. It may be used to identify potential problem modules by concentrating upon outliers (i.e. those with abnormally high complexities). This technique has been used with some success in a recent case study type validation [KafR87]. Another application described by the authors in their original study of the UNIX operating system [Henr84] is the analysis of metric trends between levels in a calling hierarchy of procedures. A sharp increase in

complexity between levels indicates design problems, possibly a missing level of abstraction.

The type of problem that a designer might hope to identify include:

i) lack of cohesion (i.e. more than one function);
ii) stress points where there is a high level of "through traffic";
iii) inadequate refinement (e.g. a missing level of abstraction);
iv) overloaded data structures where there is a need to segment.

Henry and Kafura have applied their metric to the UNIX operating system and have had some success in identifying problem areas. They also found a high correlation (r=0.95) between information flow and number of errors - measured as the number of program changes [Henr81b]. Interestingly, they found that the procedure length component of the metric actually detracted from its performance and that Spearman correlation of r=0.98 was obtained without the inclusion of length in the metric. Encouragingly, Rombach [Romb87a] was also able to report high correlations between information flow and various aspects of maintenance work. Kafura and Reddy [KafR87] also report the metric to be a useful predictor of effort to implement a software change. Unfortunately two more recent studies were unable to confirm the efficacy of the metric. Kitchenham [Kitc88] found that both LOC and decision counts were better predictors of errors and program changes than information flow[25]. Ince and Shepperd [Ince89a] did not find statistically meaningful correlations between information flow and development effort without making changes to many of Henry and Kafura's original definitions.

Relative to other design metrics information flow is well tested. The empirical results are somewhat mixed although the work of Ince and Shepperd suggests that refinements to the basic model produce considerably improved results in terms of the ability to predict software quality factors of interest. A more detailed analysis of the underlying model is presented in the next chapter.

A similar metric is proposed by Card and Agresti [Card88] which explicitly identifies total design complexity as comprising of inter-modular or structural complexity plus the sum of all the intra-modular or local complexity. The structural complexity is given as the sum of the squares of individual module fan-outs. The fan-in is disregarded as previous empirical work [Card86] showed it to be insignificant, coupled with the problem that counting it penalises module re-use. Local complexity for a module is the number of imported and exported variables, divided by the fan-out plus one. The rationale for this, is that the greater the number of arguments, the greater the module

[25]The Kitchenham study did not use exactly the same definitions as Henry and Kafura nor were indirect local flows of information included.

workload. On the other hand, the greater the module fan-out the greater the proportion of this workload that is distributed to other modules. Local complexities are then summed across the system.

An empirical analysis by the authors of this design metric found a correspondence between their measure and a subjective design quality rating [Card88]. In addition, they also obtained a significant correlation between the metric and error density. An attempt to employ the local complexity metric as a means of measuring module size (in terms of ELOC, decision count and variable count) found the metric to be almost orthogonal to module size [Shep89d, 90b]. This is a rather disturbing discovery. Plainly much more empirical work is required before confidence can be placed in these metrics.

Carma McClure [McCl78] argued that complexity accrued from the number of modules that modified control variables (i.e. those program variables that make up predicates for branching decisions). Ideally control variables would be local to the module that utilises them. Unfortunately little evidence is proffered to support this hypothesis, other than a case study analysis, of a small database management system, where some relationship was found between the metric to subjective maintenance complexity [KafR87]. From a design metric perspective there is the additional disadvantage that one would be unlikely to have an accurate picture of all the control variable required, or their usage until coding was underway thus making the metric difficult to obtain at design time.

Another widely cited metric in this family is Yau and Collofello's stability metric [Yau78,Yau80]. This metric considers a design from the point of view of its resistance to change. In a poor design a simple maintenance change will ripple through a large number of modules. Conversely, a good design will contain the change within a single module. Clearly, a design that is made of decoupled cohesive modules will have low resistance to change.

Since maintenance tasks are so variable it is difficult to select a representative task with which to measure system stability. Yau and Collofello use a fundamental task which they argue is common to all maintenance changes; that of a modifying a single variable. Module interface and global data structure information is required to calculate the inter-modular propagation of change. Additionally, detailed knowledge of the internal structure of each module is needed to calculate the intra module change propagation. As they observe, the metric cannot be used as the sole arbiter of good design since a single module of 20,000 LOC will contain most maintenance changes, however it may lead to many very undesirable side effects! Although an attempt has been made to infer the metric from purely design information [KafR87] the results were considered unreliable. So, it would appear that to calculate the worst case ripple effect, code is required. This is a major drawback with an otherwise novel and promising approach.

A more recent contribution, is McCabe's family of design metrics [McCa89] which is derived from the module calling hierarchy (inter-modular input) and psuedo-code (intra-modular input) to describe the control flow of the module invocations. The approach is graph theoretic in an analogous fashion to their better known code metric counterpart, *viz.* cyclomatic complexity [McCa76]. The metrics are module design complexity, design[26] complexity and integration complexity. Each of these will be briefly reviewed.

Module design complexity, *iv* is based upon the internal logic of a module as might be captured by pseudo-code. This can be depicted as a flow graph in the usual manner [McCa76, Fent85] and then reduced in order to arrive at a graph of only those paths that contain module invocations. Module design complexity is deemed to be the cyclomatic number of such a reduced graph. Consequently, all leaf modules will have an *iv* value of one.

Design complexity, S_O is the sum of the module design complexities for a module and all descendants, excepting modules that are called from more than one point within the design in which case the module is only counted once.

Integration complexity is the basis set of module invocation subtrees. McCabe uses the basis set, rather than the actual number of distinct subtrees, since unbounded iterative module invocations lead to the latter being uncountable. His argument is - as *per* the code metric - that the basis set when taken in linear combination, can yield all possible subtrees and therefore is a good indicator of testing effort, or in this case integration testing effort.

It would seem that McCabe's main concern is that of testing and to that end he offers a "structured integration testing methodology" [McCa89]. Unfortunately this leaves one a little unclear as to the meaning of the design metric, S_O. The approach is interesting in that it recognises that software testing effort is strongly dependent on structural features but is possibly handicapped by the need for intra-modular information, thus delaying its availability. The absence of any published empirical support for the metrics must also regarded as disappointing.

2.3.3 Inter-modular design metrics

The third and final family of design metrics are those based purely upon inter-modular considerations. The simplest of these is a metric based on graph impurity of Yin and Winchester [Yin78]. Their complexity metric is based upon the design notation of a

[26]This refers to the complexity of the overall design.

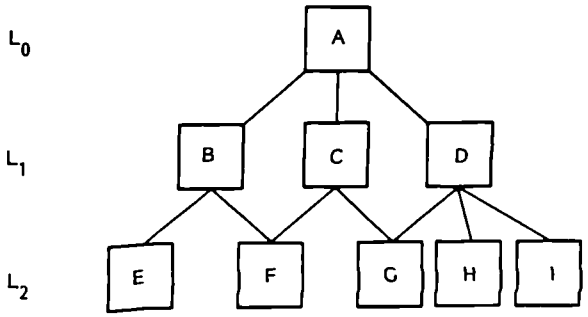
module hierarchy chart extended to include global data structure access information. This is treated as a graph (or network). Design complexity is deemed to be a function of how far the network departs from a pure tree (i.e. its graph impurity). Essentially this is measuring the complexity of connections within the design and giving a crude indication of module coupling. Figure 2.6 illustrates a simple example.

$$C'_i = A'_i - T'_i \qquad (16)$$

where: N'_i = no. of modules and data structures from L_0 to L_i

T'_i = no. of module and data structure tree arcs from L_0 to L_i (i.e. $N'_i - 1$)

A'_i = no. of module and data structure network arcs from L_0 to L_i



$N'_2 = 9$

$T'_2 = 8$

$A'_2 = 10$

$C'_2 = 10 - 8 = 2$

Figure 2.6: An example of a graph impurity measurement

Although the designer should seek to minimise C'_i where a choice exists, this should not override the re-use of components where possible. The other application of this metric is to examine trends between levels within a hierarchy, as with the Henry and Kafura information flow metric previously described.

A validation of C'_i against two projects at the Hughes Corporation produced a high positive correlation between the metric and the error count. As Yin and Winchester note, this was in part due to the effect of a small number of outlier modules on the correlation. Once these were removed a correlation coefficient of r=0.52 was obtained

indicating that the metric has some statistical significance but cannot fully "explain" the error count. In many ways this is not surprising since the metric is based on a very naive model that departure from a pure tree structure is the major determinant of complexity. In the author's opinion it is more likely that the graph impurity is a proxy for size (i.e. strongly dependent) particularly since the metric is not normalised for size, and therefore what is being observed is an example of cross correlation.

Another graph based metric was proposed by Benyon-Tinker [Beny79] after his study of a large commercial software system written in Algol. Again modules are represented by nodes and calls by edges. In this case the assumption is made that if a module is invoked more than once this need not be reflected by the metric because the module, or sub-graph representing it, has already been analysed and presumably understood. Thus the graph of what Benyon-Tinker called distinct nodes will in fact be a pure tree since whenever a module is invoked for a second or nth time this will not be incorporated into the graph.

Complexity, C for a whole program is given by;

$$C_a = \left(\sum_{1}^{m} n_r . r^{\alpha} \right) / \left(\sum_{1}^{m} n_r \right) \qquad (17)$$

where: n_r = no. of distinct nodes at level r
m = maximum depth of the tree
α = power law index

Complexity is a function of the depth and breadth of the tree. Benyon-Tinker's study lead him to suggest that values for α, the power law index, of in the range 2-3.

The empirical evidence to support this metric is based upon subjective evaluation. Benyon-Tinker used the metric to identify potentially problematic modules which he then found corresponded well with the user's subjective, but well informed judgement. Although this constitutes a basis for validation, it cannot be considered sufficient in its own right.

Adopting, a slightly different approach, though still based upon inter-modular measurements, is Chapin's Q metric [Chap79]. In this case he not only identifies the inputs and outputs to each module (i.e. the module interface) but also attempts to give a weighting factor dependent on the purpose of the data since this influences the complexity of the module interface. The following types of data are identified:

'P' data - inputs required for processing;
'M' data - inputs that are modified by the execution of the module;
'C' data - inputs that control decisions or selections;
'T' data - through data that is transmitted unchanged.

Module complexity is calculated as:

$$Q_m = (3C_m + 2M_m + P_m + 0.5T_m).(1 + (E/3)^2) \quad (18)$$

where E is a term that represents the additional complexity that accrues when a module communicates with another and is invoked iteratively. E is zero except where a module contains the exit test for an iteratively invoked module. For each 'C' data item that is imported that is used in the iteration exit test E is incremented by one if it originates from a subordinate module and two if it is from a non-subordinate module.

Although the different weights are rather arbitrary they could be refined by careful empirical analysis. The main problem is the usual one of a complete lack of empirical validation, coupled with the difficulty of automation since the interface data has to be categorised manually.

Beane *et al* [Bean84] also propose a metric based upon module connections and additionally suggest a design notation from which the necessary measurements may be culled. This is a special purpose design language called Component Interaction Language (CIL) which can be used to describe the structural aspects of a design in a hierarchical manner.

From this information a number of metrics can be generated at an early stage of the software life cycle in order to alert the designer of possible problem areas, to make comparisons between alternative designs and to predict development effort. Beane *et al* suggest two metrics that are of particular interest:

i) stress point metric - the no. of direct connections to a part divided by the mean no. of connections per part, the direction of the connection being ignored;
ii) path metric - sum of all path lengths through the system or area of interest including indirect connections.

A small, subjective case study type validation suggested that these two metrics can be applied in conjunction with a variety of design methodologies, due to the design independent notation CIL. The most convincing use was found to be as an evaluation tool, comparing alternative design solutions.

An approach rather different from the previous metrics is based upon an exploratory data analysis technique known as cluster analysis. The technique attempts to group objects together on the basis of similarity so that most similar objects will grouped together first. The output is usually a dendogram or cluster tree which reveals the order of clustering. A number of researchers [Bela81, Hutc85, Selb88, Ince89b, Ince90a] have tried to harness this method towards the generation of some idealised module hierarchy based upon the principle of grouping the most similar modules closest together, within the module hierarchy. The usual indicator of similarity has been taken to be module couplings or shared information flows. Although, this is an intriguing idea there is a long way to go before this class of metric reaches maturity. Possibly the greatest stumbling block is that the shape of the dendogram is highly dependant upon the choice of clustering algorithm.

The validity of concentrating upon inter modular measures and connections in particular has been considerably strengthened by the empirical work of Troy and Zweben [Troy81]. Their study of 73 designs and associated implementations indicated that those measures related to module coupling were most effective at predicting the incidence of errors. These measures included the number and type of module interconnection and the number of global data structure references, and in fact form the basis of the majority of design metrics reviewed above.

2.4 Specification metrics

The benefits of design metrics, in that they provide early feedback about the developing software product, are even more evident for metrics derived from the product specifications. Correspondingly, the difficulties are greater and research in this field is still scarce.

Albrecht [Albr83] suggests a simple metric based upon a count of "function points". A specification is analysed to identify the different functions described, and these are given weightings according to the relative complexity of the function type. For example, interfacing with external systems is considered to be more complex than processing queries. The total number of function points for a specification is given by:

$$(4*I) + (5*O) + (4*E) + (7*P) + (10*F) \qquad (19)$$

where:

I = no. of external input types
O = no. of external output types
E = no. of enquiries
P = no. of external files (program interfaces)
F = no. of internal files (i.e. those generated, used and maintained by the program).

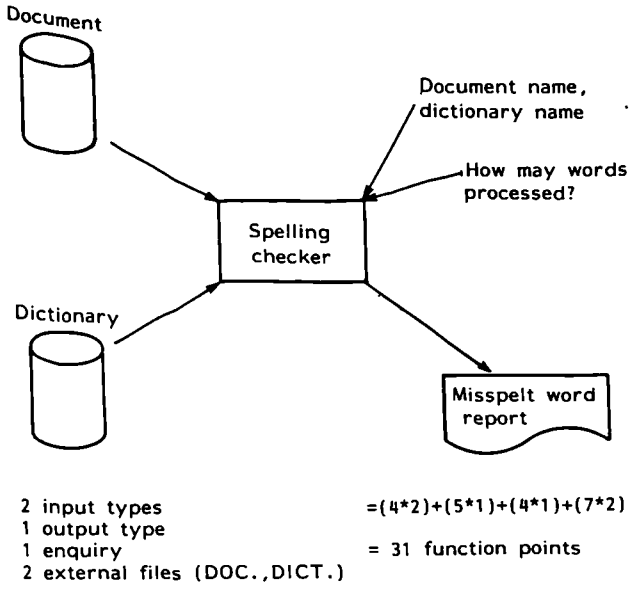


Figure 2.7: Function point analysis of a simple spelling checker

Consider the example of a simple spelling checker in Figure 2.7. From the following simple specification the function point metric can be extracted. The checker accepts a document file as input and lists all the words that are not contained in a dictionary file. The user is able to query the number of words processed whilst the checking is in progress, a feature which is useful when long documents are being examined. It is important to avoid counting items twice, thus in this example DICT is not treated as an input type because it is accounted for as an external file.

Albrecht suggests that an adjustment of up to plus or minus 35% can be made according to complexity. Such "fudge factors" can be something of a two edged sword if not carefully controlled. This is because they can permit excessive subjectivity and *post hoc* adjustments or rationalisation to creep into the metric.

The function point metric is usually used as a predictor of development effort, although there seems no reason in principle why it could not be used to predict characteristics of

the final software. For useful predictions to be made, function points require calibration so that function points are converted into units of interest (e.g. person days). Since organisations and software development environments differ so widely this may well be best done at a local level [Keme87]. Symons describes some modifications to the basic approach [Sym88].

Function points have been used very successful at a number of different sites [Behr83], and once calibrated found to be able to "explain" 75% of the variation in program size in a study of 15 commercial software systems [Keme87]. There are, however, drawbacks. The weightings in the formula are fairly arbitrary and may need considerable modification according to the type of application, experience and ability of the software developers, the development environment and the programming language. Identifying functions from the specification can be a rather subjective process, particularly if working from an informal and unstructured specification (e.g. where excessive reliance is placed upon natural language). Low and Jeffery [Low90] report up to 30% variation from the mean for counting points in an experiment based upon 22 analysts working from the same specification. Notwithstanding these difficulties the metric has been an adopted by a considerable number of commercial organisations, although this has tended to be most successful when structured specification techniques, such as data flow diagraming, are employed[27].

A more sophisticated approach to specification metrics is provided by DeMarco's Bang metric [deMa82]. The metric is derived from more formal specification notations (data flow diagrams, data dictionaries, entity relationship diagrams and state transition diagrams) thereby simplifying the measuring process. DeMarco attempts to classify software as either "function-strong" (e.g. a robotics system) or "data-strong" (e.g. an information retrieval system). In order to make this distinction more objective, deMarco suggests that it be based upon a ratio between two counts. First, the number of relationships (R) identified in the data model, usually depicted as a an entity-relationship diagram where the relationships are arrows linking entities or objects such as customer or invoice. Second, the count of primitive functions (F) identified in the data flow diagrams.

$R/F < 0.7$ = function-strong system
$R/F \leq 1.5$ = hybrid system
$R/F > 1.5$ = data-strong system

The "function-strong" system metric is derived from a count of primitive functions extracted from data flow diagrams. On its own this is unlikely to be satisfactory since

[27]One of the most successful sites to use function points has been the Inland Revenue's Telford Development Centre, where Yourdon type systems analysis methods are enforced.

individual functions vary markedly in size and complexity depending upon their task. This can be compensated for by a table of weightings that allow for variations in size and type of function. This might range from 0.3 for a function that channels data according to some selection criteria, through 1.0 for a data management function to 2.5 for a device management function. In addition the size of a function will be dependent upon the number of data flows into, and out of a function. Thus the size for function *i*, is given by:

$$SIZE_i = complexity_i(flows_i * log_2(flows_i)) \qquad (20)$$

and system size by:

$$SIZE = \sum_{1}^{F} SIZE_i \qquad (21)$$

where F = no. of functions.

The "data-strong" system metric is calculated from the count of relationships per entity (RE). This count is modified by the number of relationships each entity has with other entities. This modification is non-linear so that an entity with four relationships is more than twice that of an entity with only 2 relationships. System size is given by:

$$SIZE = \sum_{1}^{E} modification * RE_i \qquad (22)$$

where E = no. of entities.

Hybrid systems present more of a problem. The obvious approach of combining the "function-strong" and "data-strong" calculations is flawed due to the intrinsic difficulty of mixing two dissimilar measures. DeMarco argues that the most satisfactory solution is to calculate both measures but keep them separate, and partition the project into those activities that relate to system function and to those that relate to the system database.

Despite the intuitive appeal of deMarco's Bang metric there is no published empirical support for this metric. This may in part be because of the need for considerable tailoring of the metric for particular environments for which large bases of historical

development data are required. It is also based upon specific notations for system requirements in absence of which the Bang metric cannot be applied. Not unnaturally the metric is also dependent upon the quality of the inputs. Unsatisfactory and inaccurate data flow or entity-relationship diagrams inevitably lead to poor results. Nevertheless for some software development environments, the Bang metric could offer useful assistance.

Samson *et al* [Sams87] take the logical step of attempting to derive useful measurements from OBJ, a formal specification notation, thus avoiding any of the ambiguities and problems of less formal techniques. Studying a small set of modules they found significant relationships between the number of equations required to define an operator on a abstract data type and the cyclomatic complexity and length of the final implementation. Unfortunately the very small size of the empirical validation renders it rather unconvincing, especially when coupled to the observation that a given specification may be expressed in a variety of ways in OBJ, although they could all be implemented by an identical piece of software[28]. Although the idea of measuring formal specifications may have potential, particularly if it can be used to predict other more useful product characteristics than the cyclomatic number, it will be limited by the infrequent use of formal specification techniques within the software industry.

2.5 Summary

In some ways this review of current developments in software metrics has presented a rather bleak view of what is a potentially promising and important field of software engineering. There are several recurring themes.

First, the majority of metrics are presented in such a fashion that it is unclear exactly what is being measured. Terms such as "complexity" and "quality" proliferate. As a direct result, empirical evaluations have interpreted metrics in widely differing ways. Divergent results have ensued for every metric evaluated above. Almost the only unifying feature has been the strong statistical associations with size metrics such as LOC.

Second the metrics presented appear to be based upon very ambitious models of software, in that their proponents anticipated that one or two (at the most) measures

[28]A point vividly brought home to the author when trying to develop OBJ specifications in an MSDOS environment making it necessary to "shoe horn" the specification into a minimal set of equations. An initial specification comprising 65 equations was eventually reduced to 33 without making behavioural changes to the system.

would be able to predict a wide range of software quality factors in an equally wide range of applications and environments.

Third, the majority of early life cycle metrics have been subjected to minimal levels of empirical analysis. The most notable exception is the information flow metric [Henr81a], however, even in this instance the results have an ambivalence about them.

Fourth, though useful applications exist for code metrics there is little doubt that design and specification metrics are the way to proceed. This is for three reasons. First, these metrics provide much earlier feedback about the nature of the software product. Second, the metric is measuring aspects of the software that are more abstract, namely function and structure. The difficulty with code metrics is that they tend to be oriented towards specific languages or types of applications. Third, most of the design metrics, especially those based on measures that include inter modular aspects of complexity are founded on much more convincing models of software complexity. Certainly these models are more in line with current thinking in software engineering.

If the present is not that all that auspicious, what of the future? There are three emerging principles that may combine together to result in measurement becoming an integral part of the software engineering process.

- *The realisation that we should be concerned with contexts as well as the metrics in order to have meaningful measurement.*

- *Measurable software products are not restricted to code and that there many important benefits to be realised from metrics derived early on in the software life cycle.*

- *It is insufficient to propose metrics, without supporting empirical investigation, preferably from a variety of environments.*

3. PROMISES AND PROBLEMS

Geuying them faire wordes, and makyng large promises"

Anon. 1548

"MENE, MENE, TEKEL, PARSIN'

Daniel ch. 5 v. 25

Synopsis of chapter

This chapter examines in more detail the metrics of the software science model [Hals77], cyclomatic complexity [McCa76] and information flow [Henr81a]. These are selected on the basis of their popularity within the software engineering literature, and the significance of the claims made by their progenitors. Claimed benefits are summarised. Each metric is then made the subject of an in-depth critique and each metric found wanting. The chapter then goes on to argue that this not mischance, but indicative of deeper problems of methodology employed in the field of software metrics. It concludes by outlining the difficulties that exist for the software metrologist, particularly with respect to design measurement. This position is then adopted as the starting out point for the research described in this thesis.

3.1 Why Software Science, Cyclomatic Complexity and Information Flow?

The need for some objective measurement of software has been long acknowledged. Two early contributions to this field are Halstead's "software science" [Hals77] and the cyclomatic complexity approach of McCabe [McCa76]. Both metrics are based upon the premise that software complexity is strongly related to various measurable properties of program code. Software complexity, is in turn interpreted as being an indicator, and often a predictor, of a variety of quality factors that might be useful to the software engineer.

Both the above measures can be regarded, with little fear of contradiction, as the cornerstones of the work of the last fifteen years in software metrics. Even a brief survey of the field of metrics reveals copious empirical studies of different aspects of Halstead's

software science model, ingenious applications, inventive permutations, its integration into more sophisticated models and more recently a growing number of critiques. It is widely cited in texts of software engineering and to many people software science and software metrics are synonymous.

One could make similar observations for McCabe's cyclomatic complexity measure. Stemming from his pioneering work, and in particular application of graph theory to software modelling, has arisen a veritable "blizzard of refinements" [Curt83]. As *per* the work of Halstead the application of the cyclomatic complexity metric, and its related offshoots, has been far ranging. It is hard to under-estimate the impact of McCabe's work and today the creation, investigation and promulgation of graph theoretic software measures is a major industry. Yet, surprisingly, in the view of the author, the usefulness of cyclomatic complexity as a software metric has been allowed to pass relatively unquestioned. Indeed it is still widely cited in textbooks [Wein84, Arth85, Pres87, Somm89], subjected to many minor "tinkerings" [Myer77, Hans78, Wood79, Iyen82, Stet84, Sinh86] and applied as a design metric [Hall84]. Yet its empirical basis would at present appear to remarkably sparse and theoretical underpining extremely suspect.

A more recent contender is the information flow metric due to Sallie Henry and Denis Kafura [Henr81a, Henr81b]. This metric has achieved a great deal of prominence. The measure may be extracted from a system design, and is thus available far earlier than the code metrics. The authors claim it may be used for prediction and also to pin-point weaknesses in the system architecture, thereby providing much needed feedback during the design process. Furthermore, it is virtually the only early life cycle metric to have received any serious empirical validation. Consequently, it is almost inevitably cited in current papers on system design measurement[1].

The three metrics *viz.* software science, cyclomatic complexity and information flow, have therefore been selected as being representative of a great deal of current work in the software metrics domain. Moreover, much additional work treats these metrics and their underlying models as fundamental. In addition these metrics are the most widely applied, generating a considerable amount of empirical evidence. The approach has been adopted that since these metrics may reasonably be regarded as typical, it is appropriate to subject them to more detailed scrutiny; and, that findings relating to these metrics may be treated as representative of the field.

---

[1]Even Navlakha's woefully inadequate survey of design metrics [Navl87] managed to include reference to the work of Henry and Kafura!

3.2 Software science and the magical number eighteen

Halstead's so called software science [Hals77] was one of the earliest attempts to provide a code metric based on a coherent model of software complexity. The underlying concept was that software comprehension is a process of mental manipulation of program tokens. These tokens can be either operators (executable program verbs such as IF, DIV and READ) or operands (variables and constants). Thus a program can be thought of as a continuous sequence of operators and their associated operands[2].

Software science attracted considerable interest because of the then novelty of measuring software. Also, the basic inputs measurements for the metric are all easily extracted automatically. A Pascal program to extract the software science metrics provided by deMarco's well known text book [deMa82] is scarcely 200 lines of code in length.

Despite the initial enthusiastic reception for software science, a number of serious problems have emerged more recently. Since Halstead published his work software engineers have carried out a multifarious attempts at empirical validation of these metrics, however, careful analysis of the results reveal a number of disturbing problems. These are summarised in Table 3.1.

[2] A more detailed account of the individual software science metrics is provided in the previous chapter.

Study	Corr. with LOC	Better than LOC?	Resource	Useful[3] predictor?
[BasP81]	n.a.	=	effort	WEAK
[BasS83]	n.a.	*NO*	*bug location*	*WEAK*
	n.a.	NO	effort	WEAK
[Bowe78]	n.a.	NO	errors	NO
	n.a.	NO	bug location	NO
[Curt79a]	YES	NO	program recall	NO
[Curt79c]	YES	YES	bug location	YES
[Elsh76]	n.a.	n.a.	length	YES[4]
[Evan84]	n.a.	n.a.	program style	NO
[Funa76]	n.a.	n.a.	errors	NO[5]
[Fitz78b]	n.a.	n.a.	debugging effort	NO[6]
[Gord76]	n.a.	n.a.	effort	YES
[Hals77]	n.a.	n.a.	effort	NO[7]
[Henr81b]	YES	n.a.	changes	YES[8]
[Lind89]	YES	NO	effort	WEAK
[Otte79]	n.a.	n.a.	debugging effort	NO[9]
[Shep79]	YES	NO	effort	NO
[Wood80]	n.a.	n.a.	effort	*WEAK*

Table 3.1. Empirical validations of software science

First, different researchers have applied the metric to a large number of different software attributes, ranging through ease of maintenance, number of errors, program recall, development time, to documentation effort. This results in difficulties in

[3] This is rather a subjective judgment on the part of the author. A threshold of $r^2 = 0.4$, modified by considerations of experimental quality has been adopted. The WEAK classification is applied where the correlation is statistically significant but does not meet the above criterion.

[4] These findings have subsequently been criticised by Card and Agresti [Card87].

[5] Hamer and Frewin [Hame82] reveal statistical and experimental errors in this work.

[6] Hamer and Frewin [Hame82] reveal statistical and experimental errors in this work.

[7] Halstead's results although superficially supportive, have been demonstrated to be flawed [Hame82].

[8] The experimental and statistical techniques employed are somewhat suspect for this study. The section reviewing Henry and Kafura's information flow metric provides more details upon this matter.

[9] Ottenstein was only able to obtain significant correlations by multiplying her results by the fraction of bugs in the project reaching validation.

comparison of work, and evaluation of the metric since it is unclear precisely what the metric is addressing, or if it is intended as a *General Model of Software* which is capable of predicting almost any aspect of software on the basis of program operand and operator counts. Kearney *et al* [Kear86] outline some of the difficulties inherent in such an approach. Factors that reduce development effort may not result in more reliable and maintainable software.

Second, even where there is agreement as to which *aspect of software engineering* Halstead's metrics are addressing, they and many of their associated empirical validations, encounter problems of what exactly to include within the ambit of their studies. For example, what is meant by development time? Does this include time spent analysing requirements or time spent upon design work? Should effort expended on documentation be included? What about the scenario where the developer lies awake at night pondering some particularly intractable problem? These issues are not addressed by software science because the underlying model is couched in terms too nebulous to admit more precise definition.

Third, a seemingly simple task as counting operands and operators is fraught with difficulties. This is because Halstead relied upon our intuitive understanding of these concepts. Clearly this is not adequate, particularly for what is the primary input to the metric. This alone should cause us to treat empirical results with a certain degree of caution. A more formal definition is required of these counts and the mapping process from code to the metric.

Fourth, there is considerable *disquiet concerning the quality of many empirical* validations and their associated statistical analyses. Hamer and Frewin [Hame82] re-examine the experimental data from two studies [Hals77, Come79] that claim high correlations between E and actual programming time. They report that coding time is not proportional to E, rather that it is proportional to the square root of E. Even applying this relationship, however, still yielded an unacceptable level of inaccuracy. They also revealed flaws in Halstead's approach to larger systems, as on certain occasions the E metric was obtained by calculating the metric for individual modules and summing across the system[10]. On other occasions the metric was obtained by treating a system as a single module[11]. As Hamer and Frewin note that this "minor" error results in a 20-fold over-estimate of effort required and thus casts a rather substantial shadow over the claimed confirmation. Parenthetically, we note this type of error is unsurprising given the extremely restricted view of software that Halstead's model embodies, being originally concerned only with trivial FORTRAN algorithms. It is also unfortunate that the

[10] This is the approach advised in what may be taken as Halstead's definitive work [Hals77,pp47-48].

[11] This is the approach adopted to analyse the Walston and Felix data taken from 60 IBM projects, and also presented in [Hals77] as a major confirmation of software science.

majority of studies deal with very small scale programs and use non-professional programmers and in one case a single student [Gord76].

Discrepancies have also been identified by Hamer and Frewin between the analyses of Funami and Halstead [Funa76] and Ottenstein [Otte79] derived from the *same* source, *viz.* Akiyama's debugging data. The Funami study used mis-reported data and incorrectly estimated n_1 and n_2 but nevertheless obtained a high correlation with the actual debugging data. By contrast, Ottenstein correctly applied the software science model and obtained results that did not agree with the actual data, and as a consequence multiplied her estimates by a project dependent constant in a manner entirely inconsistent with the software science model[12].

Card and Agresti [Card87] also argue that many of the impressive empirical results supporting software science are a more apparent than real. More specifically, they show that the high correlations found between estimated and actual program length are the consequence of the fact that N and N^ are dependent by definition, thus a positive correlation must exist.

In addition to questionable empirical support are a number of theoretical objections. Firstly, there are serious problems with the definitions of operators (n1) and operands (n2). Many of Halstead's counting rules appear rather arbitrary; it is not obvious why all I/O and declarative statements should be ignored, particularly as in many languages this can be a significant part of the total development effort. A COBOL programmer may not be in full agreement with the view that all non PROCEDURE DIVISION statements are mere "syntactic sugar" [Otte76]! The treatment of GO TO <label> as a unique operator for each unique label, is quite inconsistent with the treatment of IF <condition> as a single operator irrespective of the number of unique conditions [Lass82]. Overloaded operators cause further problems, for example in a language like Ada. These counting problems are significant because software science metrics are sensitive to rule changes [Bese82, Shen83] which is rather disturbing since it implies that results are dependent on arbitrary decisions rather than the underlying model[13].

A second area of objection are the psychological assumptions that the model makes. The volume metric is thought suspect given the lack of empirical evidence for a binary search mechanism, within the context of programming [Coul83]. Just as serious, given its

---

[12] Hamer and Frewin [Hame82] present a table comparing the differing results which seem to have excited no comment other than that both studies lent support to the Halstead's work. Clearly such support is entirely illusory.

[13] The dangers inherent in such a situation are well illustrated by the decision of Balut *et al* [Balu74] to treat each GO TO as a unique operator was done without justification other than it improved the correspondence between N and N^. The consequence of this ad hoc decision was the introduction of new anomalies into software science.

potential use for software engineers, are the problems with time equation T=E/S where S is the so called Stroud number, after the psychologist Stroud [Stro67]. Stroud stated that:

> *"there are approximately ten moments of psychological time for every second of physical time, though there may be more; as many as twenty, or less, or few as five".*[14]

The adoption of a value of 18 for S, despite the fact that it improves correlation coefficients [Hals77], must be regarded as arbitrary. Furthermore, Stroud's studies were confined to sensory memory and there is no good reason to suppose that computer programming can be treated in the same way. Indeed, there are a number of more recent models that would suggest otherwise. Apart, from Halstead's work [Hals77] and a study by Ottenstein [Otte79] there has been little other validation of the conjecture S=18. Zweben and Fung [Zweb79] report difficulties with S and questioned the assumption of its being constant.

Thirdly, the view of software as a sequence of tokens is very simplistic as it ignores control structure, program structure and data structure. Lassez *et al* [Lass81] point out that for many programming languages operators and operands cannot be considered to be mutually exclusive (e.g. the use of procedure calls as parameters). They dispute that such a simple characterisation of software is an appropriate model, and argue that a distinction should be made between operators that control flow and those that are more functional in nature (e.g. +, DIV, etc.). The Johnston and Lister [John81] study of Pascal software and a subsequent critique [List82] of Halstead indicate that the lack of distinction causes problems, at least in some domains.

Finally, there are difficulties that relate to the scale of the software. The original work was concerned with small scale algorithms, for example the experiments of Gordon and Halstead [Gord76] deal with programs of between 7 and 59 statements, as opposed to large scale software systems, and therein lies the problem. One of the main tenets of software engineering is that large scale software does not exhibit similar properties to its small scale counterpart.

Although to Halstead's credit he did attempt to postulate an underlying model for software science, many of the psychological and software engineering assumptions have been challenged and shown to be without foundation. Moreover, what upon first sight appears to have been an impregnable empirical position, has also been found wanting. In particular, many experiments have been shown to be unrepresentative, produced few data points, generated results based upon inappropriate statistical techniques and been incapable of refuting the hypothesis under examination. It is also noteworthy that for a significant number of studies, software science measures have performed no better than

[14] Stroud is quoted by Coulter [Coul83]; the emphasis is mine.

LOC metrics. Given the serious criticisms that have been levelled at software science, its role within software engineering would appear to be at best marginal. This is despite widespread and uncritical reference to it in much of the literature.

3.3 Decision count plus one - alias the cyclomatic number

At about the same time as Halstead, an alternative measure of software complexity was proposed by McCabe, in the form of cyclomatic complexity [McCa76]. McCabe was particularly interested in the number of control flow paths through a piece of software, since this appeared to be related to testing difficulty and to the most effective way of dividing software into modules. Even at this juncture the dual, and not necessarily complementary, aims of McCabe's metric should be underlined.

Programs can be represented as directed graphs to show the control flow. From such a graph the cyclomatic complexity can be extracted which is the number of basic paths within the graph (i.e. the minimum set of paths which can be used to construct all other paths through the graph). The cyclomatic complexity is also equivalent to the number of decisions plus one within a program[15].

Like Halstead's software science, McCabe's metric was well received by the software engineering community; made the subject of innumerable minor variations and empirical studies. Also like software science, the metric has been widely interpreted as a general measure of software complexity, able to predict any facet of software or software development that is even remotely linked to complexity.

Unfortunately, we are confronted with an underlying model which is almost completely vacuous. In essence, it is that almost all software properties are in some way linked to the number of decisions contained within the program code. Careful analysis of the model behaviour reveals anomalous behaviour, particularly with respect to modularisation [Shep88a] and widely accepted principles of good programming practice [Evan83, Evan84]. Again, the metric relies upon our intuitive understanding of what constitutes a decision. In practice a number of significant counting issues have been thrown up. These illustrate the deficiency of the metric and the need for a more formal mapping from a program to its flow graph. Furthermore, even if the empirical evidence is accepted at face value, though it is arguable that it should not, the case for cyclomatic complexity is remarkably unconvincing. In many cases it is out-performed by the straightforward lines of code (LOC) metric (e.g. [Kitc81, Wang84a]).

The extremely simplistic view of software complexity that McCabe adopted can be challenged from a number of quarters.

---

[15] A more detailed account of how to calculate the metric is given in the previous chapter.

Firstly, he was chiefly concerned with Fortran programs where the mapping from code to program flow graph is well defined. This is not the case for other languages such as Ada. For example, it is unclear how implicit exception event handling construct can be adequately represented by flow graphs.

A second type of objection is that v = 1 will remain true for a linear sequence of code of any length. Consequently the metric is insensitive to complexity contributed from linear sequences of statements. Some workers suggest that software can be categorised as either decision or function bound [Henr81b]. The function bound software represents a major class of systems for which the metric is a poor predictor.

A third difficulty is the insensitivity of cyclomatic complexity to the structuring of software. A number of researchers [Bake80, Ouls79, Prat84, Sinh86] have demonstrated that cyclomatic complexity can increase when applying generally accepted techniques to improve program structure. Evangelist [Evan83, Evan84] reports that the application of only 2 out of 26 of Kernighan and Plauger's rules of good programming style [Kern78], invariably result in a decrease in cyclomatic complexity.

It could be argued that this argument is a specific case of a more general point that the metric ignores the context or environment of each decision. *All decisions have a uniform* weight regardless of depth of nesting or relationship with other decisions. In other words, McCabe takes a lexical rather than structural view. Modifications have been proposed that allow for nesting depth [Mage81, Piow82, Prat84].

A fourth objection to cyclomatic complexity is the inconsistent behaviour when measuring modularised software. It has been demonstrated that cyclomatic complexity increases with the addition of extra modules but decreases with the factoring out of duplicate code [Shep88a]. All other aspects of modularity are disregarded. This is contrary to current ideas of proper modularisation and causes problems with respect to McCabe's objective of helping the designer to select an effective software architecture.

The difficulties described so far could be termed theoretical, but the empirical evidence is no more encouraging.

Many of the early metric validations were based merely upon intuitive notions of complexity, for example McCabe stated that "the complexity measure v is designed to conform to our intuitive notion of complexity". Hansen stated that a good measure of program complexity should satisfy several criteria including that of relating "intuitively to the psychological complexity of programs". He does not suggest that there is a need for any objective validation. Likewise Myers [Myer77] treated intuition as sufficient grounds for employing the metric.

This strikes one as a rather curious approach in that if intuition is a reliable arbiter of complexity this eliminates the need for a quantitative measure. On the other hand if intuition cannot be relied upon, it hardly provides a reasonable basis for validation. Clearly a more objective approach to validation is required.

The metric is vulnerable to the criticism that it ignores other aspects of software such as data and functional complexity. It is easy to construct certain pathological examples, however, this need not invalidate the metric if it was possible to demonstrate in practice that the metric was a strong predictor with factors that are associated with complexity. McCabe suggested this should include effort required to test and maintain modules.

Study	Corr. with LOC	Better than LOC?	Resource	Useful[16] predictor?
[BasS83]	n.a.	n.a.	bug location	NO
	n.a.	n.a.	effort	NO
[BasP84]	YES	n.a.	error density	NO
[Bowe78]	n.a.	n.a.	errors	WEAK
	n.a.	n.a.	bug location	NO
[Curt79a]	YES	NO	program recall	NO
[Curt79c]	YES	n.a.	bug location	WEAK
[Feur79]	YES			
[Gaff79]	n.a.	n.a.	effort	YES
[Henr81b][17]	YES	n.a.	changes	YES
[Kitc81]	YES	NO	errors	WEAK
[Lind89]	YES	NO	effort	WEAK
[Paig80]	YES	NO	testing effort	WEAK
[Schn79]	YES	n.a.	errors	NO
[Shen85]	n.a.	n.a.	errors	YES
[Shep79]	YES	n.a.	effort	*NO*
[Suno81]	n.a.	n.a.	effort	WEAK
	n.a.	n.a.	design effort	YES[18]
[Wang84a]	YES	NO	effort	YES
[Wood81]	n.a.	n.a.	effort	NO
[Wood79]	YES			

Table 3.2. Empirical validations of cyclomatic complexity

As Table 3.2 indicates the results of various empirical validation studies do not lend much credence to the metric. A more detailed account is given in [Shep88a]. The clearest result is the strong relationship between cyclomatic complexity and LOC. Ironically, it was the "inadequacy" of LOC as a module complexity metric that led to McCabe proposing cyclomatic complexity as an alternative to the more traditional LOC.

[16] This is rather a subjective judgment on the part of the author. A threshold of r^2=0.4, modified by considerations of experimental quality has been adopted. The WEAK classification is applied where the correlation is statistically significant but does not meet the above criterion. This is justified on the grounds that such a coefficient of variation accounts for 40%, or less of the variation in the dependent variable.

[17] The experimental and statistical techniques employed are somewhat suspect for this study. The section reviewing Henry and Kafura's information flow metric provides more details upon this matter.

[18] Only confirmation, rather than prediction, is possible for the reason that a code metric cannot be made available until after the design has been completed!

The correlation with programming effort, although erratic, is not as damning as it appears upon first sight. Testing is only a component of programming effort, and McCabe's original paper did not suggest that the metric be used as a predictor of software development effort. Instead, the objective was to create a measurement to provide an upper limit to module complexity. Thus, counting cyclomatic complexity across entire programs, rather than individual modules, as some researchers have [Wood81a], is not entirely appropriate. By contrast, the study of Basili and Perricone [BasP84] deals with individual modules in which they looked for possible associations between the error rate and module v(G). One would expect the former to increase with the latter, presumably with a distinct step around the point of v(G) = 10. Their results suggested the complete reverse of this proposition, and although the result is counter-intuitive[19], it throws considerable doubt upon v(G) as a predictor of error proneness. A more meaningful study would be to demonstrate that v(G) is strongly correlated to testing effort, in particular unit testing, although clearly the actual testing strategy employed would have a considerable bearing upon the outcome of the investigation.

The small size of tasks being undertaken is another problem area. Both Woodward *et al* [Wood79] and Woodfield [Wood81a,b] use programs of <300 LOC which by software engineering standards are trivial. In such situations the onus is upon the researcher to demonstrate that results at a small scale are equally applicable for large systems. Such a finding would be counter to current directions in software engineering.

As equation 2 indicates, v(G) is sensitive to the number of subroutines within a program, because McCabe suggests that these should be treated as unconnected components within the control graph. This has the bizarre result of increasing overall complexity as a program is divided into more, presumably simpler, modules.

[19] A possible explanation for this finding is that it is an artifact of using LOC for size normalisation. A short module containing a single error will have a high error rate per unit length which can distort results if the overall number of errors relative to modules is not great.

Furthermore the limited work that has already been carried out is not very encouraging. The only possible role for cyclomatic complexity is as an intra-modular complexity metric. Even this is made to look rather suspect by the work of Basili and Perricone. In any case, many researchers (e.g. [Stev74]) would argue that the problem of how to modularise a program is better resolved by considerations of inter-modular complexity.

It may well be that some find McCabe's metric "intellectually very appealing" [Wood79], but there are few grounds for its widespread adoption. Careful study of the empirical evidence shows erratic support for the measure, but a rather more consistent relationship with LOC. The likely explanation is that of cross correlation. Decisions have a fairly constant incidence (for a given application domain and development environment) and therefore the cyclomatic number is a proxy for size and so correlates with such metrics as LOC. The theoretical basis is equally insubstantial. The underlying model was not targeted by McCabe at any particular facet of software engineering. Nor does it embody any notion of software structure, either at the intra-modular or inter-modular level.

One can only conclude that the utility of cyclomatic complexity as software metric is extremely restricted[20].

3.4 Henry and Kafura's information flow measure

More recently attention has focused on metrics derived from early on in the software lifecycle. The metric that has excited the most attention is the design measure proposed by Sallie Henry and Denis Kafura [Henr79, 81a, 84 etc.] known as the information flow metric[21]. It is widely considered to be the classic design metric, being more widely cited and investigated than any other design metric.

Information flow is based upon the concept that the complexity of a software module - that is a functional design unit - is a related to the number flows or channels of information between it and its environment. In addition, a module has an internal complexity which they suggested might be based on module size and measured as LOC. Thus, the model incorporates the concept of internal and external module complexity, although in practice most attention has been given to external complexity. Such ideas

[20] It could be argued that the metric might be useful where one was interested in testing effort and intended to apply some sort of path coverage strategy. Even, this simple application has been disputed by Humphreys [Hump87] due to the trade-off between decision complexity and data structure complexity, as exemplified by the use of decision tables.

[21] For a more detailed description of Henry and Kafura's metric refer to the previous chapter.

are loosely derived from the design evaluation criteria of module coupling and cohesion described by Stevens *et al* [Stev74].

As has already been remarked, the information flow metric is unusual in that it has been made subject to a number of empirical investigations. These are summarised in Table 3.3 below.

Study	Resource	Better than LOC?	Sig. corr. with LOC	Sig. corr. with Resource	Better with length?
[Henr81a]	No. of changes	YES	NO	YES	NO
[KafC85]	Coding time + errors	NO	n.a.	YES	YES
[KafR87][22]	Maintainability	EQUAL	YES	YES	n.a.
[Romb87]	Modifiability	NO	n.a.	NO	YES
	Maintainability	YES	n.a.	YES	YES
	Comprehensibility	YES	n.a.	*YES*	*YES*
	Locality	YES	n.a.	YES	NO
[Kitc88][23]	Errors	NO	WEAK[24]	WEAK	n.a.
[Shep88c]	Development effort	YES	WEAK	NO	NO
[Shep90c]	Maintainability	NO	YES	YES	YES

Table 3.3. Empirical validations of information flow

It is immediately apparent that the studies are varied in nature and have produced results that are very mixed. This is in part due to the approach of *Henry and Kafura*. Just as for the other metrics previously discussed, *their basic model does not indicate* what aspects of software or the software process that are being addressed, leaving as a

[22] The Kafura and Reddy empirical study tends to be a rather anecdotal case-study based upon subjective assessments of maintenance complexity. Nevertheless, they report that they find strong evidence to justify the use of structural metrics to identify potential maintenance problems.

[23] Barbara Kitchenham's study uses a slightly modified definition of the information flow metric which disregards indirect flows. Our empirical work suggests that this is likely to be of little consequence as regards the final results.

[24] In a private communication (March 1989) Barbara Kitchenham writes of her finding of a correlation coefficient of r=0.58 between information flow and LOC.

default a degree of universalism that may not be fully warranted. In their original work [Henr79, 81a] the metric is used to predict the number of changes per procedure within the UNIX™ operating system. These are treated as a proxy for error data[25]. However, the motivation for the work is variously reported as the high cost of software maintenance [Henr81a], the high cost of software development [Henr79,p1], improving software reliability [Henr84], providing quantitative guide-lines for the software designer [Henr81b] and controlling software complexity [Henr79,p1]. Nowhere is this confusion more evident than in the statement of the problem in Sallie Henry's doctoral dissertation:

> *"The thesis of this research is that a set of measurements based on the flow of information connecting system components can be used to evaluate software design and implementation"*
>
> [Henr79,p14].

The 64,000 dollar question - evaluate with respect to what - apparently has been overlooked. Such oversights are all too typical of the "software complexity syndrome". It is not, thus, surprising to report the varied interpretations given by differing researchers to the model. We will now review the empirical evidence in more detail.

The empirical work, cited by Henry and Kafura to support the metric, is based upon a change analysis of parts of the UNIX™ operating system. Very high correlations are reported (e.g. Spearman r=0.94, p=0.0214 [Henr81a] and Pearson's product moment[26] r=0.95 [Henr81b] between the metric and module changes[27]). A matter of slight concern is their discovery that information flow is out-performed by McCabe's cyclomatic complexity metric [Henr81b][28]. Given our previous discussion linking McCabe's measure to LOC, this is highly suggestive that a simple size measure is as good or better predictor of software changes. Closer inspection, however, raises a number of question marks over the entire empirical study.

---

[25] Henry and Kafura argue that this is a legitimate assumption citing the work of Dunsmore and Gannon [Duns77] and Basili and Reiter [BasR79] as precedents, although in her doctoral dissertation Henry does concede that the "suggested changes to UNIX consist of both actual errors and some necessary performance enhancements" [Henr79,p92].

[26] This is in actual fact an inappropriate test given the assumption of normally distributed data points is scarcely credible for change or error data.

[27] Henry and Kafura rather misleadingly refer to change data as error data on a number of occasions (e.g. [Henr81b]). A model of maintenance work is unlikely to be isomorphic to a reliability model.

[28] To complete a picture of confusion, Kafura gives different correlation coefficients for the same study [KafH81]. Reference back to the Sallie Henry's original thesis [Henr79] suggests that the Kafura paper contains typographical errors.

Table 3.4 is taken from Henry's doctoral thesis [Henr79,p94]. This reveals that of the 165 UNIX procedures examined, 80 were modified or, as Henry would have it, contained errors. Elsewhere [Henr79,p92] Henry states that a total of 80 changes were analysed. Presumably the explanation for there being exactly one change per changed procedure lies in the counting method, so that no account is given for the size of a change and the analysis covers only one UNIX update. If this is not the case, one can only comment that it is remarkable that no procedure is subject to more than one change.

Order of complexity	No. of procedures	No. of "error" procedures	%
0	17	2	12
1	38	12	32
2	41	19	46
3	27	19	70
4	26	15	58
5	12	11	92
6	3	2	67
7	1	0	0
Totals	165	80	

Table 3.4 Data from the UNIX study

The technique of logarithmic class interval analysis is a surprising statistic to employ. First, because one loses a lot of data, particularly from the higher order classes, resulting in what is at best a weak ordering. Second, it decreases the number of data points from 165 to 8 making statistical significance harder to obtain and trends less reliable. Unfortunately, insufficient data is provided to perform an alternative analysis, though we feel compelled to point out that on the basis of the above data we obtain a Spearman correlation coefficient of 0.21 and not 0.94. Figure 3.1 shows the distribution of data points as a scatter diagram. The difference is presumably due to Henry and Kafura eliminating the upper two class intervals from their analysis, the justification being that there are insufficient procedures within these classes [Henr81a] and that the procedures are too complex to be changed [Henr79, Henr81a] and thus their major result rests on a correlation of just 6 data points in rank ordering.

The other major result that Henry and Kafura present is that the length, or the internal module complexity component of their metric actually detracts from its overall

performance [Henr81a]. Without length and using class intervals based on $10^{(n/2)}$, a Spearman correlation of r=0.98 was obtained. No justification is proffered for what appear to be rather arbitrary class intervals, which is rather disappointing given their material bearing upon this style of analysis.

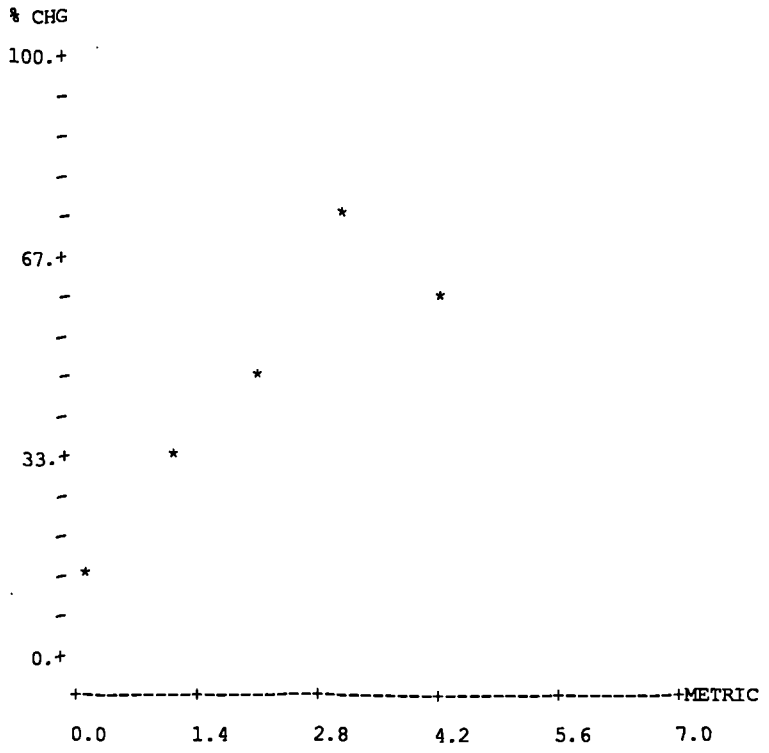


Figure 3.1: Percentage of procedures changed vs. information flow metric

These results, suggest that the information flow metric has identified some relationship between design or system architecture and software reliability, but that it is less compelling than it might appear at first sight. If one were to summarise the remainder of the results given in Table 3.3 it would be to support this view that underlying the metric is a powerful idea, but considerably more work in verification and refinement is required. Three out of the six remaining investigations find useful - from a software engineering perspective - relationships between the metric and a variety of software quality factors. A fourth study, [Romb87] indicates that for 3 out of 4 maintenance factors the metric proved to be an effective indicator. Kitchenham [Kitc88] found only weak relationships and superior performance by more traditional code based metrics. A sixth study [Shep88c] found no statistically valid relationship with either information flow or ELOC and development effort, although the design metric performed marginally better. The table is weakly suggestive of some association with LOC [KafR87, Kitc88,

Shep88c, Shep90c] and ambiguous as to the efficacy of including LOC into the formulation of the metric. Two studies [Henr81a, Shep88c] suggest that this detracts from performance, but Rombach [Romb87] finds in most cases it enhances performance, as does the Kafura and Redddy case study [KafR87]. It is fortunate that the module length component of their metric is of marginal significance, as otherwise this would diminish the ability to use information flow at design time.

One of the difficulties confronting those wishing to empirically validate the information flow metric stems from the inadequate and conflicting definitions that Henry and Kafura give, particularly concerning indirect flows. Again, as with the other two metrics, informality predominates. For example, what is a global data structure? What is a procedure? The definitions of information flows are ambiguous and capricious.

A major source of anomaly are the many local indirect flows that may only be detected by internal analysis of a module. This is shown in Figure 3.2.

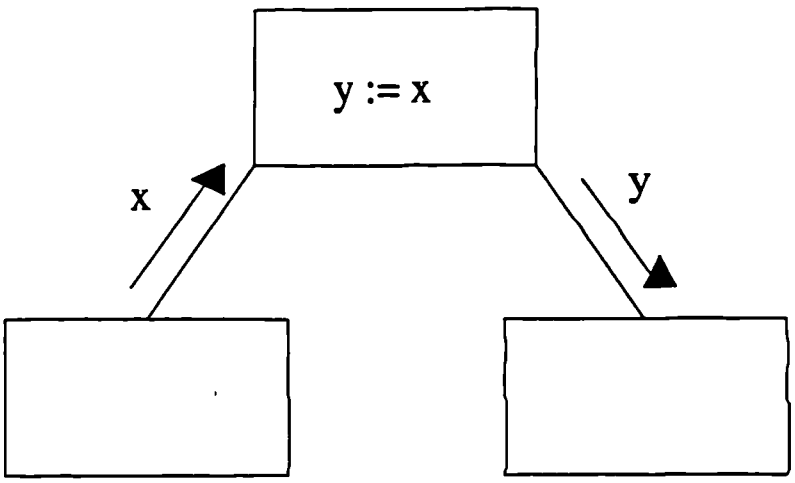


Figure 3.2: Indirect local flows "hidden" at design time

Unfortunately such information is unlikely to be available during the system design stage of software development. Even if one obtained the necessary data, a static analysis would be inadequate, as the existence of the flow is entirely dependent upon the execution order of the code. Dynamic analysis, apart from being a difficult enterprise, generates results that reflect the input chosen to exercise the code. There are no obvious guide-lines to steer the would-be-software-metrologist as to the choice of input. However, to fail to capture indirect flows-by-assignment leaves the measure vulnerable to the whimsy of the software designer in his or her choice of data object name.

The definition of local indirect flows as given by Henry and Kafura, for example [KafH81], would appear to encompass flow only over two levels of a system structure. If such flows are to be counted, there is no good reason why the number of levels should be restricted to two. For instance, Figure 3.3 shows an indirect local flow which has a scope of three levels.

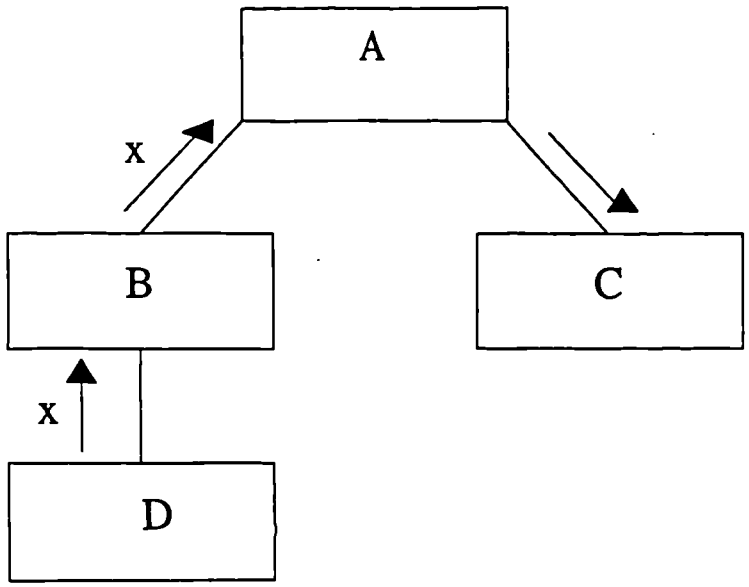


Figure 3.3: Indirect local across more than two levels

However, simply extending the number of levels included is unsatisfactory since the outcome is potential over counting particularly if the problems over indirect flow-by-assignment, described above, are addressed. The consequence would be that all modules are linked to all other modules except where a system comprises entirely, independent modules that could run in a completely unconstrained fashion, in an arbitrary order. Naturally, such systems are rare and it might be conjectured: more amenable to an entirely separate family of metrics and models. Therefore, one must have grave reservations concerning the validity of indirect flows[29]. These reservations are reinforced by the fact that indirect flows do not seem to correspond to any obvious "real world" design process or entity [Ince89b].

The equation for the metric is poorly formulated in that a single zero term is propagated through to result in an overall measure of zero. This is possible even if the module has, say, a large fan-out and comprises many LOC. It would seem that Henry and Kafura circumvented this problem in their original analysis by ignoring what they termed memoryless procedures on the basis that to do otherwise, "connections between procedures would be generated that do not functionally exist" [Henr79,p63]. In fact the problem that

[29] Barbara Kitchenham takes a similar view in her decision to omit such flows from her empirical investigation of design metrics [Kitc88].

Henry and Kafura are trying to avoid is the problem of module re-use. In other words, the separate instantiations of a re-used module should not be a vehicle for information flow transmission. Unfortunately, they seem to have adopted a solution leading to three problems. First, the fact that a module is memoryless does not ordinarily mean that there is no interest in its behaviour. Second, potentially complex components but, with either a zero fan-in or out, are *invariably* identified as having the minimum level of errors (presumably zero), or whatever quality is being analysed. Third, the serious problem of module re-use leads to specious information flows being counted. As the model stands it penalises the re-use of any module that exports or imports any information, due to its quadratic nature. Such an example is given in Figure 3.4 and Table 3.5.

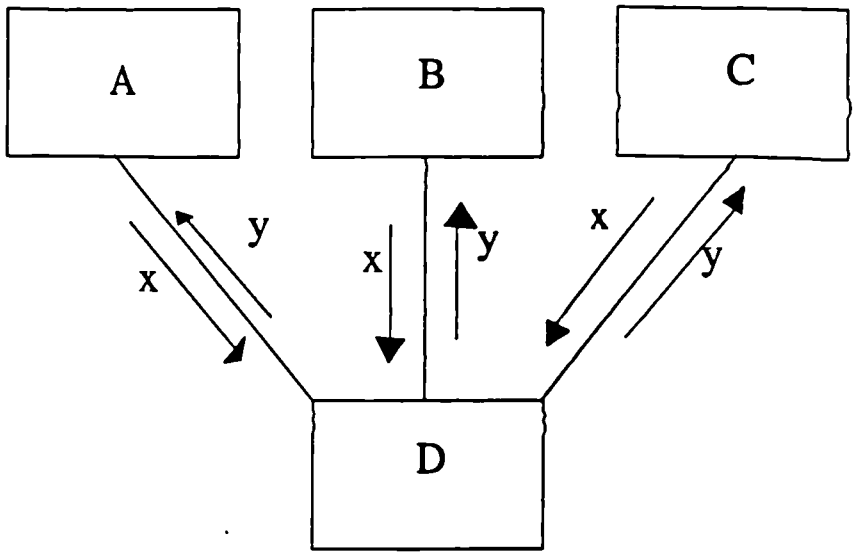


Figure 3.4: Information flows and module re-use

	With re-use			With duplication		
Module	Fan-in	Fan-out	Complexity	Fan-in	Fan-out	Complexity
A	1	1	1	1	1	1
B	1	1	1	1	1	1
C	1	1	1	1	1	1
D	3	3	81	1	1	1
E				1	1	1
F				1	1	1
Total			84			6

Table 3.5 Comparison of module re-use and duplication

Such problems are disconcerting in that they lead to the metric being difficult to apply and analyse [Ince88b] and must in part explain the extent to which empirical results differ, as clearly, module re-use levels will vary between different environments.

A more rigourous analysis of the metric equations reveals anomalies in the treatment of parameterised communication as compared with communication via global data structures. Despite defining global information flows (i.e. those via global data structures) Henry and Kafura fail to incorporate them into their definition of fan-in and fan-out. Instead they merely use a count of data structure accesses. This potentially has a considerable impact upon their measure as illustrated in Figure 3.5 and Table 3.6

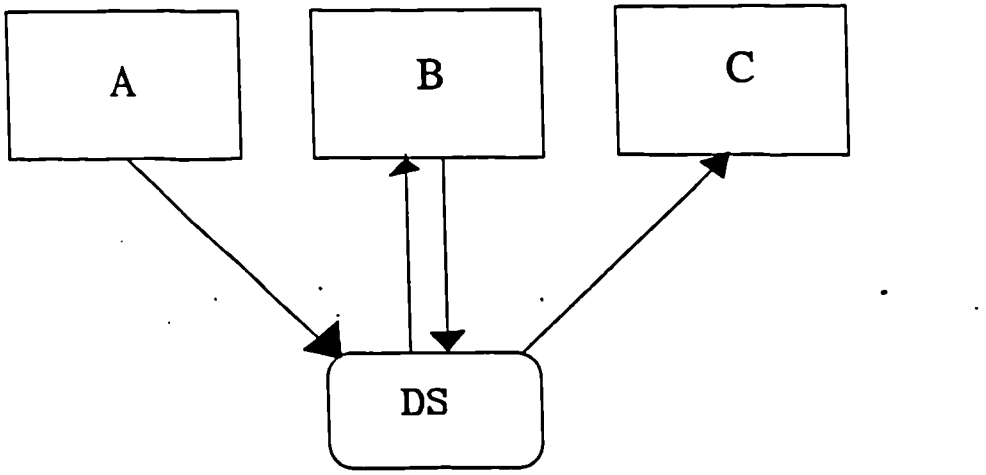


Figure 3.5: Information flow mechanisms

Module	Accesses counted Fan-in	Fan-out	Global flows counted Fan-in	Fan-out
A	0	1	0	2
B	1	1	1	1
C	1	0	2	0

Table 3.6 Comparison of the treatment of global flows

Counting reads and writes as opposed to global flows becomes significant when more than two modules communicate via a global data structure [Ince89b].

Another ground for criticism of information flow as a metric, is that the model makes extremely simplistic assumptions concerning the nature of the information. All flows are considered to have uniform complexity, but, the information might be a simple boolean or a complex structure containing many record variants. The metric is not sensitive to the difference. Thus, complex connections between modules could be disguised and not captured as information flows.

Moreover, as discussed with respect to the empirical evidence, the use of length as a measure of intra-modular complexity is debatable. Its late availability is also a problem if information flow is to be used as a true design metric. Henry and Kafura raise the possibility of refining this measure by replacing it with either Halstead's E measure or McCabe's cyclomatic complexity [Henr84]. However, given the problems that are inherent within both of these metrics it is doubtful whether this would represent much of an improvement. Further, their use would also delay availability of the metric.

The would-be-investigator is also confounded by the absence of definitions. In particular, global data structures and parameters remain undefined. Certainly, in many environments there would seem to be good grounds for treating devices as global data structures. Whether this should be extended to include screen output - and even keyboard input - are moot points. What is clear though, is the absolute paucity of Henry and Kafura's underlying model in these regards.

Kitchenham also criticises the metric as an example of a "synthetic" [Kitc88], in that combines disparate primitive counts, thus leading to potential confusion and difficulty in application. There is no doubt that it has confused and obscured notions of information

and procedural flow. It is arguable that these are best kept distinct, so as to facilitate the diagnosis of the underlying causes of any symptoms detected by the information flow metric.

A final, though lesser difficulty is the problem of obtaining the metric. Due to the difficulty of calculating the metric for a large system a software tool is the only practical possibility. Although the metric can be extracted from the code, a computer processable design notation would be preferable. Such notations are not currently in widespread use[30].

In conclusion, there are two main problems with Henry and Kafura's information flow metric. They have no clear idea what they are modelling and seem to commute between a variety of problem domains, almost as if they are interchangeable. Until goals for the metric are clearly stated it will be severely hampered in its application. The other problem is one of approach. In having adopted what appears to be a plausible idea, and one consistent with current thinking in software engineering, they have proceeded to obscure it under a facade of informality and arbitraryness. Many of the inconsistencies and anomalies contained within their model could have been avoided had a more formal approach been adopted.

3.5 Unfulfilled promises

What can we conclude from the dissection of the three most influential software metrics of the past 15 years? Is the fact that none of them appear satisfactory ill chance? If only the metrologists had chosen different numbers would all have been different? In the author's opinion to be so seriously in error three times is not mere bad luck but carelessness; it suggests that current approaches adopted towards software metrics are inappropriate.

It should be stressed, however, that it is not the individual factors being measured that are being criticised, but rather the approach to the application and interpretation of software measurement. First, our investigations have revealed a recurring pattern of ill conceived and poorly articulated models that underlie the metrics presented above. This in turn has led to models that are anomalous and out of step with current developments in software engineering. Despite the fundamental nature of these problems, they have all too often been obscured due to the lowly role ascribed the model. Directly stemming from modelling weaknesses are the problems of empirical validation. Empirical validation

[30] An alternative approach for data collection is described in [Shep89d] where the software tool creates a dialogue type interface between the designer and the tool.

of a software metric is quite difficult enough, without being uncertain as to what is being validated!

In short, the problems of software metrics are those of foundation and methodology. And this is the starting point for the research described in this thesis. It is evident from the survey of metrics research described in the last two chapters that the general tenor of the work has been a preoccupation with detail, arguably at the expense of higher level issues such as what does the metric mean, how might it be evaluated and finally how might it be integrated into the software engineering process? Each of these questions will be briefly reviewed.

First, what does a metric mean? The reader will have noticed copious references to models and underlying models, in the preceding chapters because it is only within the context of a model or a theory that a measurement has a meaning [Kybe84]. Consider the measurement observation that a piece of code has a luminosity of x. This cannot be interpreted because there is no theory or model to link the measurement with any other software engineering property of code - at least as far as the author is aware, apart from a slight suspicion that day-glo code might be injurious to one's eyesight! We cannot say whether a luminosity of x is good or bad. Nor can it be stated whether it will lead to problems of reliability and so on, and so forth. Evidently, this is an extreme example, but, equally it should be clear, that the metrics that have been reviewed have all relied upon implicit ideas, unspoken assumptions and partial definitions in terms of their underlying models. Consequently, the first item upon the research agenda is the development of a more formal framework for software metric models.

The second research agenda item is model evaluation. Since in many respects a model may be thought of as a theory, then it is natural that we should wish to evaluate the theory; this might be accomplished by means of empirical methods - for example through experimentation - or by more formal techniques, such as the application of axioms and proofs. It has been a major theme of this and the previous chapter, that there has been scant regard paid to model evaluation. To some extant this has been inevitable given the informal approach to modelling, but, it is also the consequence of *ad hoc* ideas and the absence of any systematic method for tackling the problem of model evaluation. The importance of validation cannot be over-stressed since metrics based upon flawed models are worse than valueless: they are potentially misleading. This research seeks to remedy this deficiency, by the development of a coherent infrastructure for the validation of software models.

The third and final area on our research agenda is the clear need for some method to guide software engineers in the selection and tailoring of software metrics to be suitable for their particular measurement goals. A major criticism of much past work has been the unfounded belief in "General Complexity Metrics" of form or another, which are suitable for close to all problems in almost any environment. This has led to the view that metric

selection is something akin to an "off the shelf" process. The search for some alternative approach yields two benefits. Firstly it raises the level of concern above the current obsession with metric *minutiae* that bedevils so much current work. And secondly, it forces more careful definition of measurement goals, with the attendant reduction in evaluation difficulties. If it is unclear what is being measured it is not easy to know if it is being done effectively!

These three issues then, form the research aims for this thesis and will be addressed in the subsequent chapters.

4. A THEORETICAL FRAMEWORK FOR DESIGN MEASUREMENT

"As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality."

Albert Einstein

"The first thing that we should observe is that theory and measurement are more intimately related than is often thought ... The lesson to be learned from this, particularly in the mental and social sciences, is that, however cleverly we measure something, however reliable the test or reproducible the measurement, without a theoretical framework into which that quantity enters, it is useless."

Henry Kyberg [Kybe84]

Synopsis of chapter

Now we commence addressing the research agenda outlined at the end of the previous chapter, that is the need for a method to articulate models behind software metrics, a mechanism to evaluate models more rigourously and lastly a means whereby metrics appropriate to a given software engineering problem or opportunity may be tailored and selected. First, however, the chapter examines the various types of measurement, direct and indirect, and the different measurement scales that are identified in the theoretical metrology literature. The necessary axioms for each type of scale are described in conjunction with the bearing this has upon software metrics. The relationship between metric and model is then explored. It is suggested that a measurement without contextual model is meaningless. The following components of a well formed model are identified: input variables, output variables, parameters, relationships between the above, mappings between the model variables and the "real world", assumptions, accuracy.

The lack of such a framework coupled with endemic informality and intuition are the prime contributory factors to the difficulties that are being encountered with the majority of current metrics, as described in the previous chapter.

The topic of metric evaluation is addressed and divided into theoretical, or axiomatic validation and empirical validation. Specific axiomatisations for software measurement due to Prather [Prat84], Fenton [Fent86] and Weyuker [Weyu88] are reviewed. Prather's and Fenton's approaches are found to be insufficiently discriminative, whilst Weyuker's axioms are too restrictive. A new flexible approach is proposed. A list of desirable and mandatory characteristics for convincing empirical validations is presented. It is observed that few empirical validations fulfill these criteria, partly as a consequence to the poor regard paid to modelling by the majority of software metrologists.

In the light of this theoretical background, a methodology for the development and evaluation of software models and their related metrics is advanced. This is contrasted with, and in many aspects found to be complementary to the Goal/Question/Metric (GQM) paradigm of Basili *et al* [BasR88]. A simple example is presented based upon the development methodology.

4.1 Introduction

The preceding analysis of developments in software metrics suggests that measurement is understood and applied on a very informal basis. In particular, the previous chapter demonstrates that software metrologists have a very weak notion of modelling, which as a consequence, has profound implications upon progress within the whole subject area. The justification for the use of metrics as an adjunct to software development has tended to run along the following lines.

> *"You can't control what you can't measure. In most disciplines, the strong linkage between measurement and control is taken for granted."*
>
> Tom deMarco [deMa82]

Other engineering disciplines are able to take and utilise measurements in order to provide quality control and even more alluringly, accurate predictions. Software engineering does not compare favourably with the construction of other artifacts. *Ergo* we should make greater use of software measurements, which will result in cheaper, more reliable and useful software products.

Thus far it would be difficult to disagree, but the corollary that many infer from this position could characterised as being representative of the "Kelvin syndrome",

whereby the possession of almost any measurement renders the software engineer better off. That the measurement, even if it is not directly useful, will cause the software engineer to regard the software product in a more analytical fashion and by some mysterious inductive process, yield great insights. Such an analysis fails to understand the type and use of measurements in other more mature disciplines. It is perhaps instructive to examine the measurement of temperature which is a well understood and tolerably reliable process, prior to the development of a more explicit technique for presenting the models behind metrics, a method for the more rigourous evaluation of these models, and a means for selecting metrics tailored to the measurement goals and environments.

One of the interesting features about temperature measurement is that it is not, in general, measured directly. The most common approach is to measure length; length of a column of mercury contained in a thermometer. This is an example of indirect measurement. It is not that temperature cannot be measured directly [Kybe84] but that it is more convenient and accurate to do so indirectly.

How is it that we can satisfy ourselves that there exists a reliable relationship between the length of a column of mercury and temperature? It is possible that we might start with the supposition of some theoretical relationship (e.g. a linear function) between the indirect measurement and actual temperature. Empirical observation and experimentation is then required to confirm or refute the relationship. Alternatively, our starting point may be drawn from the observation that mercury expands with increases in temperature and this may then be formalised into a set of equations.

Careful observation of our thermometer in a wide range of environments reveals that what appeared to be a simple linear relationship between length of mercury and temperature is in fact rather more complex. For instance, it is particularly evident that the thermometer fails to perform accurately on the top of Mount Everest. Also mercury thermometers are restricted by the fact that mercury will vaporise at high temperatures and are therefore cannot measure temperatures much over $350^{o}C$. Thus the measurement of temperature, by thermometer, is dependent upon a number of factors and subject to certain limitations.

In practice one can compensate for temperature readings at different altitudes since we can make the model more sophisticated as we begin to better understand the problem domain. So, we model the way in which temperature, altitude and pressure are related in a systematic way. Thus, more than one measurement is required. The perception of temperature as part of a system is an example of a major development in our ability to measure accurately.

The final point to draw from this brief peroration into thermometry is the problem of error. The quantitative/qualitative or objective/subjective dichotomy is, in at least once sense, misleading because in practice one is confronted by a continuum. Some measurement practices, such as thermometer measurement of temperature, are more reliable than others, for example measures of aesthetics. Reliable is meant in the sense of a low frequency of errors, small error size and a normally distributed set of errors (i.e. the absence of systematic errors or bias).

Let us now return to software engineering. The remainder of this chapter examines in more depth the issues raised by the forgoing discussion, namely: the different types of measurement and their characteristics; the conditions that must hold true for each type of measurement; the relationship between model and metric and the issue of measurement reliability. From this are developed some requirements for an ideal metric and a methodology is proposed for the development and evaluation of metrics appropriate for a particular measurement objective and environment. A simple example of a module based metric is presented which illustrates many of the difficulties that exist with software metrics and some of the benefits of a more rigourous approach.

4.2 The Theory of Measurement

At this rather belated stage it is appropriate to consider more exactly what is meant by measurement. Using Pfanzagl's definition [Pfan68], measurement is taken to be the process of assigning numbers[1] to describe some empirical attribute of a product or event *by rule*. The rules that govern the number assignment are fundamental since they introduce varying degrees of objectivity into the process. Nonetheless the boundary between measurement and non-measurement is a fuzzy one.

In this section we briefly review classical measurement theory and discuss the extent to which it does, and ought to, impinge upon current practice in the software metrics field.

4.2.1 Types of measurement.

As the example of temperature has revealed, measurement may be one of two types, direct or indirect. Indirect measurement occurs whenever one, or more, other quantities

[1]On occasions one might assign other mathematical entities instead of numbers [Kran71], for instance vectors as in the case of the Myers' [Myer77] extension to the cyclomatic number metric. To do so, is however, rarely judicious as this leads to deep methodological problems, given the absence of any generally plausible greater-than or less-than relations and as a sequitur little theoretical basis for anything other than a nominal scale. Another example of non-numeric measurement, is the case of nominal scales where the numbers merely serve as labels and can therefore be legitimately substituted by letters or any other unique identifier.

are measured in order to provide a measure of the actual object or event of interest. Direct measurement occurs if no other measurements are required. Within the domain of software metrics, indirect measurements are the most commonplace. Indirect measurement is usually employed for the reason that temporal considerations prevent direct measurement because the measurement is being used in a predictive capacity. This has an important implication, in that theory is required to link the indirect measure, or measures to the object of the measurement process. For instance, it might be considered desirable to measure implementation effort indirectly so as to be able to obtain it in advance (i.e. predict). If implementation effort were measured directly it would not be available until after the implementation has been completed. In order to measure indirectly it might be hypothesized that there is a relation between the total number of data structure references and implementation time. So, we have a basis for measuring indirectly by counting data structure references. A theory is being developed that posits a relationship between the two measures and therefore must be incorporated into any underlying model. The relationship between measure and model is re-examined in a subsequent section, but, in passing we note the obligation to validate any model particularly to justify the use of indirect measurement.

4.2.2 Scales for measurement

Classical measurement literature identifies the following four types of measurement scale[2].

Nominal scale, for example the numbering of football players.
Ordinal scale, for example the hardness of minerals.
Interval scale[3], for example temperature in degrees Centigrade.
Ratio scale, for example temperature on the Kelvin scale or length.

Naturally each scale has different properties or empirical operations associated with it, and is required to satisfy different axioms. These are given in Table 4.1 adapted from Stevens [Stev59].

[2]To be more exact, there are a non-denumerable infinity of scales and include such exotica as the logarithmic interval scale [Stev59] and the hyperordinal scale [Supp71], however these have no, or marginal empirical application!

[3]Strictly speaking we mean a linear interval scale and unless stated to the contrary all references to the interval scale should be read as such.

Scale	Basic Empirical Operations	Mathematical Group Structure
Nominal	=	permutation group - M'=f(M)
Ordinal	=,<,>	isotonic group - M'=f(M) where f(M) is any monotonic increasing function
Interval	=,<,>,equality of intervals	general linear group - M'=αM+β , α>0
Ratio	=,<,>,equality of intervals and ratios	similarity group - M'= αM , α>0

Table 4.1: Properties of measurement scales

Let us review each scale in turn. The *nominal scale* is the least restricted, and therefore the simplest of all the measurement scales. The only empirical operation that is required to enable this form of measurement is the determination of equality (by which we mean empirical indistinguishability). This binary equivalence relation is reflexive, symmetric and transitive. Examples of nominal measurement are the oft cited case of football player numbers and, within the area of software engineering, the classification of software systems by the basic COCOMO model [Boeh81] as organic, semi-detached or embedded. It is noteworthy that in the latter case no use is made of numerals. However, the assignment process is still carried out according to rules, even if these are only informally stated, hence it may be considered a process of measurement.

The *ordinal scale* introduces rank ordering, derived from the empirical weak ordering relations, less-than and greater-than. These relations are reflexive and transitive. Again, COCOMO can be used to provide examples, this time from the intermediate model, where cost drivers (e.g. DATA, the size of the database) are placed on a scale ranging from very-low to extra-high. Although, we may empirically determine that a very-low sized database is less than an extra-high database, it is not possible to assert anything concerning the size of the difference. Whilst this is fairly self evident for the above example, were we to substitute numerals for the names of the classes (e.g. rate database size from 1 to 5) it might be more tempting to succumb to such a temptation. For

similar reasons, one must exercise caution concerning the application of statistics to ordinal, or for that matter nominal scales [Stev46, Stev59].

The possibility of empirically comparing measurement intervals introduces the *interval scale*, a scale that is quantitative in the normal meaning of the word, excepting a defined absolute zero. The most commonplace example is temperature measurement using degrees Centigrade. Since, zero degrees is arbitrarily placed, it is not possible to state that $10^{o}C$ is twice as hot as $5^{o}C$, although it is empirically possible to determine that the difference between $10^{o}C$ and $5^{o}C$ is the same as the difference between $20^{o}C$ and $15^{o}C$.

Ratio scales differ from interval scales in that absolute zero is always implied, as in temperature measurement by the Kelvin scale. Consequently it is possible to determine equality of ratios in addition to equality of intervals. Software metrics offer many such examples, ranging from LOC to Halstead's E metric.

In many cases it is obvious from the empirical operations available to us, as to what scale we are dealing with. More formal mechanisms for the determination scale type do, however, exist. These are based upon transformations that are possible upon the measurement group structure[4] which still preserve empirical orderings [Stev46, Stev59, Supp71, Fink84]. Nominal scales may undergo transformation by any one to one function, due to the absence of even weak ordering. By contrast, the only permissable class of transformation for a ratio scale is of the form $M = \alpha M$ for non-negative values of α. Table 4.1 lists all permissible transformations for each scale type.

The final point to make whilst examining scale types, is the extent to which ratio measurement is in some sense "better" than nominal or ordinal measurement. Adams [Adam66] argues that by merely viewing measurement in terms of the axioms or conditions necessary for a certain scales of measurement to be possible, one can lose sight of purpose. He gives the example of mineral measurement using Mohs hardness scale (ordinal) which is wholly adequate for the purpose of mineral identification in the field; not an unfortunate necessity imposed due to the difficulty of establishing the conditions necessary for interval or ratio measurement.

What significance does the foregoing discussion have for software metrology? The answer is threefold. First, the type of scale may restrict meaningful statistical manipulations and observations. A common problem is the use of statistical means on ordinal type measurement, but to do so requires the assumption of constant intervals

[4]By structure we mean the relational structure <N,R> where N is the set of all numbers (or observations for an empirical structure) and R is the set of all relations. For an ordinal scale the measurement structure is therefore $< N,\geq >$, for a ratio scale it would be $< N,\geq,+ >$.

between all the points on the scale. Should such a situation prevail the scale would be interval or possibly a ratio.

The Henry and Kafura information flow metric [Henr81a] provides an interesting application of empirical meaningfulness being restricted by scale. The metric is an example of an indirect measure of software complexity (whatever that might be!) based upon counts of information flows between modules. As a direct measure (i.e. treated merely as simple counts), the information flows must be placed on a ratio scale[5]. However, given the formulation of their model, or that part that relates the indirect measure to the quantity of actual interest, in this case software complexity is:

$$(\text{fan\_in} * \text{fan\_out})^2$$

where fan_in and fan_out are modified information flow counts, the quadratic nature causes it to behave as a monotonically increasing[6] function. This transformation is weak order preserving, but no more. As a consequence the only meaningful remarks that may be made concerning this metric are those that are based upon the relations of equivalence and weak ordering. For example it would not be admissable to state that module x with a value of 100 is twice as complex as module y with a value of 50. Observations of interval size or ratio have no empirical meaning. Though in general meaningless statements should be avoided, occasionally pragmatic considerations may overrule. Finkelstein and Leaning [Fink84] give the example of using the arithmetic mean, rather than the median, of a set of examination marks despite being measured in an ordinal fashion.

Second, the distinction between direct and indirect measurement is an important one. Indirect measurement, whilst being widely used within the field of software metrics, places an additional burden upon the researcher - *viz* the need to justify the theory linking the indirect to the direct measure.

Third, the identification of scale type introduces certain axioms that must be satisfied by a theoretical analysis of the metric and its underlying model. In brief, it is necessary to show that two theorems hold for the measurement; the Representation Theorem and the Uniqueness Theorem [Supp71, Kran71].

The Representation Theorem requires that the numbers assigned by the measurement process must properly represent observed empirical relations. In order for this to be the case there must be a homomorphic (or isomorphic) mapping from the empirical to the

---

[5]This is for the reason that the only transformation that does not lead to loss of ordering information is of the form $M = \alpha M$.

[6]The function is monotonically *increasing* because the information flow counts and the fan-in and fan-out must be non-negative.

chosen number system[7]. There are two aspects to this homomorphism, first the measuring function that maps empirical observations onto numbers and second the mapping between empirical and number system relations (e.g. the empirical relation heavier-than may have a corresponding numeric relation >). Unfortunately, this theorem is less useful in practice than one might suppose, since for any finite or denumerable empirical system there can always be found some numeric system that is isomorphic to it [Supp71], though this may be by virtue of employing "unnatural or pathological" relations. Worse still, there may be little agreement upon the empirical relations, because, for instance, we may not all agree that program x is more complex than y. Consequently, regard to this theorem, is only a minor aspect of metric evaluation. By itself, it is insufficient. The effort of formal proof is therefore seldom justified, rather informal reasoning is adequate to check for "gross" errors[8].

The discussion of scales has already touched upon aspects of the Uniqueness Theorem. Measurement is unique up to certain levels of transformation, so for example non-Kelvin temperature measurement may multiplied by, or added to, a constant (or both), without loss of uniqueness for the type of scale adopted. Satisfaction of this theorem therefore, is concerned with the formal proof that all numerical relations are equivalent to all empirical relations for all permissible mappings from the empirical system onto the numerical or measurement system. This is a formalisation of our earlier discussion as to suitable scales.

Although observance of these theorems is a necessary step in the theoretical validation of a metric - a point that will be taken up again, later in this chapter - they are not a panacea for software metrics. Fundamental difficulties remain. These are, errors in the modelling and the measuring processes. In any case, as Zuse and Bollmann [Zuse89] observe, we do not have an agreed view on the empirical relational system and therefore are not in a position to comment upon the mappings between it and our chosen measurement system with the certainty that discharge of the above proof obligations would require. The corollorary is that there do not exist generally accepted scales for software complexity metrics.

---

[7]As we have already noted measurement need not be limited to the assignment of numbers. In such circumstances the establishment of a homomorphism can be an involved process, although extremely important since the properties of the measuring system will be less well understood than the familiar number system.

[8]This conclusion would be strongly disputed by other researchers such as Kaposi and Myers [Kapo90]! However, the problem in the end reduces down to one's views of the empirical world and the need to find mappings between this world and the closed system of classical measurement theory.

Clearly there is no simple redress for erroneous modelling of software[9], because it is the outcome of our current lack of understanding about software engineering processes and their interaction with products. However, the theory described thus far is idealistic, in that it incorporates no notion of uncertainty in obtaining measurements, the empirical relations. Two possible solutions are, the application of a statistical theory of errors coupled with a linguistic framework to describe reports (observations) and statements from Kyberg [Kybe84], and an extension of the formal machinery in order to axiomatise uncertainty, for example semi-orders described by Luce [Luce56]. Kyberg's approach is more promising in that it attempts to use statistical means to establish a link from the uncertain empirical world into an ideal world of absolute certainty. Luce provides no such mechanism. In a sense, formality and closure imply an inevitable retreat from the "real world".

The conclusion must be, that software metrologists ought to be aware of classical measurement theory and the restrictions that it imposes for meaningful measurement. Yet it should be equally clear that the majority of deep problems remain unaddressed, the chief of these being errors in modelling and measurement.

4.3 Modelling and Measurement

4.3.1 Measurement and purpose

One of the fundamental factors to consider within metrology is the relationship between measurement and model. Although a model *is not an imperative for* measurement, in any absolute sense, the model establishes the context and the meaning of the measurement. A metric such as the number of source lines of code *(SLOC) has* little meaning and defies evaluation in the absence of a model. Even a rudimentary and informal model is an important starting point. For example, the model might suggest that programmer productivity is some function of SLOC, normalised for time and number of staff. Another model could employ SLOC as a predictor of stationery requirements. We may well have misgivings concerning the first model; possibly fewer concerning the second, but at least contexts for the measurement and its evaluation are now provided.

Having introduced the idea that measurement requires a model to permit meaningful application and evaluation, let us scrutinise the concept a little more carefully. A model is an abstraction or simplification of reality so the first concern is that the model must meet a purpose. Without a purpose we cannot know what aspects of the real world

[9]Obviously there is a blurred distinction between model and measurement error, to the extent that the former will lead to the latter. Whether this is amenable to statistical remedy depends largely upon the size of the scatter, and the nature of the distribution.

to exclude and which to incorporate. Modelling in a vacuum is a rather curious endeavour. It is, however, a potential pitfall for academics who may not be immediately confronted by a problem and thus software metrics can degenerate into rather an aimless activity. *Vide* Bache and Tinker [Bach88] for an example of an mathematically rigorous model, but one which unfortunately happens to be almost orthogonal to the objectives of the research which are variously stated as measuring "cognitive complexity", "bugs", "number of changes". And whilst one would wish to applaud their desire to introduce a little more formality into the world of software metrics, one is drawn inescapably to the conclusion that the model, rather than the problem has become pre-eminent in the minds of the researchers. Similar charges could be levelled at the work of Fenton *et al* [Fent86, Fent87a, Fent87b] where the problem domain appears to be of little importance relative to the development of a formal model.

A model, preferably of the problem domain, will establish relationships between various entities. In the above example of paper consumption we postulate a relationship between SLOC and the quantity of paper required. It is often helpful to express these relationships more precisely as equations. In doing so this encourages rigorous thinking and a better understanding of the problem domain. It also facilitates validation. Developing the stationery example, we might more formally specify the model as:

$$\text{consumption} = \approx (\text{SLOC})$$

Immediately this raises issues of how we relate our abstract model to the real world. What is consumption of paper? What units do we propose to use? Is there a time dimension to consumption? To keep matters simple let us suppose the answer to the last question is no. As for units, the page seems a reasonable candidate. This now gives some clues for the function f which will the reciprocal of the number of lines per page giving:

$$\text{consumption} = 1 / n(\text{SLOC})$$

where n is the number of lines that may be printed per page.

To generalise, a model will have inputs and outputs, sometimes known as endogenous and exogenous variables. These are linked by relationships, or a set of equalities, which may use parameters, as in the example above where n allows the model to be used for a variety of printers and stationery sizes. This is illustrated below.

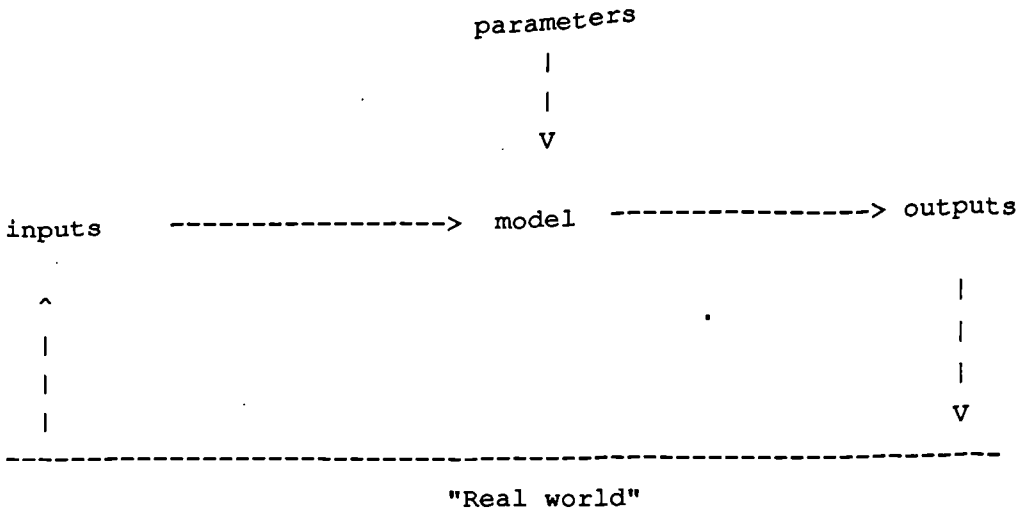


Figure 4.1 The relationship between a model and "reality"

A model should also define the mapping from the real world onto the inputs and outputs identified. This establishes the essential two-way link between abstract space and reality. There is the measurement process, (effectively a mapping from the real world onto the model) and the prediction process (a mapping from the model back onto the real world. When no such connections exist the model is metaphysical.

It is immediately clear that this framework is not made explicit in any of the metrics previously discussed. Difficulties with the unspecified or ambiguous mappings have manifested themselves as counting problems and as the prediction of metaphysical properties. Perhaps the latter is the most disturbing issue in that metrics being used to predict metaphysical properties, most frequently complexity. When we are faced with definitions, even if intended to be ironical, such as "a not-so-warm feeling in the tummy" [Curt79b] this is must be treated as a warning sign, unless of course proponents of complexity metrics intend to measure, or even predict the drop in stomach temperature of a software engineer when confronted with the offending software!

Developing the notion of the metaphysical further, I will make a rather contentious proposition.

> *Not all software properties are measurable or even directly observable, in any useful engineering sense.*

Recall that measurement implies the existence of rules for the assignment of numbers, labels, vectors or whatever. Where the rules are unknown or non-existent I do not

consider this to be measurement and the property is metaphysical. For example, "user-friendliness" cannot be measured as such, unless the rules for this task are defined. This does not preclude informal rule definition and it may also be necessary to accommodate error in the measurement process. In practice, we attempt to measure some of the possible consequences of "user-friendliness", or the lack of it (i.e. employ an indirect measuring technique). We may choose to measure the time taken by a user to perform a particular task and hypothesize that the results have a particular significance; that we have indirectly measured "user-friendliness" in a manner analogous to the measurement of temperature by the vertical height of a column of mercury. However, I feel this analogy to be somewhat specious, as "user-friendliness" is sufficiently nebulous as to make operational definition difficult. If it cannot be defined then the rules for assigning numbers cannot be defined and no intelligible measurement is possible. Nor can it be claimed to be measured indirectly, as there is no way of establishing the validity of the hypothesis linking the indirect measurement to "user-friendliness" because of the problems of establishing an empirical relationship between the indirect and our undefinable measure[10]. Instead we must conclude that "user-friendliness" has something of a metaphysical flavour about it. Metaphysical properties can be of immense importance and interest, however, to set out to measure something that is fundamentally immeasurable is would seem to be an uneccessarily fraught enterprise, if not wilfully perverse.

It is singularly ironic that computer scientists should choose software complexity as their chief target for software metrics research. The consequence of claiming to measure the metaphysical has been a plethora of supposition, intuition and instinct masquerading as general models of software and software construction. Without doubt something is being measured. But what, and what it means, is quite unclear.

To summarise, measurements must be made in the context of a model in order to have meaning and to admit validation. Models must address some problem or purpose. To be useful, it is necessary to be able to relate the model to the "real world" and this cannot

---

[10]This argument could be construed as a rather intoxicating concoction of Popperian empirical refutation and Bridgman's doctrine of operationalism. Both ingredients are potentially troublesome. Empirical refutation taken to its logical conclusion argues that only the metaphysical may be regarded as irrefutable, a position that could be somewhat inconvenient for the software metrologist as it may not be desirable to call into question the fundamental axioms of measurement, derived from the classical theory on every occasion that empirical results do not satisfy our predictions. Imre Lakatos [Laka70] offers one way out, suggesting that within the framework of a research programme certain laws and assertions may be treated as axiomatic. Operationalism suggests that the measure defines what is being measured. The danger that lurks here is that we may lose sight of how adequately a measure captures the property that is of interest, since measurement must satisfy the tautology that it measures what it measures! On the other hand it is difficult to see how we may have any meaningful notion of measurement quality if we are unable to define what is of interest (i.e. it is metaphysical).

be accomplished if it contains metaphysical entities, consequently operational definitions are required for all the endogenous and exogenous model variables. Unfortunately, this framework is rarely applied to software metrics, and in particular, we note the predominance of implicitly, informally and inadequately defined models coupled with frequent reference to metaphysical variables such as complexity.

4.3.2 Models and theories

By now it should be clear that a model embodies a theory. This is expressed in the relationship between the inputs and the outputs of the model, and is the basis for useful models, enabling them to have predictive power. The most common form of theory, within software metrics, is that which is required to link direct with indirect measurement.

Referring back to our original illustration of the temperature measurement, it will be recalled that mercury thermometers are affected by altitude, or atmospheric pressure considerations. In order to compensate for this problem additional measurements are required. It is a general principle that as models are generalised across larger domains, so they require increasing numbers of inputs and grow in sophistication. Given our limited understanding of the software development process, it is an ambitious objective to continue to propose the type of all-encompassing, all-purpose, yet remarkably simplistic, models that have abounded in the literature (e.g. [Hals77, McCa76, Henr81a, Card88], without consideration of the limitations and applicability of the model. By way of contrast, some of the cost estimation models, such as Barry Boehm's COCOMO [Boeh81] are relatively complex because they explicitly take into account the wide range of environments to which they may be directed. The detailed COCOMO model has more than 50 inputs and parameters or cost drivers as Boehm terms them.

This suggests an additional component to any useful model - that of its limitations. These are best expressed in terms of the assumptions that are made, thereby allowing the would-be-user to determine its applicability to a particular problem domain. Where assumptions are deeply embedded, or merely implicit, this leads to difficulties in the validation and application of the model. Nowhere is this better illustrated than by some of the empirical work based on the Software Science model [Hals77]. It is clear that this model assumes a small scale programming[11] environment and a FORTRAN like language. Arising from this assumption, is the fact that the meaning of "effort" is self evident - it is the time, usually no more than a few minutes to develop

[11] Indeed Halstead initially was only concerned with algorithms as indicated by [Hals72].

the software[12]. Subsequent work, for instance by Basili and Phillips [BasP81] of very larger scale, team based, software development highlights the ambiguities lurking behind the term "effort". Should requirements analysis be included, or for that matter changes in requirements during the development? Is time expended upon communication between team members to be included? How are library routines handled? What about fixing faults once the system has been released? It is thus not surprising that Basili reports a much weaker relationship between predicted and actual "effort" than that found by earlier studies of very small scale programs (e.g. [Gord76]).

The majority of criticisms of the Software Science model, outlined by the previous chapter are a direct consequence of violations of many implicit assumptions that Halstead made. Within a much smaller domain, as originally intended, of small algorithms, FORTRAN and single programmer environments, the Software Science model *might* have had more chance of succes, although other theoretical difficulties remain [Coul83].

There is one outstanding ingredient, for at least the ideal model, and that is reliability. How much confidence can one have in the model and the measuring process that it supports? This may be expressed in a variety of ways, the most suitable being determined by the nature of the problem. Alternatives include identification of the worst case and more commonly the probability of obtaining a certain degree of accuracy as in the COCOMO model where it is suggested that there is a 70% probability of the model being accurate to within plus or minus 20% [Boeh81]. Accuracy of design metrics are also on occasions given [Kitc88, Shep88c]. Although confidence in our confidence is an intriguing philosophical (and recursive) problem it does not justify the widespread omission of this constituent of a model by software metrologists. Obviously, much empirical work is required to generate this data, but it is of great practical value, especially in a field such as software engineering where uncertainty is the order of the day.

Despite the widespread reference to software modelling within the software metrics literature, there has been little discussion of the structure of a well formed model. I have argued that a model comprises: input and output variables, mappings between the "real world" and the model thereby allowing "real world" use, relationships linking inputs to outputs, assumptions that the are made and thus delineation of the domains to be modelled and some indication of accuracy of model outputs (i.e. the extent to which the predicted value deviates from the actual value). In addition a model may contain

[12]Halstead and his colleagues were typically working with programs ranging from 7 to 59 lines of Fortran [Gord76], and averaging 24 LOC.

arameters in order to enhance generalisation capabilities. It is rare that all these components be even implicitly defined within a software model.

4.4 Model Evaluation

Having scrutinised some of the deficiencies of existing software metrics and their underlying models, it is now appropriate to consider what are the desirable features of a metric. Although it is usual to speak of metric evaluation, strictly it is the model, into which the metric enters, that is validated. As has already been shown, the model establishes the meaning of a metric. Without a model no evaluation is possible. It has also been demonstrated that a well formed model has seven components, some of which are frequently omitted or are at the very least deeply disguised. Evaluation is unquestionably more difficult, of partial or poorly articulated models.

There are two complementary approaches to model evaluation. The model may be analysed on a theoretical basis. In particular, one might search for the following characteristics:

1. The model must conform to widely accepted theories of software development and cognitive science. Admittedly this is a rather subjective criterion. However, consider the following example. A metric which predicts that monolithic piece of software will have a lower incidence of errors than one divided into a number of modules must be viewed with suspicion. In such circumstances the onus would certainly be on the proponent to demonstrate the adequacy of the underlying theory.

2. The model must be formal as possible. In other words, the relationship between the input measurements and the output predictions must be precise in all situations. Further, the mapping from the real world onto the model must be made as formal as possible.

3. The model must use measurable inputs rather than estimates or subjective judgments. Failure to do so leads to inconsistencies between different users of the metric and potentially anomalous results[13]. Automation is not possible without satisfaction of this criterion.

These rather general model attributes can be refined in order to provide a good deal more precision by means of the axiomatic approach. Existing work in this area is reviewed and a new, more powerful approach described.

[13]This seems to be a potential shortcoming of function points, vide Low and Jeffery [Low90a].

A model should also be subjected to empirical evaluation. A model may have all the attributes listed above, might satisfy a large set of axioms, and yet completely fail to describe the "real world" that it purports to capture. Empirical validations are an equally necessary and complementary validation technique. In order that these may be meaningful, they also require certain attributes. In brief the desiderata[14] are: large scale empirical validations, in a variety of different environments, particularly industrial ones, adequate controls so that it is possible for a null hypothesis to stand, and different teams of workers. These characteristics are described in more detail later in this chapter.

The ordering of model evaluations is intentional, since meaningful empirical work is of questionable significance when based upon meaningless models of software. Therefore, theoretical analysis of the properties of a model ought to precede empirical validation. Furthermore, theoretical evaluation is often much quicker, and is consequently a cheaper and easier method of exposing some of the potential weaknesses in a model than a full blown empirical study.

4.4.1 Theoretical criteria

(a) Prather's axioms

In an early attempt to provide some unifying framework for the evaluation of the extraordinarily divergent collection of software metrics, Prather [Prat84] proposed a set of axioms which a "proper complexity metric" must satisfy. These are:

Axiom 1: The complexity of the whole must not be less than the sum of the complexities of the parts.

Axiom 2: The complexity of a selection must be greater than the sum of all the branches (i.e. the selection predicate must contribute complexity).

Axiom 3: The complexity of an iteration must be greater than the iterated part (for the same reason as Axiom 2).

Although this is an interesting idea, a number of problems remain. First, the axioms are restricted to structured programs, since they do not address non-structured forms of control flow such as branching out of the middle of iterations. Second, the axioms are clearly aimed at a specific family of metrics, *viz* those that have underlying models based upon program control flow; for instance cyclomatic complexity [McCa76] and the

[14]Of course in an ideal world one would give full regard to all these factors. Unfortunately in the world of limited resources that we happen to inhabit compromises must often be made.

knot metric [Wood79]. As a consequence they lack general applicability. Third, the axioms provide very little constraint upon the imaginations of the inventors of software metrics. It is true that Prather suggested upper bounds for Axioms 2 and 3 of twice the lower bound, but these seem to be more conjectures than axioms. No justification is offered to support these values.

Prather's approach can be characterised as a set of weak axioms for the family of control flow based metrics. Nowhere is this better demonstrated than by the impasse that he finds himself when having explicitly shown weaknesses in McCabe's metric he is unable to evince any violation of his axiom set. Nevertheless, it represents an important new approach which has been subsequently further developed by others. It is important in that it provides a foundation for the comparison of essentially similar metrics. Also, it allows for the proscription of unacceptable model behaviours in a rather more formal fashion than has been current practice.

The first development of this work was by Fenton *et al* [Fent86] in order to extend the axiomatisation to any procedural program, structured or otherwise. Program flow graphs are reduced to a hierarchy of irreducible graphs (prime trees) which permit any program structure to be uniquely defined. The axioms were redefined in terms of prime trees and thus made applicable to any procedural software. Subsequently, Prather [Prat87] further developed this work, noting a distinction between hierarchically and recursively defined metrics and marginally extending the scope of axioms defined over prime trees [Whit85].

Clearly these are significant contributions to the problem of metric evaluation. The use of formalisms such as graph theoretic approaches for modelling and metric definition facilitate reasoning about, and making comparisons between, models. Unfortunately the substance of the problem remains; the axiom sets are weak and the modelling incomplete. This is because the mappings from model to "real world" are generally un- or ill-defined. The latter is the price paid for formality, a retreat into metaphysics. There is also an additional problem concerning closure of the concatenation operation which will be returned to, in due course.

(b) Weyuker's axiomatisation

A contrasting perspective is due to Weyuker [Weyu88] who presents a set of nine axioms or properties that a well formed "complexity measure" and its underlying model should satisfy. These axioms might be regarded as strong, in the sense that they impose considerable restrictions upon the scope of permissible metrics. This is borne out by the observation that none of the metrics evaluated by Weyuker satisfy more than seven out

of nine axioms. For programs p, q and r[15], where | | denotes a hypothetical measuring function yielding a non-negative number, the axioms are:

Axiom 1: The measure must not assign the same number to all programs (i.e. it must discriminate).

$$\exists\, p,q{:}\text{program} \cdot |p| <> |q|$$

Axiom 2: There exist only a finite number of programs for a given measurement value. To judge this one needs to make certain assumptions concerning the programming language and the target machine. The stated purpose of this axiom is to "strengthen" Axiom 1 as violation suggests that the measure is comparatively insensitive.

Axiom 3: There are programs drawn from the same equivalence class (i.e. the measure is not too sensitive).

$$\exists\, p,q{:}\text{program} \cdot |p| = |q|$$

Axiom 4: There must exist programs that compute the same function but have different numbers attached to them (as a consequence of internal or syntactic differences). As Weyuker notes, for all practical purposes Axioms 1 and 4 are equivalent.

Axiom 5: The measure must be monotonic.

$$\forall\, p,q{:}\text{program} \cdot |p| \leq (p \circ q) \wedge |q| \leq (p \circ q)$$

This axiom centres around the meaning ascribed to the concatenation operation for the object or process being measured. For many software metrics it is not self evident how to define this operation, a point that will be returned to later.

Axiom 6: Concatenation of a program p to another program must not always yield a constant increment to the total program measure (i.e. the impact of the concatenation depends upon the program to which p is being added).

$$\exists\, p,q,r{:}\text{program} \cdot (|p| = |q|) \wedge (|(p \circ r)| <> |(q \circ r)|)$$

Also:

$$\exists\, p,q,r{:}\text{program} \cdot (|p| = |q|) \wedge (|(r \circ p)| <> |(r \circ q)|)$$

[15]Whilst Weyuker talks in terms of programs, plainly, one could substitute any measurement object.

This not a universal desideratum of software measurement; consider the measurement of LOC! The issue depends upon the choice of measurement scale type, as described earlier in this chapter.

Axiom 7: The measure must be sensitive to the ordering of the program components. If q is some permutation of p then:

$$\exists\, p,q:\text{program} \cdot |p| <> |q|$$

Our comments concerning Axiom 6 are also applicable here.

Axiom 8: The measure must be insensitive to renaming changes of program components. Thus, if p is a renaming of q, then

$$\forall\, p,q:\text{program} \cdot |p| = |q|$$

As Weyuker herself observes, this is only appropriate for syntactic measures. If we were concerned with cognitive complexity or programming style naming might be thought to be highly significant.

Axiom 9: The measure must permit synergistic concatenations.

$$\forall\, p,q:\text{program} \cdot |p| + |q| < |(p \circ q)|$$

This is a generalised form of Prather's second and third axioms since the type of component remains unspecified, but also more stringent as the axiom demands, rather than accepts, synergy.

Weyuker's axiom set is considerably more restrictive than Prather's, but this in turn creates complications. The properties that are required of a measure depend, to a large degree upon the its purpose and the type of scale adopted. In developing a general axiom set, one is confronted with a dilemma. The axiom set is either generalised but weak, as in the Prather approach or more restrictive, but rejects measures on grounds that are not believed to be undesirable, as in the case of Weyuker's axioms. The latter point can be demonstrated with regard to her second axiom. Suppose there is a design metric which assigns a value to a particular design. From this design we are able to generate an infinite number of possible programs that implement it, and thus the metric violates the axiom, since it demands that there be at most a finite number of programs for a given measurement value. No design measure is therefore able to fully satisfy her axiom set; manifestly an unacceptable position to adopt.

Consider, also, Axiom 5 concerning the property of monotonicity. Whether this is desirable depends upon the measurement scale adopted, for nominal scales it would be inappropriate. It is also dependent upon the meaning of concatenation, as has already been remarked, and in particular whether we admit closure or not (i.e. whether any two objects may be concatenated) [Luce69, Kran71]. This is important for data flow [Ovie80] and information flow metrics [Henr81a, Card88, Shep88d] where by adding extra program components, one might decrease the measure because the measures are based on inter-modular or block flows. Weyuker presents such an example [Weyu88]. The problem is: do we allow the concatenation of any group of program components to any other group components? The answer must be no for two reasons. First, measures of syntactically incorrect programs are meaningless. Second, for non-code metrics, such as design, the components of interest are not program statements. Consequently there is no closure of concatenation and therefore Axiom 5 is not universally applicable to software measures.

The concept of axiomatisation of measurement is powerful. It allows a formal description of desired and undesired model behaviours, which is invaluable for the theoretical evaluation of metrics and their associated models. Unfortunately, neither the weak nor the strong axiom sets are sufficient for even a significant subset of software metrics. It is therefore necessary to consider a new approach of tailored axiomatisations.

(c) Tailored axioms

In order to steer a course between the Scylla of weak, undiscriminative axiom sets, and the Charbydis of strong, restrictive axiom sets, for acceptable software metrics and their underlying models, it is necessary to tailor axioms to specific measures and models[16].

Measures must satisfy three classes of axioms:

- those axioms that are fundamental to all measurement;
- axioms necessary for the type of scale adopted;
- axioms specific to the model underlying the measure.

It will be noted that the axiom classes decrease in scope of application from universal to specific for a single, or small family of metrics. Each class will be reviewed in turn.

---

[16] An alternative flexible approach has been presented by Zuse and Bollmann [Zuse89] in the form of viewpoints which allow for the specification of varying sets of fundamental requirements for different metrics, or even the same metric. The method described in this thesis differs in that it employs an equational rewrite system to define and reason with the axioms.

The following are axioms that must hold for all measurement for it to be meaningful.

Axiom 1: It must be possible to describe, even if not formally, the rules governing the measurement [Pfan68][17]. This axiom is somewhat difficult to apply in practice, but in essence, once the error-proneness of the measuring process has been accounted for, all measurements of the same object or process must assign it to the same equivalence class.

Axiom 2: The measure must generate at least two equivalence classes in order that, as Weyuker [Weyu88] points out the measure be capable of discrimination.

Axiom 3: An equality relation is required[18]. Without an empirical equality operation each measurement, if it could be called that, would generate a new equivalence class with exactly one member.

Axiom 4: The previous axiom is further strengthened such that if an infinite number of objects or events are measured, eventually two or more must be assigned to the same equivalence class. This is a restatement of Weyuker's third axiom [Weyu88]. We note that some forms of measurement using a nominal scale, for example car number plates, do not satisfy this axiom - a hardly surprising observation when one considers that such a process must lie at the limits of what could reasonably be called measurement.

Axiom 5: The metric must not produce anomalies (i.e. the metric must preserve empirical orderings). In other words the Representation Theorem [Supp71, Kran71] must hold.

$$\forall p,q:\text{object} \cdot P\, r_e\, Q \rightarrow |P|\, r_n\, |Q|$$

where r_e is any empirically observable relation and r_n is the equivalent relation within the number or measurement system.

Axiom 6: The Uniqueness Theorem must hold [Supp71] for all permissible transformations for the particular scale type (i.e. there is a homomorphism between the transformed and the measurement structures).

Regarding the second class of axioms, those that are sufficient for different measurement scales are well documented in the classical measurement literature, (for example Stevens [Stev59] and Krantz *et al* [Kran71]) and have already been reviewed

---

[17]This does not imply that the rules must always be applied correctly, since there is the possibility of error in the measurement process - a point eloquently made by Henry Kyberg [Kybe84] amongst others.

[18]This is not dissimilar in impact to Weyuker's third axiom [Weyu88].

earlier in this chapter. Clearly our axiom set must be tailored to take account of scale and this is a fundamental decision for the software metrologist.

The third class of axioms are those that relate to the specific model underlying the measure in question. Again, it is possible to provide categories under which axioms may be selected. These are:

- resolution;
- empirically meaningless structures;
- model invariants.

Under resolution it may be desirable to include Weyuker's second axiom that asserts that there only exist a finite number of objects of a given measurement score. This would be important if metrics insensitive, in certain respects[19], are to be avoided. One has certain reservations as to whether there is a practical distinction between infinite and a very large number but there are, nevertheless, occasions when the axiom may emphasise required metric behaviour.

Having chosen the axioms necessary for the type of measurement one must consider the concatenation operations available for the objects or processes under scrutiny. The importance of concatenation is that it is the constructor operator, and allows us to describe different objects or processes, in a recursive [Fent86] or hierarchical [Prat87] manner. What the existing approaches fail to embrace is the possibility of metrics where there is no concatenation closure[20]. This is an important aspect of any axiomatisation, that we define meaningless structures where any measurement operation remains undefined or is described using three-valued logic in a manner similar to that outlined by Suppes [Supp59].

Model invariants are clearly going to be extremely diverse. Examples include Prather's [Prat84] second and third axioms which relate to measures of control flow structure. This is a difficult aspect of an axiomatic evaluation of a model, because in the end the choice of axioms will be dependant upon intuition and insight. Where it cannot be shown that a model satisfies such an axiom, two conclusions are possible. First, one might infer that the model is deficient in some respect, or second, that the axiom itself is inappropriate. Whatever, this axiomatic method at least draws the attention of the metrologist to such potential problem areas. It does not provide necessarily an answer.

In concluding this section, there are three points of note. Axiomatisations of software metrics are a vital tool for the theoretical validation of metrics and models, as they

---

[19]The classic example, is of course, McCabe's cyclomatic complexity [McCa76] where one may infinitely vary the number of procedure nodes for a fixed number of predicate nodes, for a program flow graph.

[20]This will be the case for any syntactic software metric.

allow exploration of the model behaviour in a more rigorous fashion. Without doubt, they represent a step forward from merely using one's intuition. They may also permit a more thorough coverage of the model behaviour than the intuitive approach, or for that matter, than many empirical evaluations, particularly where cost or availability of data is a factor.

Second, they provide a mechanism to establish certain foundational properties of the model. These are:

- consistency, so that there exists one and only one outcome for any set of inputs;
- completeness, that the axiom set is sufficiently rich that there is no set of inputs for which no outcome is prescribed;
- the model is not rejected for violation of axioms for empirically meaningless structures.

Consistency is established by showing that the axiom set exhibits the Church-Rosser property. This is unfortunately an undecidable question. There are various notions of completeness, including the Liskov-Guttag concept of sufficiently complete [Lisk86] which is weaker than the more usual mathematical definitions of completeness[21], but these are still undecidable.

Third, theoretical evaluation provides early feedback for the design and development of metrics and models. Given that empirical validation is a costly and time-consuming enterprise, any technique that helps identify models that are manifestly inadequate must be lauded.

4.4.2 Empirical criteria

A recurring theme in the history of software metrics (and of the previous chapter) has been, the presentation of empirical "evidence" supporting a metric, only for a rebuttal to be published a few years subsequently. This is well exemplified by Hamer and Frewin's critique [Hame82] of much of the empirical work claiming to support the Software Science model. The question therefore arises, are there any general criteria by which empirical validations may be judged?

In a review of empirical validations of design metrics, Ince and Shepperd [Ince88b] isolate three general factors which are pertinent for assessment of empirical validations. They are:

---

[21] An axiom set is usually said to be complete if it is impossible to add an independent axiom because *all* well formed formulae either follow from, or are inconsistent with, the existing axiom set.

- the hypothesis under investigation;
- the artificiality of the data used;
- the validity of the statistics employed.

These criteria will now be examined in more detail. Probably the most serious, and commonplace charge that can be levelled at empirical work, is that it is seldom clear what is being validated. This arises from imprecise, incomplete or metaphysical models. In such situations effort would be better directed at the model and the development of an unambiguous hypothesis rather than launching into an empirical validation. The efficacy of this is confirmed by the empirical study by Ince and Shepperd [Ince89a] of the information flow metric [Henr81a] where initial work removing anomalies and inconsistencies from the underlying model was rewarded by greatly improved empirical results, and statistically meaningful results obtained from the "cleaned up" model. It must be stressed again, that statistically meaningful empirical results derived from a meaningless model are in themselves meaningless.

The second criterion of an empirical validation, has in turn three dimensions. These are the number of data points (ideally a large number), the type of environment (ideally an industrial environment producing large scale software systems) and the type of staff (ideally professional software engineers as opposed to students). Applying this criterion to generate a five point classification of design metric validations, Ince and Shepperd [Ince88b] found less than 10% to be fully satisfactory, and more than 50% of studies investigated received the lowest classification of the scale. Subsequently the situation has improved slightly, but there still remains much to be desired.

Statistical validity is the third criterion by which to evaluate an empirical validation. First, and foremost, is the need for any evaluation to be capable of refuting the hypothesis under investigation. Randomly searching for statistically significant correlations, is almost certain to unearth some relationship. However, in absence of a clearly defined hypothesis derived from the model under study, claims of causality must be viewed as extremely tenuous. Careful controls and "null" hypotheses are methods for making it possible to refute a hypothesis. Use of LOC as a benchmark for comparison with other metrics [Bas84] is another possibility. It also introduces the possibility of statistical alternatives to correlation coefficients, for instance tests to see if data points are drawn from the same population[22].

Tests of correlation are also vulnerable to problems of dependence between the dependent variable and external variables as in the case of McCabe's cyclomatic complexity measure [Shep88a] and Yin and Winchester's graph impurity measure [Yin78, Ince88b]. In both instances the measures correlate more strongly with program

[22]Examples are the Student t test for parametric data and the Mann-Whitney U test for non-parametric data.

length than with the program attributes they purport to measure. Program length in turn correlates with program attributes. What is almost certainly occurring is that cyclomatic complexity, graph impurity and the program attributes are all dependent on program length. There is a causal association from length to attribute but not, or only very weakly, from cyclomatic complexity or graph impurity to attribute. The celebrated example, is of the course, the correlation between the spatial distribution of prostitutes and ministers of religion where there is no causality, (or at least one hopes not!) but rather the independent influence of demographic factors (i.e. they both cluster around urban areas).

Inappropriateness of statistical technique is another source of difficulty. Most frequently this reveals itself as using statistics that require normal data distributions, when such assumptions cannot be justified. The empirical validation by Henry and Kafura [Henr81a, KafH81] applies the parametric Pearson correlation test to the information flow metric that contains a quadratic term and is consequently highly skewed. A non-parametric test would of been more suitable. Yin and Winchester fall foul of a similar problem. Removal of four outlier data points reduces their correlation coefficient from 0.98 to 0.52 [Yin78, Shep88b]. *Statistics that are meaningless for the* type of measurement scale can also lead to confusion.

Selectivity of data points can be another problem area for empirical validations. Henry and Kafura eliminate four data points from their study where their model fails to predict adequately, on the basis that the modules were too complex to be changed [Henr81b]. However, this appears to be a deficiency of their model and therefore is unwarranted. A more common practice is the elimination of zero scoring data points, typically modules that do not contain errors. Again this is a statistically dubious practice as there is no way which these data points may be identified *a priori*.

This review of empirical practices may have painted a rather pessimistic picture. The criteria outlined describe an ideal empirical evaluation. In reality there exist constraints of cost, time and availability of data. It is therefore appropriate to distinguish between those features that one might regard as mandatory and those which are merely desirable. Any empirical validation must address an unambiguous hypothesis, be capable of refuting it and use statistically sound techniques. Representative data points are clearly advantageous, but this criterion must sometimes, of necessity be relaxed. In doing so the onus remains upon the researchers to demonstrate that any results obtained translate to other less artificial environments.

This section on model evaluation has described what must be construed as almost a "wish" list. It is concluded that evaluating software models is altogether more difficult than might be supposed. Consequently, it is more appropriate to address smaller and more manageable problems. The search for "Holy Grail" type metrics would not seem either productive or feasible at the present time.

4.5 A Method for the Development of Software Metrics

There has been little effort directed at the development methods for software metrics, in part due to the belated realisation of the problems involved in the field. One of the few contributions of significance is the Goal/Question/Metric (GQM) paradigm of Victor Basili *et al* [BasR87, BasR88, Romb87b][23]. However, given the importance of the model for software measurement, the GQM is only a partial methodology. We then extend this metric development methodology so that it explicitly recognises the central role of a model, and this is illustrated by a simple example. A fuller example is presented in the next chapter.

4.5.1 The Goal/Question/Metric paradigm

The GQM paradigm for software metrics [BasR88, Romb89] is based on the fact that measurement is carried out for a purpose, and that it is only within the context of a purpose or goal can it be determined which metrics might be useful. The approach can be characterised as top-down, and is marked contrast to current practice of obtaining a metric and then hunting around for some meaning.

The primary question to ask is "What is the measurement goal?". Each goal has the following attributes:

- an *object* of interest which may be either a product or process;
- a *purpose* such as understanding, characterising or improving;
- a *perspective* that identifies who is interested in the results, for instance management, software developers or the customer;
- a description of the *environment* to give a proper context to any results.

The next step is to refine a goal (or goals) into one or more questions. This is a process of substituting metaphysical concepts such as "complexity" and "quality" by more concrete and measurable definitions. The danger inherent in a top-down approach is that goals may be set that cannot be easily satisfied; some concepts may defy translation into more specific and quantifiable questions [Romb89]. In these circumstances omissions should be noted as they have considerable bearing upon subsequent interpretation of results. It may also be appropriate to review the original goals. Having identified a set of

[23]The constructive quality model (COQUAMO) methodology [Kitc87a] employs a similar hierarchic approach based around software quality factors, criterion and metrics.

questions, metrics are derived to determine what must be measured in order to answer each question.

The outcome is a hierarchy of goals, questions and metrics, where there is a many to many mapping from goal to question, and from question to metric. The strength of this paradigm is that every metric is placed into a context of answering a question in order to meet a measurement goal. Furthermore, metrics are only identified to satisfy particular goals or objectives. Rombach [Romb87b, Romb89] describes the successful application of GQM to an investigation of maintenance problems for the Burroughs Corporation.

The establishment of measurement goals by the GQM paradigm is then the first step of the Basili *et al* methodology. The next stage is to plan the measurement process by explicitly stating any hypotheses and to design data collection techniques, building in validation checks, as far as is possible. The third step is collecting the measurements, and the final step is interpretation.

Despite the seeming simplicity of the GQM paradigm, it must be viewed as a major advance for software metrics. In particular it forces problem definition followed by the identification of those metrics necessary to provide answers. Further, GQM provides a context both to understand the meaning, and with which to evaluate metrics, something that is all too often lacking from work in this field. It is also an extremely flexible approach and one which may easily applied to almost any aspect of software engineering measurement.

Modelling, or rather the lack of, is the only reservation that one has concerning the methodology. Although the identification of hypotheses suggests that there must be some underlying model, this is insufficient considering the central role of the model within metrology. As a consequence, we have modified the GQM approach, so as to provide for explicit modelling and evaluation.

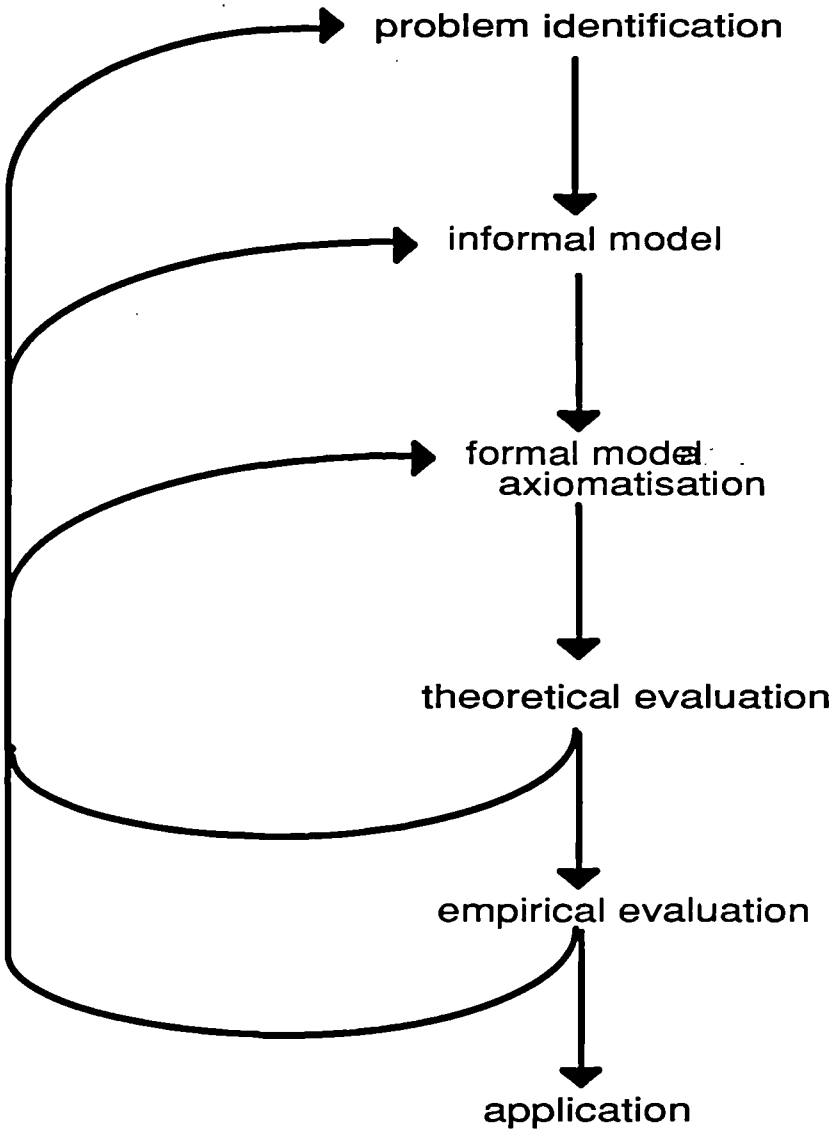


Figure 4.2: Stages of the model based metric development method

4.5.2 A model based methodology for metric development

Since a model is at the heart of our comprehension of a problem, and provides the means whereby we exclude irrelevant and unimportant factors as noise in order to concentrate upon those factors that are believed to be significant, the construction of a model ought to be at the core of our methodology. Moreover, the methodology must incorporate model evaluation and iteration where required. Software engineering processes and products are comparatively ill-understood so review and refinement must be anticipated.

Figure 4.2 presents the various stages of the methodology. This diagram suggests that there is a one-to-one mapping between a problem and a model. Obviously, this need not be the case. Some problems may be best addressed by several models, and one model may be useful for a multiplicity of problems. The latter is suggestive of the need for the re-use of models or parts of models. In the past this has been hindered by poorly and incompletely articulated models. As more structure is introduced into the modelling process, so the prospect of model and measurement re-use becomes more of a realistic possibility.

The six stages of metric development, identified by the method will be discussed in turn. These are also presented in [Shep90c].

In the same way as the GQM paradigm [BasR88], our method recognises that the first and fundamental stage is *problem identification.* The problems associated with modelling in a vacuum have already been rehearsed. Without a sense of purpose there is no means for determining what should and should not be incorporated within a model. Any description of the problem should include information concerning the problem domain. There is a considerable difference between predicting error prone modules at the design stage for a one person software house only producing simple computer games, and for the much wider domain of all software engineering activity. Problems in the latter category are unlikely to be easily solved. The problem description also needs to make clear whose problem is being solved. A company director may have a different view of productivity to the humble software engineer.

The next stage is to construct an *informal model.* Intuitions and existing software engineering knowledge is brought to bear upon the problem, in order to identify those factors that are perceived as being important. At this stage many of the factors will be metaphysical because they lack operational definitions (e.g. maintainability, ease of comprehension etc.).

A crucial part of transforming an informal into *formal model,* is fixing it into "real world" by defining the mappings between the model variables and the reality they purport to capture. The definitions must be consistent with the original problem

definition, otherwise there is a danger of "watering down" the problem into something else merely because it is easier to define or understand. At the hub of a model are the relationships that it expresses between the input and output variables. In defining these, it is often appropriate to *axiomatise* the model so as to unambiguously describe the desired model behaviour. An axiom set will also simplify the subsequent theoretical evaluation of the model. Depending upon problem domain it may be desirable to introduce parameters into the model so as to increase the its scope. The remaining two components of a well formed model are a list of assumptions that are made, and which will characterise our understanding of the problem domain, and desired level of accuracy. The latter may be obtained from the problem description. Comparison with actual levels of accuracy will help in the empirical evaluation of a model and provide a valuable basis for the comparison of competing models.

The stages of theoretical and empirical model evaluation have already been fully described in the previous section of this chapter. However, it should be stressed that model building is an iterative exercise, and provision ought to be made for backtracking. This method particularly emphasises theoretical evaluation as it is usually much quicker and less costly than empirical analysis of a model. It is entirely reasonable to suggest that the cheaper "testing" be performed first. It is not suggested that theoretical evaluation should be a substitute for empirical work as both are likely to uncover different types of problems. For example, theoretical analysis using axioms might detect "pathological" structures for which the model produces inappropriate behaviours. On the other hand an empirical analysis might detect significant factors which are not included within the model.

The last stage identified is *application*. This suggests that the model has satisfied certain theoretical and empirical criteria and therefore, depending upon the type of problem, is suitable for wider and less critical use. Evaluation of the model ought not cease once this stage is achieved, but it is regarded in a different light to the evaluation stages in order to draw attention to the distinction between nascent and therefore rather tentative models and those that have acquired rather more maturity and therefore reliability. It is not easy to think of many models that could currently be placed in the latter category - possibly some of the function point approaches that have been tuned for particular organisational environments [Albr83, Symo88].

As with the vast majority of software engineering methods, this approach is not prescriptive. Its contribution is to focus attention upon important issues; in this case modelling, and identify a set of smaller steps that lead towards the construction of an adequately evaluated model.

4.5.3 An example

It must be stressed at the outset of this example, that it is intended purely to illustrate the metric development methodology; not to provide deep insights into the problems of software maintenance. Consequently, the example is a gross simplification and is not offered as a useful model. The next two chapters provide more realistic examples of a software design model.

The starting point for the development of a software metric is the identification of a problem. Let us assume that we are interested in maintenance and wish to be able to identify hard-to-maintain systems whilst still at the design stage of developing software. This ability is only required in the restricted domain of a small software house producing very homogeneous, interactive information retrieval systems. The required level of accuracy for prediction is the identification of 75% of the systems that fall into the upper quartile of systems ranked by maintainability. Although we note even at this early stage that software maintainability has no operational definition, we postpone addressing this issue — other than by observing that our perspective will be that of the software engineer — until after we have begun to construct a prototype or informal model.

Now to develop the informal model. What factors do we think have a significant bearing upon maintainability within the problem domain? Remembering that this is a simplified example, it might be posited that there is a relationship between the number of database references a system makes, normalised for system size and the ease of its maintenance. In other words we are concerned with the density of references. The informal model has identified two input variables, a database reference count and system size that are related to the output variable maintainability. We have:

$$M = f(r/s)$$

where M is maintainability, r the count of database references and s is system size. As is frequently the case, $f(r/s)$ is an indirect measure of M since temporal considerations preclude it from being measured directly.

The next stage is the development of a formal model and an axiomatisation of desired behaviour. At the risk of being repetitive, it must be emphasised that software maintainability currently has no operational definition. We are not, therefore, in a position to measure maintainability, and it is arguable then that it is a metaphysical attribute and must, as a result be refined into terms that are directly observable. If we are unable to do so, the model cannot be anchored into reality. For simplicity's sake we will reduce our concept of maintainability to that of the percentage of modules modified per unit time per defect removed, where the most obvious unit of time is a software release. The implication is that a low percentage suggests that the

maintenance changes have been easily accommodated (i.e. high maintainability), whilst a high percentage suggests a software structure highly resistant to modification (i.e. low maintainability). Three areas require further refinement. First, what constitutes a system release, second what is a change to a module, and third when are the measurements made? Possible definitions are, any alteration that requires any part of the text of the entire software system source code to be altered in any way leading to the generation of a new version. A module change can be viewed as any change to the text within the module bounds. As regards timing of measurements, the measure of version n of the system is intended to be an indirect measure of the cost of transforming the system into version n+1. The consequence of this transformation or maintenance action, is to potentially modify the future maintainability so that the transformation from version n+1 to n+2 may be more or less difficult, since the database reference density may have been modified. Whether our measure is acceptable as an indicator of maintainability is open to question. However, at least we are making plain what is, and is not, being measured by the model.

The mapping from "real world" onto the model, in the form of database reference counts, r and system size, s must also be defined. A database is defined as any system modifiable data (i.e. not constants, type definitions etc.) that is shared by two or more modules. A reference is either the retrieval from, or the updating of a database. It might be appropriate at this stage to enquire as to whether as such information is available at the design stage. If it is not, our emerging model will not be able to solve the problem it is being developed for. System size, s will be the count of modules comprising the system. Parenthetically we note the implicit assumption that modules are relatively homogeneous, otherwise module count may be a poor choice for system size.

Finally, we require some definitions. The code realisation of a module will have the following properties:

(i) have components lexically together and visible to the software engineer either within the program text or in a library which he or she is able to update (this excludes compiler defined modules such as the Pascal readln);

(ii) have identifiable bounds (e.g. begin/end);

(iii) be referenced by name, from other parts of the program/system;

(iv) return execution to the calling software after elaboration.

The precise details of a module are obviously language dependent, as for example, a COBOL module (paragraph or section) will be very different from an Ada module (package). From this definition, it is evident that the model is not in its present form

applicable to non-procedural languages (e.g. PROLOG). Again, it is being assumed that a design or system architecture will identify all the modules and that there is at least a close approximation between the designed architecture and the realised architecture.

A database reference is defined as either a read or write access to a data object that is not private to a single module, in other words it is accessed by at least two separate modules. Furthermore at least one access must be a write access. This, therefore, excludes constants.

Although some of the above definitions may be contentious, or capable of refinement, they are at least made explicit. It is apparent that even a simple concept such as module in practice requires careful thought and definition. For this reason software metrologists might be better advised to concentrate upon simpler and more manageable metrics than software complexity metrics.

The model may now be expressed more formally as:

$$m_c/m = f(r/m)$$

where m_c is the number of modules changed per system release, and m is the total number of modules. The next step is to try to define f, which is a calibration function linking the indirect measures to the m_c/m. Before we do this it is helpful to delineate acceptable model behaviours by means of axioms. A three layered approach to flexible axiomatisation has already been outlined where the first set of axioms are those that are common to all measurement, the second those necessary for the chosen measurement scale (as yet undecided because f is undefined) and third those specific to the model. It is the last category that we wish to address.

Understanding the model behaviour focuses upon what empirical meaning we attach to the concatenation of systems. Indeed, what is the fundamental constructor operation? The primitive or indivisible components of a system that concern us are modules and database references. These can be formally described using grammar rules which define all syntactically valid concatenations. Module calling is also introduced into the grammar, because it is part of a complete system architecture, even if it does not enter explicitly into our model of software maintenance.

The grammar for our abstract system architecture notation, using an extended form of BNF[24], is as follows:

[24]Terminals are in upper case and non-terminals in lower case. Braces indicate zero or more iterations and square brackets optional elements.

```
system ::= { module } { DATABASE_NAME }
module ::= MOD_NAME [ { MODULE_CALL } ]  [ { DATABASE_REF } ]
```

Technically, the grammar is ambiguous, but since we have no intention of parsing it, this is of little concern. A concatenation is now defined as:

CONCAT: system × token → system ∪ {error}

where a token can be any terminal from the above grammar. An error condition or undefined state results from any syntactically incorrect concatenation operation. The advantage of this approach as compared to that of Prather [Prat84, Prat87] or Fenton *et al* [Fent86], is that the non-closure of concatenation is explicitly catered for. However, there remains the problem of context dependent grammar (e.g. a database reference to an undefined database). This is unimportant for our model, but there exist a number of formalisms suitable for the specification of semantics of a grammar, such as the algebraic or axiomatic methods [Gutt77, Lisk86]. These represent the behaviour of the model, or what mathematicians would call a theory, as an algebra where the axioms of such a system are rewrite rules. The next chapter employs these methods for the model of system architecture.

Having established the groundwork, the desired model behaviour can now be dealt with in more detail where p and q are systems.

Axiom 3.1 The addition of database references must always increase the measurement value of m_c.

$$\forall\ p{:}system;\ d{:}database\_reference \cdot\ |(p \circ d)| > |p|$$

This implies that multiple references by a module to the same database will be counted more than once.

Axiom 3.2 The addition of non-database reference components (modules or database declarations) will have no impact upon m_c.

$$\forall\ p{:}system;\ m{:}module\_name \cdot\ |(p \circ m)| = |p|$$

or

$$\forall\ p{:}system;\ d{:}database\_name \cdot\ |(p \circ d)| = |p|$$

This has the interesting implication, that a system may be infinitely large but make no database references and therefore minimise maintenance problems. As this is a

simplified model this need not concern us except in one respect, that of the minimum maintenance change.

Axiom 3.3 Since a system release has been defined as a change, this must be contained in at least one module, hence the next axiom

$$\forall p{:}system \cdot |p| \geq 1$$

Axiom 3.4 Similarly, the worst case for the maintenance measure is when all modules are modified. This gives

$$\forall p{:}system \cdot |p| \leq m$$

Even without formal proofs of consistency of the axiom set the conflict between Axiom 3.1 and 3.4 can be seen. There exist situations, for instance architectures containing a single module, where the addition of database references will not increase the metric. Thus the first axiom must be relaxed somewhat or the model reformulated. A more minor point is that the metric must yield integers because there is no empirical counterpart to a partial module! The strength of this rather theoretical approach is that problems of model behaviours are identified early on, and explicit decisions made. The fundamental point is not what is the best decision but the fact that the decision must be incorporated within the model. It is unsatisfactory to omit this information, allowing different individuals to make different interpretations.

A final axiom is required before f can be fully identified.

Axiom 3.5 The addition of a database reference will always have the same impact upon the metric, irrespective of the number references made, subject only to the limitations to m_c described above.

$$\forall p,q{:}system; d{:}database\_refs \cdot (|(p \circ d)| - |p|) = (|(q \circ d)| - |q|)$$

Subject to $1 \leq m_c \leq m$

From this we may deduce that function f will be linear[25] within the range 1 ... m

[25]The axiom of linearity has profound implications upon the model and causes potential complications concerning the satisfaction of $1 \leq m_c \leq m$. A curvilinear relationship would both seem more plausible and convenient.

and so will have the general form:

$$\alpha + \beta.(r)$$

The axioms give little clue to the values for the coefficients α and β, other than β must be greater than zero. In practice there exists a greater interplay between the different stages of the methodology than first might be apparent. Almost inevitably recourse to empirical analysis is required in order to provide specific values for the coefficients. Were the model to be extended to differing domains it is likely that differing values would be required.

At this stage we can consider what type of scale and unit we are dealing with. Clearly, $\alpha+\beta.(r)$ leads to an interval scale. This can be demonstrated by the fact that the following empirical operations are available:

determination of equality;
determination of greater or less;
determination of equality of intervals;

but not:

determination of equality of ratios.

The transformations that do not result in the loss of ordering are similarity and linear transformations. Such observations may appear trivial in this instance, however, this is not necessarily so for some of the other metrics that have been proposed, and indeed the absence of any obvious empirical additive operation makes the determination of scale and unit an area of difficulty.

The assumptions that the model makes are that modules are of roughly equal size. It is also assumed that database references are generally homogeneous, and that all module maintenance changes are comparable. Furthermore, we must assume that implemented systems do reflect their designs and we have not considered the possibility that maintenance changes may involve the addition of new, or the removal of existing modules.

Having established the necessary theoretical foundations it is now appropriate to empirically investigate our model of software maintenance. For this to take place, suitable hypotheses need to be established and the necessary data collection carried out. We must also establish how error prone this measurement process is, and what are its limitations. Empirical work will also enable us to see what sort of restrictions our assumptions place upon the model - the worst case being that they render the model unworkable. As has been suggested, model building is an iterative process so as our

understanding matures additional factors are incorporated into the model and the estimates for the coefficients refined. In this way we are slowly able to proceed to the perception of software engineering as a system that consumes multiple resources and engineers software systems that have multiple facets to be characterised [KafC85]. This then necessitates the collection of more than two metrics in order to indirectly measure maintainability.

Even the simplified example of a maintainability model has demonstrated that software modelling is not a trivial process and has many ramifications. This in itself is a compelling reason for the adoption of a little more rigour and an attempt to eliminate at least some undesirable model properties early on in the development of software metrics. The attendant reduction in heartache and wasted effort by those attempting to validate metrics might be regarded as a bonus.

4.6 Summary

This chapter has attempted to establish some foundations for software metrics. The development of a model is fundamental to meaningful measurement. Therefore, the model must be defined in an unambiguous and complete a fashion as possible. To that end it has been suggested that a well formed model should specify seven different aspects of the problem being modelled. These are inputs, outputs, parameters, relationships linking inputs to outputs, mappings between the "real world" and the model, assumptions made and reliability of predictions. These components are seldom brought into the open by existing models of software. This state of affairs is the primary contributor for the difficulties encountered with software models described in the previous two chapters.

Modelling is not a simple enterprise and therefore development methods are of value. The GQM paradigm of Basili *et al* [BasR88] is such a method, but unfortunately it does not sufficiently emphasise the construction of a model. An alternative model-based metric development method has been presented, which outlines the steps whereby a measurement goal may be evolved from a high level statement of a problem or opportunity into an informal model, to a formal model and then evaluated both theoretically and empirically. The theoretical evaluation has been stressed since it has not been given great attention in the past. Its particular value is, of course, that such techniques are almost invariably a good deal less resource consuming, than empirical studies. This is not to decry empirical work, but merely to observe that if we are expend considerable effort in an empirical validation of a model it should at least be internally consistent and satisfy certain criteria. Furthermore, theoretical analysis may uncover different problems with a model to those found by empirical investigation. Where it is possible to articulate a required model behaviour a mathematical proof may afford a higher degree of confidence than an empirical study, which in some ways

is akin to sampling from a large and probably heterogenous population, with no certaintity that the sample is representative. On the other hand there are many situations where it is not possible to state *a priori* what model behaviour is required. In such circumstances empirical investigation is likely to be more effective. Empirical evaluation is also likely to be more effective at highlighting models that are insufficiently broad in scope or that make unrealistic assumptions. To repeat then, both forms of model evaluation are complementary and necessary.

Axiomatic approaches, have been developed by Prather [Prat84] and Weyuker [Weyu88], for the theoretical validation of models. Unfortunately no universal set of axioms exists for all software engineering models, and so a flexible or tailored approach must be embraced. Recourse to classical measurement theory does show that certain axioms are basic to all measurement, and some to particular measurement scales. Consequently the three layered hierarchy of axioms that we propose is an effective approach to model axiomatisation.

Although the tenor of this chapter has been theoretical, as Einstein's aphorism (quoted at the beginning of the chapter) reminds us, formality alone is not sufficient. Indeed the essence of measurement is the mapping of empirical relations, drawn from an inherently informal world, into a formal model [Stev59]. However, the application of a little more rigour will make the development, refinement, validation and application of metrics a considerably less fraught process than is the present case.

5. A UNI-DIMENSIONAL MODEL OF SOFTWARE DESIGN

"When we meane to build, We first suruey the Plot, then draw the Modell"
William Shakespeare

Synopsis of chapter

It has already been demonstrated that in absence of a clearly articulated model, software measurement is an fruitless exercise. In the previous chapter we sought to remedy this situation by proposing a method to aid the development and evaluation of software metrics and their underlying models. This was illustrated by an extremely simple example. We now turn to a much larger software engineering problem for which we develop a model to relate software design and the quality factors of implementability, reliability and maintainability. A formal model is developed and evaluated using the apparatus of the method described earlier. Each step in the development of the model is outlined, namely problem identification, construction of an informal model, statement of the formal model, and axiomatisation of the model. This is followed by theoretical and empirical validations.

The worth of the method is highlighted by its ability to detect both theoretical problems - for example inability to handle software re-use - and the empirical findings indicating problems related to large variations in module size. Lastly, it is concluded that more than one model input or multidimensional modelling is necessary to improve the model performance and provide useful support for the software engineer carrying out design tasks.

5.1 Restatement of the Problem

The aim of this chapter is to develop a quantitative model of software design or architecture using the method and ideas outlined earlier in this thesis. Two benefits are anticipated. First, it is a realistic example with which to assess the method. Second, a validated model of design and associated metrics will be of considerable value to practicing software engineers.

In this section the first two stages of the development of a model are given consideration. Fundamental is the identification of the problem, if only because modelling the universe, or even just that subset related to software engineering is a moderately ambitious undertaking! The problem statement is then refined into an informal model establishing the major inputs and outputs. As the chapter titles makes clear, our model is characterised as a uni-dimensional model based upon a single input. This somewhat simplistic approach is intentionally adopted for two reasons. First, the empirical validation of multidimensional models is an extremely complex process[1]. And, second, it seemed conceptually more straightforward to start from a simple foundation and introduce complications only as they become necessary. Hence, this chapter deals with the simple uni-dimensional model and the next chapter introduces additional factors leading to the construction of a more sophisticated multidimensional model.

5.1.1 Problem identification

The problem is one of controlling the software design process in order to produce software systems exhibiting the following quality factors:

implementability;
reliability;
maintainability.

Such problems have been exercising researchers for a number of years (e.g. [Stev74, Jack75, Parn79, Jack82, Booc86] to name but a few). Design may be conceived of as a process of selecting between alternatives with the objective of providing an abstract solution[2] for a given specification or problem. Indeed, if no alternatives exist then the process is a mechanical one offering no prospect of improvement. In order to avoid being an entirely stochastic process, selection must be made on the basis of evaluation criteria. Our objective then, is to seek to provide software engineers with a means of discriminating between alternative designs in order to improve the software quality factors listed above. Furthermore, we wish the design evaluation criteria to be quantitative.

Although being able to contrast designs is fundamental, this pre-supposes that one has

---

[1] The work by Kafura and Canning on multiple resources is almost the only foray into this area, other than the cost estimation models such as COCOMO [Boeh81] and SOFTCOST [Taus81]. Unfortunately, some of the latter seem to perform poorly other than upon their own databases [Moha81, Cont86].

[2] The solution is abstract in the sense that it is not executable. Were there to exist a suitable target machine upon which it could run, then of course, the design would be an implementation.

a repertoire to draw from. In many cases the starting point may well be a single design in which case we wish, not only to be able to contrast designs, but also to pin-point weaknesses within a design. This facilitates the generation of new - and hopefully improved - designs.

Such objectives raise the question as to the relationship between this research and the various design methodologies that are emerging, such as object oriented design [Booc86]. It is our view that the use of metrics as quantitative criteria with which to evaluate software designs is complementary to methodology because although a methodology may greatly restrict the search space for candidate designs, no methodology which we are aware of restricts the space to a single design[3] - certainly not for non-trivial problems. The other key difference in approaches is that even when a methodology provides explicit evaluation criteria, as in the case of Structured Design [Myer75, Stev80], these tend to be qualitative and therefore, not amenable to automation. For designing large software systems, automated tools are invaluable.

Finally, it must be stressed that our goal is emphatically not to be able to accurately predict any of the above quality factors from a given design. Accurate prediction requires a far deeper understanding of software engineering processes than we currently possess [Shep89a]. Nor is it a likely outcome from a comparatively simple uni-dimensional model.

5.1.2 An informal model

The first issue to address is, what aspects of a design should be included in our embryonic model? Work by other researchers (e.g. [Henr81, Kafu87, Romb87b]) suggests that structural, or architectural, aspects of a design are an important determinant of the software quality factors we are interested in. System architecture also yields the benefit of being available early on in the design phase, thus allowing the more scope for strategic decision making, since few resources are yet committed. For the same reason, backtracking and re-working of designs is comparatively cheap.

Our goal imposes the constraint that the metrics must be available at design time[4].

---

[3]The author's experience of teaching even the highly prescriptive methods such as JSP [Jack75] still supports this proposition. Students were still able to generate surprisingly diverse solutions, particularly with respect to logical input data structures and whether to employ a backtracking strategy or not.

[4]This is an important restriction - where metrologists allow themselves the luxury of near perfect foresight, the resulting metrics cannot be integrated into software engineering processes without extreme difficulty. Henry and Kafura's Information Flow metric [Henr81a] provides the classic example where their design metric requires LOC as one of its inputs. COCOMO [Boeh81] similarly requires an estimate of LOC

Thus, we restrict ourselves to the following information:

- modules;
- the calling structure;
- the module interfaces (i.e. the data objects that each module imports and exports);
- global data structure references.

The next problem is to decide what aspects of system architecture to capture. All three quality factors have a software engineering perspective, in that they involve the software engineer in work - either to develop, repair or modify the software. Informally our model is that these tasks are easier to perform the more localised they are. Taking a module as the fundamental unit, flows of information between modules are the means whereby one module may impact another module. If there is the possibility of an information flow, the software engineer must examine additional modules and the task becomes increasingly global in nature. Thus, the fewer connections that exist between a module and other modules, the easier it is to implement and maintain, and the fewer errors it contains. This gives a justification for treating three software quality factors in one model - that there is a unifying process of comprehension by the software engineer[5]. Such a model is in line with current developments in design methodology and preferred system architectures (e.g. [Alex64, Parn72, Myer75, Your79, Booc86]).

As already stated our intention is the development of uni-dimensional model. However, we did not feel it entirely appropriate to ignore application domain. There is, therefore, also the background problem of establishing the extent to which this model is applicable. Consequently, specific application domains are identified. The empirical validation applies the model to interactive systems and realtime embedded systems.

---

very early on in the software life cycle.

[5] Hindsight would suggest that combining several quality factors into a single model leads to problems, particularly where tradeoffs exist (e.g. development effort could be reduced at the expense of reliability). These problems are enlarged upon, at the end of this chapter.

5.2 A Formal Model

5.2.1 Definitions

In order to achieve the goal of aiding software engineers select suitable designs the extremely informal model described above must be crystalised into something more specific. The following definitions are therefore introduced.

A *module* is an executable unit of software that may be called by name, returns execution to the caller after elaboration and is identified by the system architecture[6]. *Information flows* between modules either by means of parameters (see Figures 5.1 and 5.2) or via global data structures (see Figure 5.3).

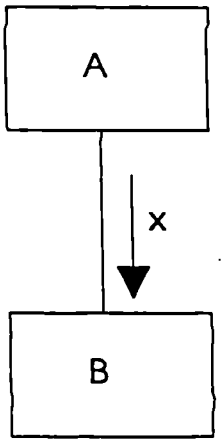


Figure 5.1: Example information flow between module A and B

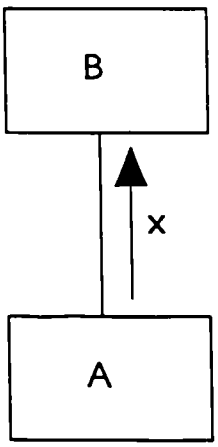


Figure 5.2: Example information flow between module A and B

[6] A more precise definition for Pascal was found to be necessary once the empirical analysis had commenced, due to the unforeseen need to "reverse engineer" designs from code, for parts of the study. The principle, however, remains unchanged.

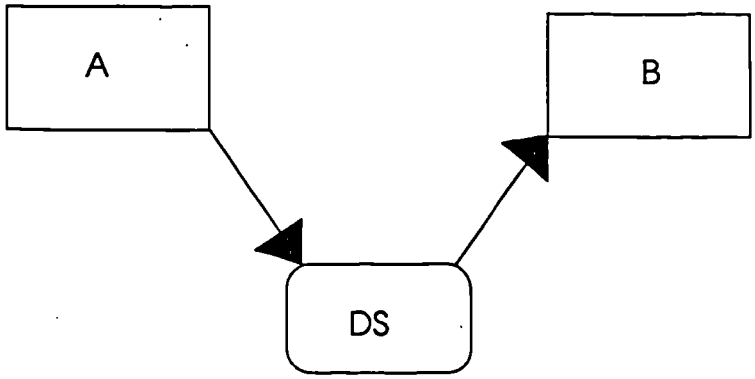


Figure 5.3: Example information flow between module A and B via a global data structure

A *global data structure* is any variable that is shared between more than one module and may be permanent such as a file or temporary such as an array.

From these basic definitions, the fan_in and fan_out of a module may be derived, where the *fan_in* is the number of information flows terminating at a module and the *fan_out* is the number of flows emanating from a module. Since, our model is concerned only with the number of connected modules, duplicate flows are ignored. These counts are then combined to give an information flow metric IF4 (so called due to it being the fourth version - versions one to three having been discarded!), which for the module m, is calculated as follows:

$$IF4_m = (fan\_in \,.\, fan\_out)^2$$

The fan_in is multiplied by the fan_out to give the number of unique information paths through a module. This is then raised to the power of two, in order to differentiate between architectures where information flows are evenly distributed amongst the modules and those architectures that exhibit a clustering of flows around a small number of modules[7]. Despite the seeming arbitrary nature of a power law of two, the approach is in line with Belady and Evangelisti's [Bela81] method for partitioning systems. It is also similar in formulation to the Henry and Kafura metric [Henr81a]. In order to contrast complete systems we use a system metric:

[7] The value of the quadratic term has been questioned, since most empirical work deals only with weak orderings. It is significant, however, when working at the system level, since the metric sums across modules.

$$IF4 = \sum_{1}^{n} IF4_m$$

where there are n modules.

Although this work is indebted to the pioneering research of Sallie Henry and Denis Kafura [Henr81a], our definitions of information flow are substantially modified, thereby eliminating many of anomalies contained in their original metric and underlying model. These anomalies are described both in Chapter Three and elsewhere (e.g. [Kitc88, Shep89b, Ince89]).

The model is now described on a component by component basis as suggested in the previous chapter.

Inputs

Design notations, such as module hierarchy charts, clearly identify the following required inputs defined as empirical relations:

imports(module, parent_module, parameter_list)
exports(module, parent_module, parameter_list)
reads(module, global_data_structure)
writes(module, global_data_structure)

From these relations the information flow measure IF4 of system structure may be derived.

This then leaves us with the question, are there any circumstances when it is ambiguous as to how to define modules, parents, parameters or global data structures? Some of these issues are tackled by the algebraic model specification, such as the meaning of parent and recursion. In other situations the problem derives from the type of design notation from which this information must be extracted, in which case it is difficult to offer much general guidance.

Outputs

The model output, in this case the software quality factors, are more dependant upon experimental and practical limitations. Again by reference to our objective it is apparent that ranking is sufficient for comparison purposes, hence the outputs are:

Development effort rank
Reliability rank
Maintainability rank

Relationships

From reference to the model goals, it is apparent that we need to be able to rank system architectures in terms of the three software quality factors of interest. This gives the following relationships:

Development effort rank = IF4 rank
Reliability rank = (1 / IF4) rank
Maintainability rank = IF4 rank

Parameters

The model is not parameterised.

Accuracy

Using the ranks the model should correctly identify components belonging to the highest quartile 80% of the time or to use the terminology of Kafura and Canning [KafC85] have a yield in excess of 80%. Although a somewhat arbitrary threshold 80% was selected for two reasons. First a yield of 80% implies an error rate of 20%. The model will wrongly identify 20% of structures as belonging to the most problematic quartile resulting in mis-allocation of resources and possibly inappropriate decisions. For the model to be of any practical utility this cannot be a frequent occurrence. Second, an accuracy level of plus or minus a quarter 80% of the time was the stated objective of the COCOMO model [Boeh81, Boeh84]. Since COCOMO is intended as a serious model for industry use the level of accuracy would seem appropriate for our model.

Assumptions[8]*:*

(i) the implementation architecture accurately reflects the architecture specified in the design;

(ii) that variations in information flow size, data structure complexity and module size will not have a material bearing upon the model.

5.2.2 An algebraic model specification

The above model description is still not fully formal, particularly if we wish to evaluate its characteristics from a theoretical stand-point. First the exact behaviour of the model must be defined, so that for any given set of model inputs the outputs may be calculated. One approach to a more precise description is the use algebras, sometimes known as axiomatic specifications [Gutt77, Lisk86]. Second, we must formally state the desired characteristics of our measure as a set of model invariants. To do this the tailored axiomatic approach outlined in the previous chapter will be employed. Once this has been accomplished we can set about the task of investigating whether there exist inputs which result in violations of the model invariants.

In order to develop an algebra to formally specify a model it is first necessary to consider the constructor operations [Geha83], bearing in mind that not all feasible concatenations will yield meaningful system architectures. For example, a flow between two data structures has no meaning in our model - a data structure must be accessed by a module. Thus we commence with a grammar for our representation language of system architectures. Since the central issue of our model is information flow connections between modules, a directed graph is a convenient tool. We distinguish between two types of node, modules and global data structures. The edges show flows of information, either from one module to another as parameters or between a module and a data structure indicating update or retrieval from the data structure by the module.

Figure 5.4 presents an example of a system architecture which can represented as the graph in Figure 5.5. From Figure 5.4 we observe that there is no flow of information, either via parameters or via data structures between the module INIT and the rest of the system hence the graph in Figure 5.5 is not connected. Similarly the modules OUT-SCREEN and MAKE-HAND are an isolated subsystem represented as another subgraph.

---

[8] We will return to the validity of these assumptions at the end of the chapter-at present we merely record the fact that they are being made.

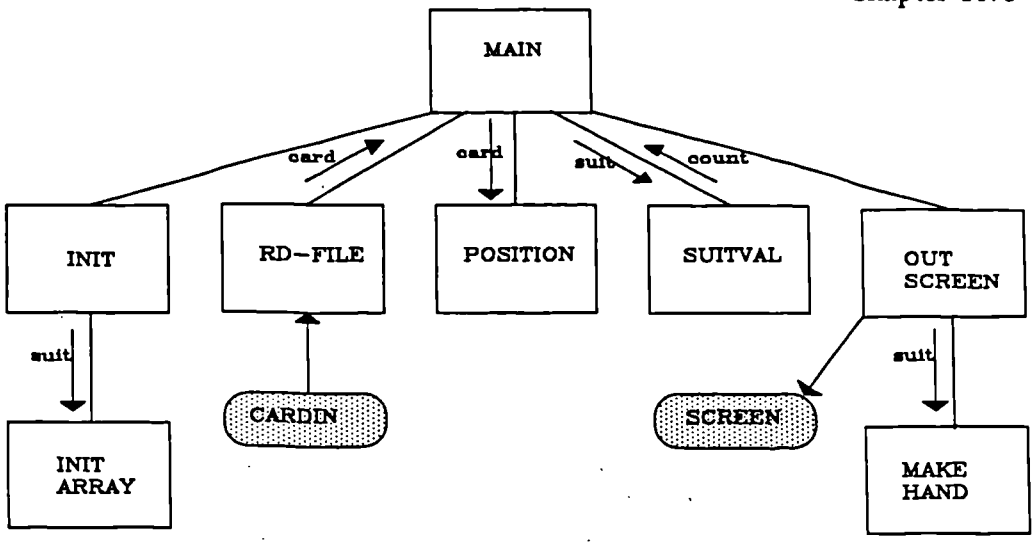


Figure 5.4: Example system architecture

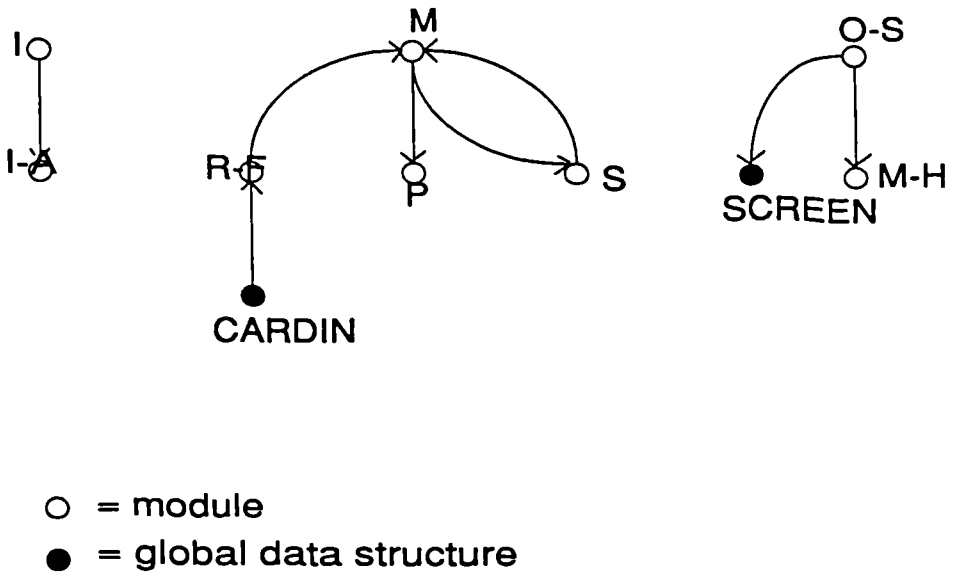


Figure 5.5: Graph representation of the system architecture

There is, however, a link from OUT-SCREEN to MAKE-HAND because MAKE-HAND imports the parameter "suit". In reality one might have doubts concerning the utility of system comprising three entirely independent subsystems but that is an issue beyond the scope of the present discussion. It is also noteworthy that there are no module to module links via CARDIN or SCREEN, the two global data structures because no module writes to CARDIN and no module reads from SCREEN. Again, a rather improbable state of affairs. The edges are not identified as our concern is only whether a flow exists or not. Since it is a directed graph, the existence of a flow from one module

to another does not imply a reverse flow unless explicitly identified as in the case of MAIN and SUITVAL.

There are certain restrictions upon the building of a graph to describe system architecture. First, to avoid any possibility of ambiguity, nodes (either modules or global data structures) must have unique names. Second, an edge must link two nodes. We avoid, at least at this stage, the issue of recursion and further stipulate that an edge cannot link a node to itself. Finally, an edge may not link two data structure nodes.

This approach yields the advantage of simplicity since we only have two types of object to deal with: nodes and edges. Consequently there will be two constructor operations[9]:

add_node
add_edge

These will have signatures as follows:

add_node: graph × node_type × name → graph ∪ {error}
add_edge: graph × name × name → graph ∪ {error}

Note that since the graph is directed, the flow will be from the first node (i.e. the second argument) to the second node (i.e. the third argument) of the ADD_EDGE operation. Unfortunately, these operations have one undesirable property for constructors, namely they are not deterministic because we cannot predict *a priori* whether the constructor operation will succeed, or whether it will yield an illegal structure and therefore result in an error being returned). To resolve this problem, two further constructor operations are required which will be deterministic; these we will call CONCAT_N and CONCAT_E. They are referred to as internal operations - instead of external operations - because they are only present to facilitate the formal specification process.

We now algebraically specify the construction of meaningful system structures in terms of graphs as:

types
graph
node_type = (module , global_data_structure)
name = string

vars

[9] The full specification is given in Appendix A.

S_1 , S_2 : graph
t_1 , t_2 : node_type
n_1 , n_2 , n_3 , n_4 : name

external operations

add_node: graph × node_type × name	→ graph ∪ {error}
add_edge: graph × name × name	→ graph ∪ {error}

internal operations

concat_n: graph × node_type × name	→ graph
concat_e: graph × name × name	→ graph
new:	→ graph
exists: graph × name	→ boolean
linked: graph × name × name	→ boolean
is_a_module: graph × name	→ boolean
is_a_ds: graph × name	→ boolean

Apart from CONCAT_N and CONCAT_E, five other internal operations are introduced. NEW is required to enable an initial state to be defined for the graph (i.e. an empty system architecture) and to detect a boundary condition for the recursive application of other operations. EXISTS, LINKED, IS_A_MODULE and IS_A_DS facilitate the checking for error conditions when building system architectures. More specifically, EXISTS tests whether a named node exists in the graph, and LINKED whether there is an edge leading from the first to the second node. IS_A_MODULE and IS_A_DS check the type of a named node.

Meaningful, and therefore allowable, concatenations are now defined by the following equations:

add_node(S_1,t_1,n_1) = **if** exists(S_1,n_1)
then {error}
else concat_n(S_1,t_1,n_1)

add_edge(S_1,n_1,n_2) = **if** exists(S_1,n_1) ∧ exists(S_1,n_2)
∧ ¬ linked(S_1,n_1,n_2)
∧ n_1<>n_2
∧ is_a_module(S_1,n_1) ∨ is_a_module(S_1,n_2))
then concat_e(S_1,t_1,n_1)
else {error}

exists(new,n_1) = FALSE

exists(concat_n(S_1,t_1,n_2),n_1) = if n_1=n_2 then TRUE
else exists(S_1,n_1)

exists(concat_e(S_1,n_2,n_3),n_1) = if n_1=n_2 ∨ n_1=n_3 then TRUE
else exists(S_1,n_1)

linked(new,n_1,n_2) = FALSE

linked(concat_e(S_1,n_3,n_4),n_1,n_2) = if n_3=n_1 ∧ n_4=n_2
then TRUE
else linked(S_1,n_1,n_2)

linked(concat_n(S_1,t_1,n_3),n_1,n_2) = linked(S_1,n_1,n_2)

is_a_module(new,n_1) = FALSE

is_a_module(concat_n(S_1,t_1,n_2),n_1) = if n_1=n_2 ∧ t_1=module
then TRUE
else is_a_module(S_1,n_1)

is_a_module(concat_e(S_1,n_2,n_3),n_1) = is_a_module(S_1,n_1)

is_a_ds(new,n_1) = FALSE

is_a_ds(concat_n(S_1,t_1,n_2),n_1) = if n_1=n_2 ∧ t_1=global_data_structure
then TRUE
else is_a_ds(S_1,n_1)

is_a_ds(concat_e(S_1,n_2,n_3),n_1) = is_a_ds(S_1,n_1)

N.B. The fact that IS_A_DS is false does not imply IS_A_MODULE is true, or *vice versa*, since both will return FALSE if the node n_1 does not exist. Therefore both these internal operations are required.

The above is a formal specification of which graphs may be constructed to represent valid system architectures. So, part of the architecture given in Figures 5.4 and 5.5 (the subsystem incorporating OUTPUT-SCREEN, MAKE-HAND and the data structure SCREEN) may be described as the following sequence of concatenation[10] operations:

new

add_node(new,module,output-screen)

add_node(add_node(new,module,output_screen),module,make-hand)

add_node(add_node(add_node(new,module,output-screen),module,make-

[10] One of the properties of our model is that the order of concatenations is immaterial other than nodes must precede the introduction of connecting edges. A proof will be given later in the chapter.

hand),global_data_structure,screen)

add_edge(add_node(add_node(add_node(new,module,output-screen),module,make-hand),global_data_structure,screen),output-screen,screen)

add-edge(add_node(add_node(add_node(new,module,output-screen),module,make-hand),global_data_structure,screen),output-screen,screen),output-screen,make-hand)

Although this may appear rather arcane, it is merely the successive application of three ADD_NODE and two ADD_EDGE operations to a NEW graph. Using this technique any legal system architecture can be unambiguously described.

Using the same algebraic approach the IF4 measures can be formally defined. Two additional external operations are introduced:

IF4: graph	$\rightarrow$ nat
IF4m: graph $\times$ name	$\rightarrow$ nat $\cup$ {error}

The IF4 operation returns the information flow measure for the entire system and is deterministic. The IF4m operation returns the information flow measure for the module named, and is not deterministic because the name may refer to a node that is either not a module or that is not present within the system.

IF4 is defined as being the sum of applying IF4m to every node in the graph that is of the type module. In order to specify this algebraically, we destructively search through the graph processing every node that is a module until the graph is empty. This creates one problem (if we are to preserve the strictly functional style of the algebra), in that the complete graph is needed to determine all information flows for each module because each module may be linked to any other, or indeed every other node in the system. Hence we introduce yet another internal operation, IF4_INT that has two arguments, the complete system and the system remaining to be searched.

if4_int: graph × graph → nat

This gives:

$if4(S_1)$ = $if4\_int(S_1,S_1)$

$if4\_int(S_1,new)$ = 0

$if4\_int(S_1,concat\_n(S_2,t_1,n_1))$ = **if** t_1=module
then $if4m(S_1,n_1) + if4\_int(S_1,S_2)$
else $if4(S_1,S_2)$

$if4\_int(S_1,concat\text{-}e(S_2,n_1,n_2))$ = $if4(S_1,S_2)$

Now we address the problem of defining IF4m, the information flow measure for a specific module. By now it will come as little surprise to the reader that the following internal operations must be added to the specification!

fan_in_l: graph × name → nat
fan_out_l: graph × name → nat
fan_in_g: graph × graph × name → nat
fan_out_g: graph × graph × name → nat

Each of these operations determines the number of local or global information flows into, or out of, the specified module. Local flows are information flows via parameters, whilst global flows are flows via shared data structures. The operations that count global flows have two graph arguments to deal with the problem of destructive searching, for exactly the same reasons as the previously defined IF4 operation. These may be combined to define IF4m as:

$if4m(S_1,n_1)$ = **if** $is\_a\_module(S_1,n_1)$
then $sqr(\ ((fan\_in\_l(S_1,n_1) + fan\_in\_g(S_1,S_1,n_1)) *$
$((fan\_out\_l(S_1,n_1) + fan\_out\_g(S_1,S_1,n_1))\ )$
else {error}

$fan\_in\_l(new,n_1)$ = 0

$fan\_in\_l(concat\_e(S_1,n_2,n_3),n_1)$ = **if** $n_3=n_1 \wedge is\_a\_module(n_2)$
then 1 + $fan\_in\_l(S_1,n_1)$
else $fan\_in\_l(S_1,n_1)$

$fan\_in\_l(concat\_n(S_1,t_1,n_2),n_1)$ = $fan\_in\_l(S_1,n_1)$

$fan\_out\_l(new,n_1)$ = 0

$fan\_out\_l(concat\_e(S_1,n_2,n_3),n_1)$ = **if** $n_2=n_1 \wedge is\_a\_module(n_3)$
then $1 + fan\_out\_l(S_1,n_1)$
else $fan\_out\_l(S_1,n_1)$

$fan\_out\_l(concat\_n(S_1,t_1,n_2),n_1)$ = $fan\_out\_l(S_1,n_1)$

$fan\_in\_g(S_1,new,n_1)$ = 0

$fan\_in\_g(S_1,concat\_e(S_2,n_2,n_3),n_1)$ = **if** $n_3=n_1 \wedge is\_a\_ds(n_2)$
then $ct\_globals\_in(S_1,S_2,n_1,n_2) + fan\_in\_g(S_1,S_2,n_1)$
else $fan\_in\_g(S_1,S_2,n_1)$

$fan\_in\_g(S_1,concat\_n(S_2,t_1,n_2),n_1) = fan\_in\_g(S_1,S_2,n_1)$

$fan\_out\_g(S_1,new,n_1)$ = 0

$fan\_out\_g(S_1,concat\_e(S_2,n_2,n_3),n_1)$ = **if** $n_3=n_1 \wedge is\_a\_ds(n_2)$
then $ct\_globals\_out(S_1,S_2,n_1,n_2) + fan\_out\_g(S_1,S_2,n_1)$
else $fan\_out\_g(S_1,S_2,n_1)$

$fan\_out\_g(S_1,concat\_n(S_2,t_1,n_2),n_1)$ = $fan\_out\_g(S_1,S_2,n_1)$

In order to determine the number of global flows into module n_1 via global data structure n_2 the entire graph S_1 must be searched using CT_GLOBALS_IN.

$ct\_globals\_in$: graph × graph × name × name → nat

To count global flows out of module n_1 via global data structure n_2 we have:

$ct\_globals\_out$: graph × graph × name × name → nat

These are defined as:

$ct\_globals\_in(S_1,new,n_1,n_2)$ = 0

$ct\_globals\_in(S_1,concat\_e(S_2,n_3,n_4),n_1,n_2)$ = **if** $n_4=n_2 \wedge is\_a\_module(n_3) \wedge n_3<>n_1$
then $1 + ct\_globals\_in(S_1,S_2,n_1,n_2)$
else $ct\_globals\_in(S_1,S_2,n_1,n_2)$

$ct\_globals\_in(S_1,concat\_n(S_2,t_1,n_3),n_1,n_2)$ = $ct\_globals\_in(S_1,S_2,n_1,n_2)$

$ct\_globals\_out(S_1,new,n_1,n_2)$ = 0

$ct\_globals\_out(S_1,concat\_e(S_2,n_3,n_4),n_1,n_2)$ = **if** $n_3=n_2 \wedge$

$$\text{ct\_globals\_out}(S_1,\text{concat\_n}(S_2,t_1,n_3),n_1,n_2) = \begin{array}{l} \wedge\ \text{is\_a\_module}(n_4) \wedge n_4 <> n_1 \\ \textbf{then } 1 + \text{ct\_globals\_out}(S_1,S_2,n_1,n_2) \\ \textbf{else } \text{ct\_globals\_out}(S_1,S_2,n_1,n_2) \\ \text{ct\_globals\_out}(S_1,S_2,n_1,n_2) \end{array}$$

This then completes the algebraic definition of the model and the information flow metrics. It yields the advantages of being unambiguous, particularly with respect to the validity of structures, and provides the apparatus for reasoning about, and evaluating the model. Formal evaluation is made easier when combined with the next step, which is to state the properties that we desire of our model as a set of model invariants or axioms. To do this the tailored, three layer approach set out in the previous chapter will be adopted.

5.2.3 Axioms of desired model behaviour

The first set of axioms for the model are those that are fundamental to all measurement, as described in the previous chapter. Since these do not vary with model we will postpone re-rehearsing them until the next section on theoretical evaluation.

Regarding the second class of axioms, these are dependent upon our choice of measurement scale. In this instance the relationships identified within the model are concerned with input and output variable *ranks* which in turn suggests that weak ordering will be sufficient for our purposes. This implies ordinal measurement. The axioms are those of transitivity of the < and > relations and symmetry, reflexitivity and transitivity of the equivalence relation. Further discussion is given in [Kran71, Kybe84, Melt90].

The third class of axioms are those that relate to the specific model underlying the measure in question. Again, it is possible to provide categories under which axioms may be selected. These are:

i) resolution;
ii) empirically meaningless structures;
iii) model invariants.

No further axioms concerning measurement resolution are required beyond Axiom 2, which states that the measure must be capable of discrimination, and Axiom 4 that states that there must exist at least two system architectures that will be assigned to the same equivalence class.

As regards the second category; the axiomatisation of the concatenation operations CONCAT_N and CONCAT_E formally define all legal or meaningful system

architectures. Thus, no further axioms are required.

The third category of axioms are those properties specific to this model which we believe are important properties, and therefore must remain invariant.

Axiom 7: Concatenating an additional module to the system architecture cannot decrease the IF4[11] measure.

$$\forall S_1\text{:graph}; m_1\text{:node} \cdot \text{is\_a\_module}(m_1) \wedge |S_1| \; |\text{concat\_n}(S_1,\text{module},m_1)|$$

Axiom 8: Concatenating an additional data structure to the system architecture will not change the IF4 measure.

$$\forall S_1\text{:graph}; ds_1\text{:node} \cdot \text{is\_a\_ds}(ds_1) \wedge |S_1| = |\text{concat\_n}(S_1,\text{global\_data\_structure},ds_1)|$$

Axiom 9: Concatenating an additional local information flow to the system architecture must increase the IF4 measure.

$$\forall S_1\text{:graph}; m_1,m_2\text{:node} \cdot \text{is\_a\_module}(m_1) \wedge \text{is\_a\_module}(m_2) \wedge m_1 <> m_2 \wedge$$
$$|S_1| < |\text{concat\_e}(S_1,m_1,m_2)| \wedge |S_1| < |\text{concat\_e}(S_1,m_2,m_1)|$$

Axiom 10: Concatenating an additional global information flow to the system architecture must increase the IF4 measure. There is a slight difficulty in formally stating this invariant because not all edges leading to or from data structures create global flows; a structure remains a sink or source (e.g. no flow exists if ten modules update a global data structure but no module retrieves from it - admittedly a rather bizarre situation!).

$$\forall S_1\text{:graph}; m_1,ds_1\text{:node} \; \exists \, m_2\text{:node} \cdot \text{is\_a\_module}(m_1) \wedge \text{is\_a\_module}(m_2)$$
$$\wedge m_1 <> m_2 \wedge \text{is\_a\_ds}(ds_1) \wedge \text{linked}(S_1,ds_1,m_2) \wedge |S_1| < |\text{concat\_e}(S_1,m_1,ds_1)|$$

$$\forall S_1\text{:graph}; m_1,ds_1\text{:node} \; \exists \, m_2\text{:node} \cdot \text{is\_a\_module}(m_1) \wedge \text{is\_a\_module}(m_2)$$
$$\wedge m_1 <> m_2 \wedge \text{is\_a\_ds}(ds_1) \wedge \text{linked}(S_1,m_2,ds_1) \wedge |S_1| < |\text{concat\_e}(S_1,ds_1,m_1)|$$

Another important set of properties of the model are that the IF4 measures are not a monotonic function of counts of system components (i.e. modules, global data structures and flows).

[11] IF4 is being used as short hand for development effort, maintainability and unreliability.

Axiom 11:

$$\exists S_1,S_2:\text{graph} \cdot \#\text{modules}(S_1)>\#\text{modules}(S_2) \wedge |S_1|<|S_2|$$

where #MODULES is a function that returns the number of modules contained in a system architecture (for a formal definition refer to Appendix A).

Axiom 12:

$$\exists S_1,S_2:\text{graph} \cdot \#\text{ds}(S_1)>\#\text{ds}(S_2) \wedge |S_1|<|S_2|$$

where #DS is a function that returns the number of global data structures contained in a system architecture (for a formal definition refer to Appendix A).

Axiom 13:

$$\exists S_1,S_2:\text{graph} \cdot \#\text{flows}(S_1)>\#\text{flows}(S_2) \wedge |S_1|<|S_2|$$

where #DS is a function that returns the number of local and global information flows contained within a system architecture (for a formal definition refer to Appendix A).

Axiom 14: One problem we noted in our critique of Henry and Kafura's metric [Henr81a] was the way in which their model behaved with respect to component re-use. This leads to the requirement that a system re-using components must not have a greater IF4 measure than a similar system duplicating the component. Figure 5.5 depicts two contrasting, but functionally similar, structures that can be defined as:

S = concat_e(concat_n(concat_n(concat_n(R,module,A),module,B),module,C),A,C)

$\forall$ R:graph; A,B,C,D:node ·
is_a_module(A) $\wedge$ is_a_module(B) $\wedge$ is_a_module(C) $\wedge$ is_a_module(D) $\wedge$ C=D $\wedge$
|concat_e(S,B,C)| |concat_e(concat_n(S,module,D),C,D)|

Similarly, we have the inverse:

S = concat_e(concat_n(concat_n(concat_n(R,module,A),module,B),module,C),C,A)

$\forall$ R:graph; A,B,C,D:node ·
is_a_module(A) $\wedge$ is_a_module(B) $\wedge$ is_a_module(C) $\wedge$ is_a_module(D) $\wedge$ C=D $\wedge$
|concat_e(S,C,B)| |concat_e(concat_n(S,module,D),D,C)|

In the next section we will assess the extent to which our model conforms to these axioms.

5.3 Theoretical Model Behaviour

The necessary groundwork has now been laid to commence a theoretical evaluation of the model of software design. Theoretical evaluation precedes empirical evaluation because convincing empirical investigations are usually lengthy and energy consuming enterprises. It is therefore appropriate to satisfy oneself that the model is internally consistent and satisfies the various criteria that are set as axioms. There is no suggestion, though, that theoretical analysis should replace empirical evaluation.

First, we will consider those properties that are fundamental to all measurement, as described in the previous chapter.

Axiom 1: It must be possible to describe, even if not formally, the rules governing the measurement. This is satisfied by the algebraic definition of IF4[12].

Axiom 2: The measure must generate at least two equivalence classes.

Here it is necessary to demonstrate that two system architectures exist, such that when the operation IF4 is applied they yield different results. The simplest structure is of course the null, or empty structure.

if4(new) = if4_int(new,new)	(Eqn. 15)[13]
if4_int(new,new) = 0	(Eqn. 16)

It now only remains to show that an architecture exists for which IF4 does not return zero. To do this we will build a system with two modules, A and B, and information flows between them (see Figure 5.6).

[12] Furthermore this definition has been shown to be fully operational by transforming the algebra into OBJ and executing the program.

[13] The numbers refer to the rewrite equation numbers given in the complete model specification in Appendix A.

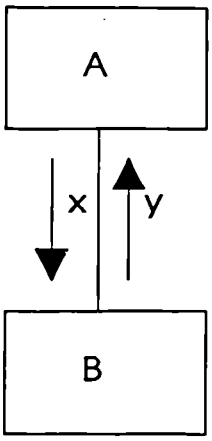


Figure 5.6: Example system design

The necessary constructors are:

concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A)

so the expression to evaluate is:

if4(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B)),B,A)

= if4_int(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),
(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A))
(Eqn. 15)

By applying Eqn. 18 twice:

= if4_int(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),
(concat_n(concat_n(new,module,A),module,B)) (Eqn. 18)

Instantiating into Eqn. 17 t_1 = 'module' gives:

= if4_int(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),
if4m (sqr (concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),B)) +
(concat_n(new,module,A))) (Eqn. 17)

Instantiating into Eqn. 19 IS_A_MODULE will return true for B yielding:

= if4_int(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),
sqr(
((fan_in_l(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),B) +
(fan_in_g(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),B)) *
((fan_out_l(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),B) +
(fan_out_g(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),B)) +
(concat_n(new,module,A)))) (Eqn. 19)

Evaluating the FAN_IN_G and FAN_OUT_G terms by applying Eqns. 26-31 we obtain zero in each case, since there is no node for which IS_A_DS returns true and therefore the THEN part of Eqns. 27 and 30 are never used when rewriting. Consequently the argument S_2 must reduce to the empty structure and by Eqns. 26 and 29 yield zero.

FAN_IN_L and FAN_OUT_L are a little more complex so we show each stage in the rewriting process.

fan_in_l(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),B)

By instantiating into Eqn. 21 $n_3 <> n_1$:

= fan_in_l(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B)
(Eqn. 21)

Again we apply Eqn. 21 but this time $n_3 = n_1$, thus:

= 1 + fan_in_l(concat_n(concat_n(new,module,A),module,B),B)
(Eqn. 21)
= 1 + fan_in_l(concat_n(new,module,A),B)
(Eqn. 22)
= 1 + fan_in_l(new),B)
(Eqn. 22)
= 1 + 0
(Eqn. 20)

Similarly, with FAN_OUT_L and Eqns. 23-25 we obtain:

```
fan_out_l(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B
,A),B)

=1
```

Substituting back into the previous expression we now have:

```
sqr( ((1+0)*(1+0)) ) +
if4_int(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A
),
(concat_n(new,module,A)) )
```

(Eqn. 17)

```
=1+
if4m(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A),
A) +
if4_int(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A
),new)
```

(Eqn. 17)

IF4m for module A will behave as for module B yielding

```
sqr( ((1+0) * (1+0)) )

=1
```

Our overall expression for IF4 now becomes:

```
=1+1+
if4_int(concat_e(concat_e(concat_n(concat_n(new,module,A),module,B),A,B),B,A
),new)

=1+1+0
```

(Eqn. 16)

As the rewrite sequence now terminates we have a result to applying the IF4 operation to the system of two, which is clearly distinct from zero so we are able to satisfy the second axiom.

As will be all too evident to the reader constructing this style of proof is both lengthy and tedious. For subsequent axioms only the outline of a proof is given and the reader referred to Appendix B.

Axiom 3: An equality relation is required. No proof is presented as the relation is axiomatic to our algebraic system, along with propositional logic and natural numbers.

Axiom 4: There must exist two or more structures that will be assigned to the same equivalence class. To satisfy this axiom, all that is required is to find two different design structures that yield the same measurement. Two such structures are shown in Figure 5.7. Because the proofs are both intuitively obvious and somewhat tedious they are omitted[14].

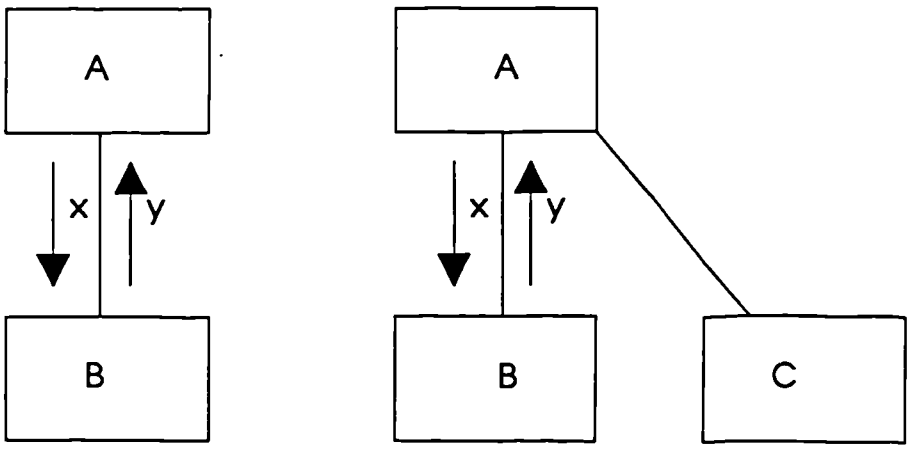


Figure 5.7: Two designs with equivalent IF4 values

Axiom 5: The metric must not produce anomalies (i.e. the metric must preserve empirical orderings). There are two aspects to this axiom. The first is the mapping of

[14] For the benefit of the reader who is both of a masochistic and sceptical disposition the proof is based upon the observation that the FAN_IN and FAN_OUT terms only potentially increment for CONCAT_E operations. Thus, no number of CONCAT_N operations (i.e. addition of modules or global data structures) will impact the FAN_IN or FAN_OUT terms and hence IF4m and hence IF4. Extending this argument it can be shown by induction that there in fact exist an infinite number of structures for each equivalence class.

the measurement function and the second is the definition of relations upon the measurement system. Unfortunately, as has been previously indicated, the first part of the axiom is an open problem due to the absence of generally agreed empirical relations. Hence the first part of the proof of this axiom cannot be discharged, however, since the measurement system is based upon the number system of natural numbers, the existence of equivalent relations in the empirical and measurement systems can be demonstrated[15]. For the proof refer to Krantz *et al* [Kran71].

Axiom 6: The Uniqueness Theorem must hold [Supp71] for all permissible transformations for the particular type of scale (i.e. there is a homomorphism between the transformed and the measurement structures). The underlying question is whether the measurement system is adequate for the type of measurement scale selected. For an ordinal scale the measurement structure must be order preserving for any monotonically increasing transformation function. This is trivially true [Stev59, Supp71, Kran71].

Axiom 7: Concatenating an additional module to the design structure cannot decrease the IF4 measure. The proof is the corollary for that of Axiom 4, in that if by adding nodes (modules or global data structures) additional members of the *same* equivalence class are generated, the IF4 measure cannot be decreased.

Axiom 8: Concatenating an additional global data structure to the design structure cannot decrease the IF4 measure. This holds for the same reason as Axiom 7.

Axiom 9: Concatenating an additional local information flow to the design structure must increase the IF4 measure. To show that the axiom holds for the model we argue inductively that it is true for the following two structures where the outside CONCAT_E represents the information flow being added to the design:

concat_e(concat_n(concat_n(new,module,a),module,b),a,b)
concat_e(concat_n(concat_n(N,module,a),module,b),a,b)

The first structure contains zero flows to which we add a local information flow. The second structure is the case where the structure N already contains n+1 flows where n is non-negative integer. A full discussion is given in Appendix B, however in brief, it is found that the axiom does not hold for the first structure above as it has an IF4 value of zero, both before and after an information flow is added. This is a significant result, because it indicates that Axiom 9 does not hold over our model since, adding the edge to the graph to represent the local information flow has not increased the IF4 measure; it remains at zero. The reason for this is not hard to find. The definition of IF4 involves multiplying the fan-in by the fan-out of each module. Should one term be zero this will

[15] In practice this axiom only becomes non-trivial when the measurement system is based upon such mathematical exotica as vectors, when it is not at all obvious that the Representation Theorem holds (e.g. Hansen's modification to the cyclomatic measure [Hans78]).

propagate through the metric definition giving zero overall[16]. The significance of this axiom violation will be discussed more fully at the end of the section.

Axiom 10: Concatenating an additional global information flow to the design structure must increase the IF4 measure. This axiom can be violated by presenting the same type of example as for the previous axiom, where adding an additional flow to a module where either the fan_in or fan_out remains zero will not increase the IF4 measure. Again the implications will be discussed more fully at the end of the section.

Axiom 11: Larger designs in terms of the number of modules may have lower IF4 measures than smaller designs. This axiom can easily be supported by reference to the two examples below and depicted in Figure 5.8:

concat_e(concat_e(concat_e(concat_n(concat_n(concat_n(new,module,a), module,b),module,c),module,d),a,b),c,d),b,d)

concat_e(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b,a)

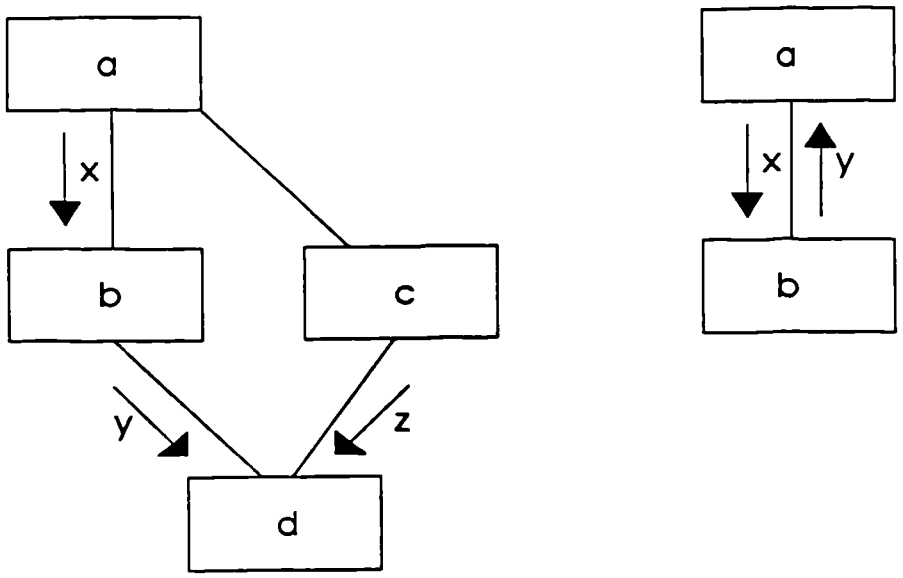


Figure 5.8: An example of where number of modules is not related to IF4 values

The first design has 4 modules whilst the second has only 2 modules yet it has an IF4 of 2 as opposed to zero.

Axiom 12: Larger designs in terms of the number of data structures may have lower IF4

[16] This difficulty is also to be encountered with Henry and Kafura's original information flow metric [KafH81].

measures than smaller designs. This axiom is demonstrated by reference to the two examples below and depicted in Figure 5.9:

concat_n(new,global_data_structure,x)

concat_e(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b,a)

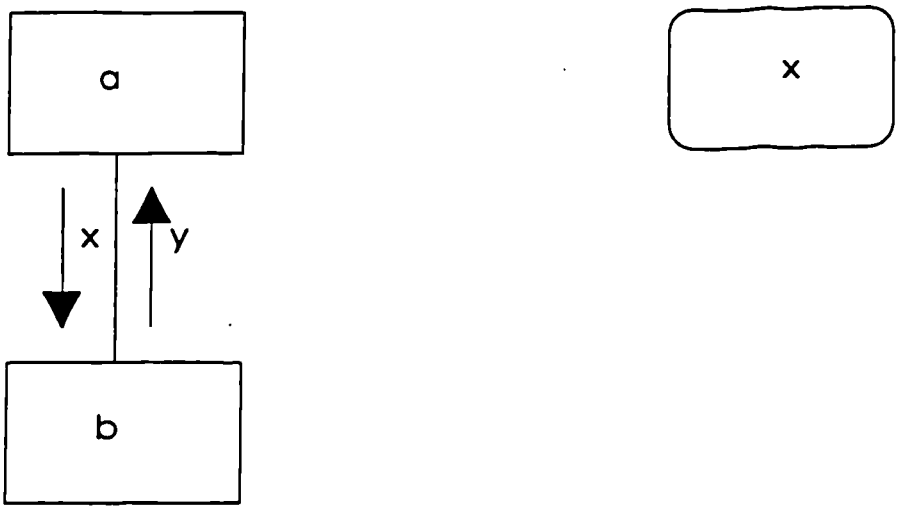


Figure 5.9: An example of where number of data structures is not related to IF4 values

The first design contains one data structure but has no flows and therefore IF4=0. The second design is the same as for the previous axiom, has no data structures yet the IF4=2.

Axiom 13: Larger designs in terms of the number of information flows (both local and global) may have lower IF4 measures than smaller designs. This axiom is demonstrated by reference to the two examples used in the discussion relating to Axiom 11. The first design contains three flows, whilst the second design has two flows yet the IF4 measure for the second example is the greater of the two. There are two reasons why this can occur. First the impact of a zero term in a module fan_in or out, as already illustrated with respect to Axiom 9. This is the factor at work in this case. Second the quadratic term will penalise distributions of flows that exhibit clustering tendencies.

Axiom 14: Module re-use must not be penalised in comparison to component duplication. Clearly, the issues at stake are the number of re-uses and the flows from the module independent of the re-use interface (i.e. all global flows and local flows to, or from sub-ordinate modules).

Unfortunately, as with the original Henry and Kafura metric [Henr79], this axiom does not hold for all cases. This may be demonstrated by a simple example. Referring back to Figure 5.5, the two architectures described differ only in that the first architecture re-uses module C, that is both modules A and B invoke module B, however, the in the second architecture module B uses a duplicate of module C, that is module D. The following table shows the IF4 values by module to arrive at an overall comparison of the two architectures.

Architecture 1				Architecture 2			
Module	Fan_in	Fan_out	$IF4_m$	Module	Fan_in	Fan_out	$IF4_m$
A	1	1	1	A	1	1	1
B	1	1	1	B	1	1	1
C	2	2	16	C	1	1	1
				D	1	1	1
			IF4 = 18				IF4 = 4

Table 5.1: Comparison of re-use strategies between two architectures

From the above, it can be clearly seen that the architecture that re-uses a module has a higher overall IF4 value than the architecture that duplicates a module. Not only does this violate generally accepted good software engineering practice but it also violates our axiom. More serious still, is the implication that the model of system architecture and information flow is inadequate because it does not properly capture the concept of separate instantiations of the same module that occur whenever a module is invoked from different parts of a system architecture. In particular, it suggests that our definition of information flow is insufficiently broad in scope to deal with module re-use. This is an important finding because it requires us to rethink certain aspects of the model that underlies the metric.

Three potential problems have been identified from the theoretical analysis of the IF4 metric. First, the introduction of an additional local information into a design does not always result in an increase in the metric value. The reason for this being the effect of zero flows into or out of a module. The second problem is likewise: that is the addition of a global flow does not always lead to an increase in IF4. The third problem, is possibly more serious in that software re-use is penalised as compared with module duplication. The reason for this is that the metric does not distinguish between memoryless or deterministic modules and those whose effect depends upon the previous

calling history. Some suggested solutions to these problems, will be presented in the concluding section of this chapter.

5.4 An Empirical Analysis

The next step was to validate the model of information flow against the stated goal (i.e. to aid the designer select appropriate architectures to maximise the software quality factors of implementability, reliability and maintainability). To do this, two empirical studies were carried out.

The first study addressed the software quality factor of development effort. Data was used from 13 software teams, where each team comprised three or four second year students from the BSc Computer Science course at Wolverhampton Polytechnic. The students were unaware that an experiment was being conducted. Each team implemented the same problem, thus facilitating comparison between systems. Students were allocated to teams in such a fashion as to minimise differences in ability and background and this was accomplished by examining past grades in software courses coupled with the judgment of tutors.

Each team was required to produce an adventure game shell which could be customised by the players of the game to meet their own requirements. Despite implementing the same specification, systems varied in size from 14 to 33 modules, and 313 to 983 of executable lines of code (ELOC) in Pascal. Development effort was recorded by monitoring computer connect time and by requesting students to submit a record of effort expended on the software development. Unfortunately, cross checking revealed the manual records to be extremely unreliable, for instance in several cases computer connect time exceeded *total* development reported manually. They were therefore discarded.

Error data was also collected, but was found to be too sparse to permit meaningful analysis. For the majority of systems, no errors were detected. This was probably the result of relatively small scale systems; none exceeded 1000 lines of executable code. Furthermore, the standard testing that each system was subjected to may have been less demanding than if the software had been used in a "live" environment.

Although, not initially intended, it was necessary to treat the results as weak orders (i.e. place in rank order only), due to the highly skewed distributions of the metrics. As a consequence, all correlation coefficients given are non-parametric Spearman values. Table 5.2 presents the cross correlations.

Design size, is defined as the number of modules, number of information flows and the number of information flows normalised by the number of modules. In addition, we also used ELOC, although to constitute a design metric this would have to be estimated.

Since ELOC[17] was intended as a control the assumption of perfect estimating did not seem unreasonable.

	DEV	IF4	FLOWS	MODS
IF4	0.797			
FLOWS	-0.389	-0.508		
MODS	-0.190	-0.229	0.268	
ELOC	-0.217	-0.196	0.287	0.646

Table 5.2: Cross correlations for adventure game study

The Spearman correlation of r=0.797 between development time and IF4 was statistically significant, there being less than a 1% chance of such a correlation having occurred by chance. This relationship is shown in Figure 5.10 as a scatter diagram.

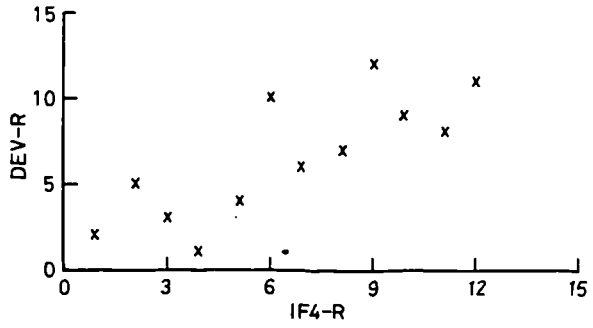


Figure 5.10: Scatter plot of IF4 versus connect time

By contrast, the size measures have weak or no correlation with development time. Parenthetically we note that the original Henry and Kafura measure had a correlation of r=0.434 with development time, which is statistically insignificant (there being more than 5% chance of this being a random occurrence). For a more detailed account of

[17] ELOC was used as it smoothed out size discrepancies due only to code layout. In general ELOC was found to be approximately 50% of LOC.

the study and analysis of the results the reader is referred to [Shep88] or [Shep90a].

Size factors do not seem to be significant determinants of development effort but, in contrast some structural factors are highly related, in particular IF4. Whether these findings would translate to systems with larger variations in size is unclear. In this study the largest system was three times greater than the smallest. However, if as stated in our goal, the measures are to intended to help the designer select between alternative architectures for the *same* problem it is unlikely that there will be a very large variation in size, so this is not necessarily a great stumbling block.

Where size seems to be of more importance is at an individual component or intra-system level. One system was discarded from the study because of an exceptionally large component and another had to be re-examined due to a "super" data structure. This is a disadvantage inherent in a uni-dimensional model, where trade-offs between module size and inter-modular information flows are ignored[18]. Indeed, taking this point to an extreme, one could construct an "optimal" system architecture comprising of a single module, where all the information flows and interface complexity would be subsumed within the module boundary. However, our understanding of software engineering principles suggests that this would not be good practice!

So far our discussion has centred on the information flow measure at a system wide level, but it can be obtained on a module by module basis ($IF4_m$). This is a potentially useful means of identifying problem areas *within* a design particularly when coupled with outlier analysis [Shep89a]. Thus, the software engineer is aided in the generation of new designs.

A second empirical study was based upon a realtime, aerospace application developed by Lucas Aerospace Ltd.. In this study we examined a system that comprised of 89 separate modules from a realtime control system, to consider the software quality factor of maintainability.

The four members of staff most closely associated with maintenance work on this project independently classified each module into one of four categories according to the perceived complexity of carrying out a maintenance task on that module. A score of one indicated a very simple module, whilst a score of four indicated a highly complex module. This procedure was adopted, since it represented the fastest method of obtaining a measure of maintenance problems, and because alternative documentary evidence concerning maintenance costs and traceability to specific modules was unavailable.

---

[18] A system architecture comprising a single module will always have zero information flows, regardless of module size.

Despite initial reservations about subjective evaluations, there was found to be a strong correspondence between the four classifications. The judgment discrepancies are summarised as follows:

max. - min.	% of judgments
0	10%
1	63%
2	27%
3	0%

Table 5.3: An analysis of maintenance judgments

It is noteworthy that for almost 75% of the judgments there was either no difference or only a difference of one between the highest and lowest scores. Also, in no case was a module judged to be a very simple module to change by one member of the maintenance team and highly complex by another member; such a situation would have lead to a discrepancy of three. To obtain an overall picture of maintenance difficulty for each module, the individual subjective judgments were summed to give a total possible score ranging from 4 (i.e trivial) to 16 (i.e. highly complex). Actual scores ranged from 4 to 15 with a mean value of 8.7.

The $IF4_m$ metric varied from a minimum score of zero to a maximum of 2,924,100, but with a median value of 196 thereby the highly skewed nature of the distribution. A Spearman correlation test was carried out between perceived maintenance complexity and the information flow metric. A correlation coefficient of r=0.70 was obtained which was statistically significant (i.e. less than a 1% chance of occurring by chance). The relationship between maintenance complexity and the information flow metric is also depicted graphically in Figure 5.11. In order to provide some basis for comparison a correlation coefficient of r=0.72 was obtained between the maintenance scores and LOC, and the cross correlation between LOC and IF4 was r=0.49 suggesting that IF4 is more than a mere proxy for size as captured by LOC. Although the traditional LOC measure slightly outperforms IF4, one must remember that IF4 is available much earlier in the development and maintenance process. Lastly, this study suggests that there is potential merit in combining an architectrure metric, such as IF4 and a size metric such as LOC, in order to provide better coverage of troublesome modules from a maintenance perspective.

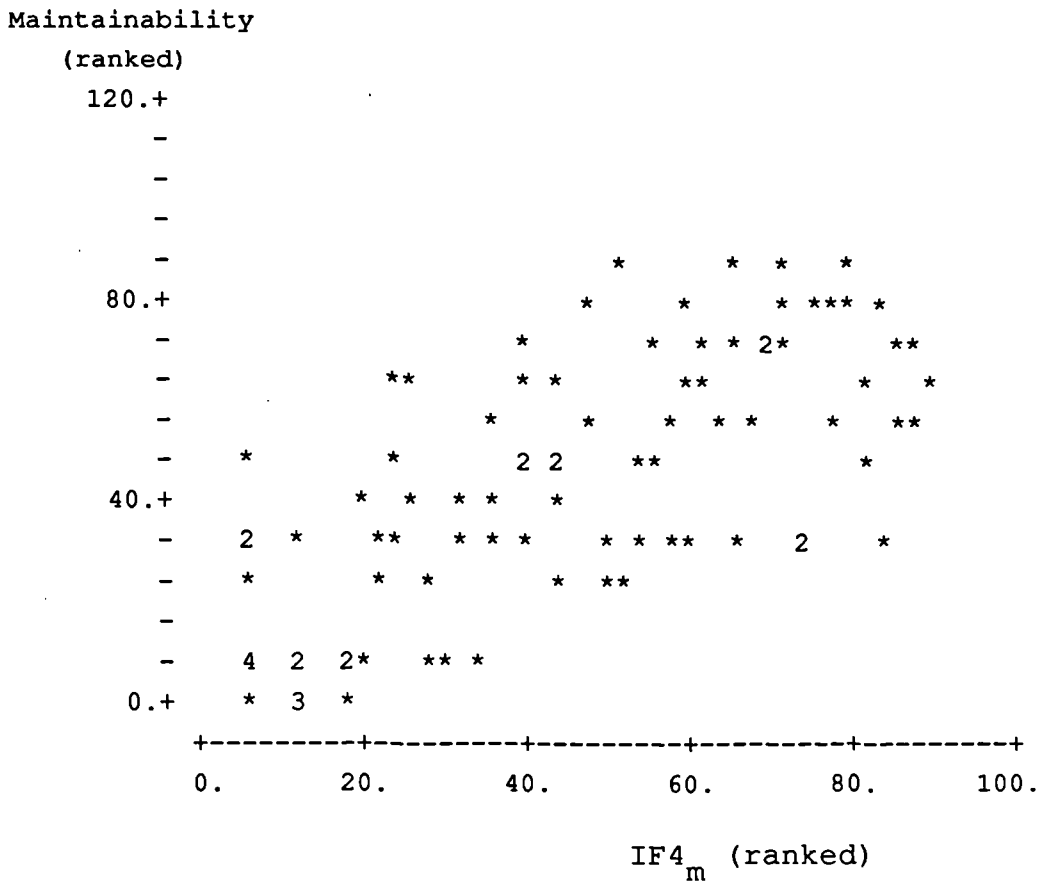


Figure 5.11: Module maintainability vs. the design metric $IF4_m$

The effectiveness of the design metric, as a means of identifying problem modules, can be demonstrated as follows. If problem modules are defined to be those that fall into the upper quartile of values for the complexity judgments, then these can be compared with those modules that fall into the upper quartile of the design metric. This indicates the predictive power of the design metric.

	HI-METRIC	LO-METRIC	TOTAL
HI-MAINTENANCE	14	9	23
LO-MAINTENANCE	9	57	66
TOTAL	23	66	89

Table 5.4: The predictive power of the $IF4_m$ metric

In this instance the metric would of identified 14 out of 23 of the most troublesome modules, a yield of about 64%. On the other hand, the metric highlights 9 modules as being complex when in fact they have not proved to be so, an error rate of 36%. By contrast, if one were selecting modules at random one would expect a yield of 25% and an error rate of 75%.

There is an obvious weaknesses in this investigation. This is the reliance upon the subjective judgment of a small number of individuals. Further work is required to explore the relationship between perceived complexity from a maintenance change point of view and actual historical data, such as mean effort per change per module and probability of a module being impacted in any one maintenance change. This is illustrative of the iterative nature of any software engineering application of software metrics and modelling.

5.5 Evaluation of the Uni-dimensional Model

To summarise the position so far, three problems have been uncovered by the theoretical analysis of the model - two of them minor and one more significant. In addition the empirical studies have raised the problem of component size, and in particular module size, within a system architecture. Each of these areas will be dealt with in turn.

First, the comparatively minor issue *of zero terms propagating through the expression* to compute IF4. This has the consequence that additional local or global flows do not invariably increase the overall IF4 value. A simple solution to this problem is the addition of one to both the fan_in and fan_out terms as advocated in [Ince89a]. Thus the new definition of $IF4_m$ becomes:

$$IF4m = (1+fan\_in \,.\, 1+fan\_out)^2$$

The solution to the re-use problem is a little more complex. As has already been remarked upon, there is need to distinguish between module re-use when behaviour cannot be influenced by previous invocation from other modules and when this is not the case. The situations can be readily distinguished on the basis of classifying a module either as having a memory or as being memoryless. A module is memoryless if neither it, nor any of its subordinates both write to and read from a global data structure[19]. An example of both types of module is given in Figure 5.12, the first architecture illustrating a re-used module with memory.

---

[19] Strictly speaking even if these conditions pertain one can only state that there exists the possibility for modules with memory. However, this may only be resolved by dynamic analysis which is outside the ambit of the model for reasons that have been previously discussed.

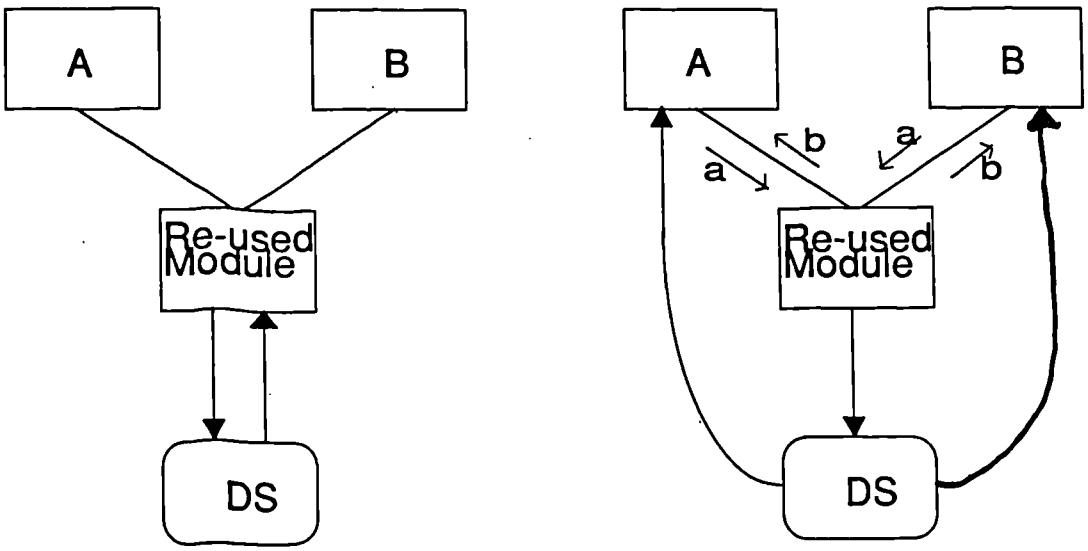


Figure 5.12: An example of a module with and without memory

where re-used modules are deemed to be memoryless their interface should only be counted once, rather than once per module invocation. The consequence of this is to be either neutral or to favour module re-use as required by Axiom 14. Returning to the example used to refute Axiom 14 in section 5.3 and applying the modified rules described above, a new set of metric values are obtained.

Architecture 1				Architecture 2			
Module	Fan_in	Fan_out	$IF4_m$	Module	Fan_in	Fan_out	$IF4_m$
A	1	1	1	A	1	1	1
B	1	1	1	B	1	1	1
C	1	1	1	C	1	1	1
				D	1	1	1
			IF4 = 3				IF4 = 4

Table 5.5: Comparison of re-use strategies between two architectures counting interfaces once

As may be seen from Table 5.5 the architecture that re-uses rather than duplicates a module now has the lower metric value[20].

[20]The concept of memory and memoryless modules may have a secondary application in that reused modules with memory are likely to be lacking in cohesion [Stev74] . This is significant since module cohesion

Turning to the issue of module size, raised by the empirical analysis, a solution is more difficult. The fundamental problem is that software architecture is rather complex to model using only a single dimension or measure. What is really required is a second measure, in order to capture the notion of module size, *in addition* to module interface complexity in terms of information flows. The next chapter goes onto explore enhancements to the model by means of introducing a second dimension.

To conclude then, the single dimensional model has some utility as demonstrated by the two empirical studies and has been shown to be significantly related to both the factors of development effort and maintainability. This is despite three theoretical problems that have been uncovered by the application of an axiomatic treatment of an algebraic or formal definition of the model[21].

---

has always proved more difficult to capture than module coupling, see for example the discussion in [Shep90f].

[21] A possible reason why the empirical analysis was broadly supportive despite the problems highlighted by the theoretical analysis was that none of the architectures significantly re-used modules, coupled with the fact that those designs that were bad were so bad that they tended to overwhelm more marginal features like the degree of module redundancy!

6. A MULTIDIMENSIONAL MODEL OF SOFTWARE DESIGN

'"Curioser and curioser", said Alice, who was so surprised, she quite forgot to speak good English.'

from 'Alice through the Looking Glass'

'In fact, the goal of the design process is never a "best" design. Instead it is an "adequate" design that satisfies the requirements and design goals and has a reasonably good structure.'

Barbara Liskov and Jon Guttag [Lisk86]

Synopsis

This chapter argues that if software engineering models are to be useful, then they will require more sophistication than can be achieved by a single input or dimension. In particular it is shown that some of the weaknesses in the model of system architecture described in the previous chapter may be overcome by extending the model to incorporate a notion of module, or design component size. This permits the exploration of trade-offs between inter and intra modular information flows. Existing measures of module size - available at design time - are briefly reviewed, but none found altogether satisfactory. An alternative approach is proposed, based upon the traceability of functional requirements from a specification onto a module hierarchy. This validated, according to both the theoretical and empirical criteria described earlier in this thesis, and found to yield useful results. The chapter then describes how using simple statistical techniques such as two-dimensional scatterplots and multi-variate outlier analysis an improved model of software design can be developed. The conclusion is drawn that, the techniques employed here enable more sophisticated and useful software models to be developed. However, this must be built upon the foundation of theoretically clean and empirically observed relationships. Otherwise our sophistication is entirely in vain.

6.1. Why multidimensional models?

This thesis has argued that there is considerable support - both theoretical and empirical - for the proposition that the structural metric IF4 can identify potentially problematic architectures and components, typically modules [Ince89, Shep90a, Shep90c]. Clearly it is of great value to the designer to be able to obtain feedback on design decisions, prior to ossifying them into code. There is a danger, however, that lurks behind concentration upon structural measures; that is the possibility of reducing structural complexity merely by adopting very large components.

Unfortunately a single measure of structure cannot be sensitive to this process. Basili and Rombach [BasR88] make a similar point, when they argue that most aspects of software development processes and products are too complex to be adequately captured by a single metric. Consequently, useful models will normally incorporate more than one dimension. Nevertheless, we still wish to remain subject to the restriction imposed by the objective of providing guidance for the software designer, that all measures must be available at design time.

The remainder of this chapter investigates the various existing metrics for system component size: that is the full Henry and Kafura [Henr81a] information flow metric, the Card and Agresti intra-modular metric [Card88], the work by Samson *et al* on formal specification [Sams87], the deMarco 'Bang' metric [deMa82] and finally Albrecht's function points [Albr83]. All of the above are found wanting in at least one respect so, an alternative metric, module "work" is proposed, based upon the traceability of functional requirements from a specification to design. The behavioural properties of the underlying model are analysed by means of the flexible axiomatic approach developed in the chapter four. This measure is then applied to empirical data and found to yield useful results and to perform more reliably than the other metrics evaluated. It is suggested that the multidimensional model is more effective at identifying problem modules than any single metric. We conclude with some remarks concerning the role of multidimensional models in software engineering processes.

6.2 Measuring module size

6.2.1 Existing module size metrics

Structural complexity is merely one dimension, albeit an important one, of software design. As we have already remarked it disregards component size, concentrating

purely upon the manner in which components are linked. Such a view of design is potentially flawed, as a designer can exploit the relationship between structure and component size by exchanging one for the other. Some notion of component size is, therefore, an important addition to a structure metric.

The previous chapter - and also [Ince89a, Shep90a, Shep90b] - have highlighted some of the problems of only measuring architectural complexity. In this empirical study of 13 different architectures that implement a single specification, we found two aberrant systems that deviated from a generally well defined relationship between structural complexity (measured as the information flows between system components) and development effort. Closer examination revealed that both the two architectures involved, exploited a trade-off between structural complexity and component size. In one case a single module comprised 43% of the resultant software system size, measured as executable lines of code (ELOC) and was more than 600% over the mean module size. As a consequence, the structure metric under-estimated development time to a significant degree. At a theoretical extreme one could minimise structural complexity by adopting an architecture containing a single component!

The problem then becomes one of how can we measure system component size at this early stage in the software development process? And having done so, how can this measure be combined with the structure measure? The following sections attempt to provide some answers.

Several existing design metrics have attempted to combine a measure of both system structure and component size. The original information flow metric of Henry and Kafura [Henr81a, Henr84] attempted to do this by multiplying the information flow or structural complexity of a module by its size, measured as estimated length in lines of code (LOC). This gives rise to two problems. First, LOC is not available at design time. Second, empirical work has been very equivocal as to the merits of introducing the LOC term to the equation [Henr81a, KafC85, Romb87]. One can draw two possible inferences from this second problem. Either LOC is a poor measure of component size, or multiplication is an inappropriate method of combining the two dimensions of design as they are not completely orthogonal. We are inclined to the view that both inferences contain a degree of truth.

Card and Agresti [Card88] have attempted to tackle the problem by defining separate metrics for inter-modular (structural) complexity and intra-modular complexity (design component size). The inter-modular measure is based on the square of the count of module calls made by each module. The intra-modular measure is the workload of a module, which, from the classical module-as-a-function perspective can be interpreted as the mapping from the set of inputs to the set of outputs. The amount of work to be

performed can be characterised as the cardinality of the two sets, divided by the number of subordinate modules plus one, as this indicates the extent to which work is shared between modules. This leads to the size measure:

$$v / (f+1) \qquad \text{Eqn. 6.1}$$

where v = no. of inputs + outputs
f = no. of subordinate modules

Unfortunately the metric is still vulnerable to the use of very large modules. Since workload is defined as the movement of data across the module boundary, it will fail to capture any work that is completely internal to a module. The metric also assumes that data and modules are of uniform size. Another difficulty is that for many applications workload includes device management. Evidently the definition of workload, or inputs and outputs, must be broadened to include devices and global data structures. Even with these changes it is disappointing to report that it performed poorly on our empirical data. The measure failed to identify the two rogue modules and produced no meaningful correlation with module size in terms of LOC, decision counts or the number of variables used (see Table 6.1 in section 6.). There are two possible explanations. First, increasing the number of subordinates may increase the workload of a module as there is work involved in scheduling the module calls. Second, it is vulnerable to the practice of splitting work across two or more levels of a module hierarchy, thus a large number of subordinates might indicate a large number of partial functions to be added to the calling module's workload.

Samson *et al* [Sams87] describe an attempt to obtain useful measures from an OBJ specification. Unfortunately the use of algebraic specifications is hardly a commonplace industrial activity, nor is the empirical support overwhelming. Furthermore, one has doubts concerning the validity of the underlying model, which suggests that the number of axioms within the specification will be equivalent to the cyclomatic number of the code implementation.

Function points [Albr83] represent another approach to specification measurement, where the aim is to try to quantify task size using inputs such as the number of queries, system inputs and outputs. This measure is more flexible in the type of specification document it may be derived from, but treats a system as a single entity, whereas our interest is components and component size. By contrast, deMarco's, some what curiously termed, Bang metric [deMa82] can be applied at a component level but requires data flow diagram and entity-relationship diagrams as input. Although these notations are more widespread than formal specification, they still represent a minority

practice, coupled with the fact that the author is unaware of any empirical support for deMarco's ideas.

6.2.2 The "work" metric

None of these specification measures fully meet our requirements of relating functional requirements to modules in the design, so as to give an indication of size. We have, therefore, developed our own approach based upon the concept of module "work". From any specification document it is possible to isolate the functional requirements for a proposed system. Note that functional requirements (e.g. a report must be produced) are distinct from constraints (e.g. the mean response time must be less than 5 seconds) as the latter typically impact most, if not all of a system, whereas a functional requirement may be mapped onto one, or a small number of components.

Functional requirements can be constructed into a hierarchy as exemplified by the Automated Requirements Traceability System at Lockheed [Dorf84]. A requirement is either primitive, in that it is not refined into sub-requirements, or is composite. For example, HANDLE_INPUT could be a composite requirement comprising of FETCH_INPUT and VALIDATE_INPUT. A composite requirement is regarded as being satisfied when all its sub-requirements are satisfied, so HANDLE_INPUT is satisfied when both FETCH_INPUT and VALIDATE_INPUT are satisfied. In this manner, requirement satisfaction is inherited up the hierarchy until the top level SYSTEM when, by definition, all requirements are satisfied.

For each module we specify the set of primitive requirements satisfied by a module i, P_i. This set may have zero or more members, though the empty set for childless modules should be regarded with some suspicion, as this suggests either that the module has no purpose, (improbable) or that a functional requirement has not been identified or allocated to that module (more probable). Another possibility is that a primitive requirement maps to more than one module, indicating either a split function or redundancy. Finally, a module may only partially satisfy a primitive requirement. In such a circumstance the primitive is treated as a composite and new primitives introduced. Where many primitive requirements map to many modules, this is indicative of the requirements hierarchy being insufficiently detailed. Alternatively, the design structure exhibits problems of module coupling [Stev74, Your79]. The converse is, of course, that many primitive requirements mapping to a single module is suggestive of the module hierarchy being insufficiently detailed. This is also the first intimation that the designer may have traded structural complexity for component size.

We must now consider the scheduling side of a module's workload. Scheduling work is required if one or more of three conditions are true for a module:

- two or more primitive requirements are satisfied;
- one primitive requirement is satisfied combined with one delegated requirement;
- two or more delegated requirements are combined.

A requirement is delegated if it is satisfied by a subordinate module.

We define the workload for module i, $work_i$ as:

$$r_i + \alpha (s_i) \qquad \text{Eqn. 6.2}$$

$$r_i = \#(P_i) \qquad \text{Eqn. 6.3}$$

$$s_i = \#(Rmax_i) + \#(Rmin_i) \qquad \text{Eqn. 6.4}$$

where:

$\#$ = set cardinality;

$0 < \alpha$ the coefficient indicating the relative contributions to workload of scheduling and requirement satisfaction;

P_i = set of primitive requirements satisfied by module i;

$Rmax_i$ = the set of all requirements satisfied or inherited by the ith module;

$Rmin_i$ = the minimal set of requirements after all possible substitutions of more primitive requirements from $Rmax_i$ by higher order or more composite requirements (e.g. $Rmax_i$ = {a,b,c} and D = {a,b} yields $Rmin_i$ = {D,c}).

The steps involved in calculating the work for each module may be summarised as the following steps (see [Shep89d, Shep90b] for a more detailed treatment).

1. Construct, from the system specification, a functional requirements hierarchy.

2. Construct from the design documentation a module calling hierarchy.

3. For each module, determine which primitive requirements are satisfied either in full or in part, thereby establishing P_i.

4. For each module calculate $Rmax_i$, that is P_i plus any inherited requirements.

5. For each module calculate $Rmin_i$, that is $Rmax_i$ with all more primitive requirement groupings replaced by higher level requirements wherever possible. Thus:
$\#(Rmax_i) \geq \#(Rmin_i)$

6. Calculate the scheduling work of each module s_i as:
$\#(Rmax_i) + \#(Rmin_i)$

7. Calculate $work_i$ for each module as:
$r_i + \alpha (s_i)$

A judgment has to be made concerning the value of the coefficient α. This will depend very much on the type of application and development environment. As an initial approximation, and no more than that we took the view that the ratio of work involved in scheduling in relation to carrying out a task was 1 : 3 and so assigned a value of 0.33 to the coefficient. This was derived as follows. First, we found that approximately 35-40% of the code appeared to be devoted to scheduling work, as opposed to satisfying primitive requirements. We also observed that ratio of primitive tasks to scheduling tasks was in each case close to 1:2. This suggests:

$$(0.375/2) : (0.625/1)$$

$$\approx 1 : 3$$

Therefore:

$$\alpha \approx 0.33$$

Further empirical work is required to substantiate this approach. Indeed it would be surprising if α does not vary for different environments and applications, since the proportion and type of scheduling work differs considerably between say, commercial dp systems and time critical, embedded control systems.

6.3 Validation of the "work" Metric

So far we have a putative model for module size at design time. The next step is to assess the validity of the model. This a accomplished by a combination of theoretical techniques based upon an axiomatic statement of the desired model properties and proofs that the axioms are satisfied. This is complemented by an empirical validation to

increase our confidence that the model does indeed capture size aspects of modules identified within a software design.

6.3.1 Theoretical validation

So far, our model of module size has been presented in a semi-formal manner. However, in order to compare desired model characteristics with actual behaviour, using the flexible axiomatic approach, a more rigourous set of definitions are required, based upon an equational rewrite system [Gutt77]. This is similar to the treatment given to the structural design metric IF4 in the previous chapter.

6.3.1.1 An algebraic specification of the "work" metric model

As in all algebraic specifications, the starting point is identification of the constructor operations [Geha82]. In this instance there are two groups of constructors: those concerned with the functional requirements hierarchy derived from a specification; and those concerned with the module calling hierarchy extracted from the system architecture, or high level design. These operations have the following signatures:

newspec:	$\rightarrow$ spec
addr: req $\times$ req $\times$ spec	$\rightarrow$ spec
newdes:	$\rightarrow$ des
addm: mod $\times$ mod $\times$ des	$\rightarrow$ des
sat: mod $\times$ req $\times$ des	$\rightarrow$ des

The *newspec* and *newdes* operations create new hierarchies and are necessary in order to introduce determinism into the specification. *Addr* and *Addm* add requirements and modules respectively to an existing hierarchy. Since a hierarchy is structure that requires all its elements to be related, the second argument is the parent of the requirement or module being added[1]. The constructor *sat* is most important because it represents the mapping or the linkage between the two hierarchies, whereby a module from the design satisfies a functional requirement from the specification. There is no reason, in principle, why one module may not satisfy zero, one or many requirements, nor why one requirement should not be implemented by one or more modules[2]. In the

[1]This style of constructor presents one difficulty in the form of the root node, or top level element. It is resolved by the use of a special null parent value represented by the empty string. The justification for this slight unpleasantness is that it greatly simplifies the algebraic specification by reducing the number of constructors from six to four.

interests of brevity and clarity, it is assumed that we are only dealing with well formed structures, that is a subset of all structures that it is possible to describe using the constructor operations. Specifically, it is assumed that the design fully satisfies its specification, that the hierarchies contain no recursive structures and that all requirement and module names are unique. This avoids the need to introduce internal operations and their attendant complications, although the algebraic technique can handle such situations as evidenced by the specification of the IF4 metric given in the previous chapter.

The description of the model, from the last section has the fundamental concept of a requirement comprising of zero or more sub-requirements. This is captured by an operation that yields the set of sub-requirements that are immediately subordinate to a given requirement. Thus we have,

comprises: req × spec → reqset

where *reqset* is a type, set of requirements. Note, that as a form of shorthand, this model specification assumes that the type set and its basic operations, such as membership, union and subset are previously defined. Again, this is in the interests of brevity. The equations to define the *comprises* operation are based upon the constructors operations already described.

1. comprises(r1,newspec)	=	{}

2. comprises(r1,addr(r2,r3,S))	=	IF r1==r3 THEN
		comprises(r1,S) ∪ {r2}
		ELSE
		comprises(r1,S)

Next we define the operator *exists?* which tests to see if a given requirement is contained within the requirements hierarchy.

exists?: req × spec → boolean

3. exists?(r1,newspec)	=	FALSE
4. exists?(r1,addr(r2,r3,S))	=	IF r1==r2 THEN TRUE ELSE exists?(r1,S)

---

[2]The pathological case of a design failing to fully satisfy its specification - that is a requirement being implemented by zero modules - is excluded from the following discussion on the grounds that in such circumstances the model will not yield meaningful metrics.

It is also useful to define requirements as either primitive, that is without sub-requirements or as composite. An operation prim? returns TRUE if a given requirement is primitive.

prim?: req × spec → boolean ∪ {ERROR}

5. prim?(r1,newspec)	=	{ERROR}
6. prim?(r1,addr(r2,r3,S))	=	IF ¬ exists?(r1,S) THEN {ERROR} ELSE IF comprises(r1,S)=={} THEN TRUE ELSE FALSE

This definition merely states that for any non-empty specification structure, any requirement that has no sub-requirements is deemed to be a primitive. The operation is meaningless if the specification structure is empty or does not contain the requirement r1, and this indicated by the special result {ERROR} which terminates the equation rewriting.

Next we turn to the module calling hierarchy and define the operation which returns the set of all modules directly called by a given module within a system architecture.

calls: mod × des → modset

7. calls(m1,newdes)	=	{}
8. calls(m1,addm(m2,m3,D))	=	IF m1==m3 THEN calls(m1,D)∪{m2} ELSE calls(m1,D)
9. calls(m1,sat(m2,r1,D))	=	calls(m1,D)

Equation nine essentially states that the operation *sat* has no effect upon the module calling structure. The specification has been slightly simplified by ignoring the possibility of module that does not exist within a non-empty calling hierarchy. In such a circumstance an empty set would be returned. Note that this algebraic specification makes it clear that any subordinate module that is invoked more than once will only be counted as one call - since a set may not have duplicate members. However, a module may be called by more than one set, in which case it will be a member of more than modset.

The next operation *descend?* is related to *calls* in that it tests whether module m1 is a descendant of module m2, where descendant means a calling path of arbitrary length[3]

from the latter to the former module. An additional, internal operation is present in order to create a second copy of the argument *des* which is passed down the recursion, as a consequence of the stateless style of specification. In other words it is an internal artifact of the algebraic specification and is not, therefore, a direct external characteristic of the model.

descend?: mod × mod × des → boolean
desc?: mod × mod × des × des → boolean

10. descend?(m1,m2,D) = desc?(m1,m2,D,D)

11. desc?(m1,m2,newdes,D) = FALSE
12. desc?(m1,m2,addm(m3,m4,D1),D) = IF ((m1==m3)∧((m2==m4)) ∨
(desc?(m2,m4,D1,D))∧(m4<>''))
THEN TRUE
ELSE desc?(m1,m2,D1,D)
13. desc?(m1,m2,sat(m3,r1,D1),D) = desc?(m1,m2,D1,D)

Now we define the set of primitive requirements directly satisfied by a given module, in terms of the operation *Pwork*.

Pwork: mod × des × spec → reqset

14. Pwork(m1,newdes,S) = {}
15. Pwork(m1,D,newspec) = {}
16. Pwork(m1,sat(m2,r1,D),S) = IF (m1==m2) ∧ comprises(r1,S)={}
THEN Pwork(m1,D,S) ∪ {r1}
ELSE Pwork(m1,D,S)

17. Pwork(m1,addm(m2,m3,D),S) = *Pwork(m1,D,S)*

The next concept from the model that requires formalising is that of inheritance, whereby a module inherits the requirement satisfactions of all its descendants, not merely those modules that it directly calls. This is captured by the operation *inherits* and the internal operation *inherit*.

inherits: mod × des → reqset
inherit: mod × des × des → reqset

---

[3] *Desc?* is distinct from *calls* in that calls returns only those modules that are directly called by the specified module, that is a calling path length of one.

18. inherits(m1,D) = inherit(m1,D,D)

19. inherit(m1,newdes,D) = {}
20. inherit(m1,addm(m2,m3,D1),D) = inherit(m1,D1,D)
21. inherit(m1,sat(m2,r1,D1),D) = IF desc?(m2,m1,D)
THEN inherit(m1,D1,D) ∪ {r1}
ELSE inherit(m1,D1,D)

Note that this definition means that a module does not inherit the requirements that it directly satisfies, and consequently leaf modules - those without descendants, unlike the meek - do not inherit[4].

The next step is to define the process of factoring out groups of requirements and substituting them by more abstract requirements.

abs: reqset × spec → reqset
abstract: reqset × spec × spec → reqset

22. abs(R,S) = abstract(R,S,S)

23. abstract({},S1,S) = {}
24. abstract(R,newspec,S) = R
25. abstract(R,addr(r1,r2,S1),S) = IF comprises(r1,S)<>{}
∧ (comprises(r1,S) ⊆ R)
THEN abstract(R,S1,S)∪R∪{r1}
- comprises(r1,S)
ELSE abstract(R,S1,S)

Note that the minus sign indicates set difference and a hash symbol should be read as set cardinality.

Finally, we proceed to define the two outputs from the model, $work_i$ and *work*, where $work_i$ is a measure of workload or size for the m^{th} module and *work* is a measure for the entire system.

---

[4]This is in fact a curious inverse of the human process and doubtless the consequence of injudicious choice of analogy during the informal description of the model!

$work_i$: mod × spec × des → real
work: spec × des → real
wk: spec × des × des → real

26. $work_i$(m1,newspec,D) = 0
27. $work_i$(m1,S,newdes) = 0
28. $work_i$(m1,S,D) = #(Pwork(m1,D,S)) + α(#(abs(inherits(m1,D) ∪ Pwork(m1,D,S),S)) + #(inherits(m1,D) ∪ Pwork(m1,D,S)))

29. work(S,D) = wk(S,D,D)

30. wk(S,newdes,D) = 0
31. wk(S,addm(m1,m2,D1),D) = wk(S,D1,D) + $work_i$(m1,S,D)
32. wk(S,sat(m1,r1,D1),D) = wk(S,D1,D)

This concludes the formal definition of the model, which is presented in its entirety in Appendix C. The next step is to consider those properties that we wish to be true of the model and state these as axioms. We will then attempt to demonstrate whether the axioms hold for the model using a similar approach to that demonstrated for the structural model of design in the previous chapter.

6.3.1.2 Desired model behaviour: an axiomatic treatment

By applying the three tiered approach of the flexible axiomatic method we obtain those axioms that are fundamental to all measurement as described in the chapter four. Briefly these are:

Axiom 1: It must be possible to describe, even if not formally, the rules governing the measurement.

Axiom 2: The measure must generate at least two equivalence classes so that the metric is able to discriminate between software designs.

Axiom 3: An equality relation is required.

Axiom 4: There must exist two or more designs that will be assigned to the same equivalence class. This is a stricter form of Axiom 3. If this cannot be shown to be true, the metric will generate a unique value for each unique design - not a very useful property for a software metric!

Axiom 5: The metric must not produce anomalies (i.e. the metric must preserve empirical orderings), so if module A can be shown to be empirically larger than module B then this must be mirrored by the metric values for modules A and B[5]. This is known as the Representation Theorem [Kran71].

Axiom 6: The Uniqueness Theorem must hold [Supp71] for all permissible transformations for the particular scale type - in this instance ordinal - (i.e. there is a homomorphism between the transformed and the measurement structures).

Next are those axioms that are dependent upon the choice of measurement scale. In this instance the relationships identified within the model are concerned with input and output variable *ranks* which in turn suggests that weak ordering will be sufficient for our purposes. This implies ordinal measurement. The axioms are those of transitivity of the < and > relations and symmetry, reflexitivity and transitivity of the equivalence relation.

Lastly, there are those axioms that relate to the specific model underlying the measure in question. Again, it is possible to provide categories under which axioms may be selected. These are:

- resolution;
- empirically meaningless structures;
- model invariants.

No further axioms concerning measurement resolution are required beyond Axioms 1 and 2 which state that the measure must be capable of discrimination and Axiom 4 that states that there must exist at least two system architectures that will be assigned to the same equivalence class.

Concerning meaninglessness: this category of axioms will be avoided as a consequence of the decision to simplify the model by not explicitly defining illegal or meaningless structures (e.g. where the specification is not fully implemented by the design). Consequently, no axioms are offered.

---

[5]The nub of this assertion is *if* it can be empirically shown. A major limitation to the usefulness of this axiom is the difficulty of obtaining agreement on the empirical relational system. Frequently it is not obvious that A is larger than B, and much depends upon our intuitive understanding of module size. Accuracy of measurement is another problem area. The situation is even more ambiguous for such metaphysical commodities as complexity; perhaps one should argue for a homomorphism between a metaphysical relational system and the measurement system!

The third category of axioms are those properties specific to this model which we believe are of fundamental significance, and therefore must remain invariant. To elaborate.

Axiom 7: Adding an additional requirement to the system specification must increase the work metric.

$$\forall\ S{:}spec;\ D{:}des\ r{:}req;\ m{:}mod \cdot$$
$$SATSPEC(D,S) \Rightarrow work(S,D) < work(addr(S),sat(m,r,D))$$

This axiom requires a further operation for the algebraic definition of the model of software design size, which is required to test that a design satisfies a given specification. Note that in order to define the *SATSPEC* operation two internal operations are required, *SATSPEC'* and *SATREQ*.

satspec: des × spec → boolean
satspec': des × spec × spec → boolean
satreq: des × req → boolean

33. satreq(newdes,r1) = FALSE
34. satreq(addm(m1,m2,D),r1) = satreq(D,r1)
35. satreq(sat(m1,r2,D),r1) = IF r1==r2 THEN TRUE
    ELSE satreq(D,r1)

36. satspec(D,S) = satspec'(D,S,S)
37. satspec'(D,newspec,S) = TRUE
38. satspec'(newdes,addr(r1,r2,S1),S) = FALSE
39. satspec'(D,addr(r1,r2,S1),S) = IF prim?(r1,S) THEN
    IF satreq(r1,D) THEN
    satspec'(D,S1,S)
    ELSE
    FALSE
    ELSE satspec'(D,S1,S)

Axiom 8: Related to this axiom is that the metric $work_i$ must increase when the i^{th} module satisfies an additional requirement.

$$\forall\ i{:}mod;\ r_1, r_2{:}req;\ S{:}spec;\ D{:}des \cdot work_i(i,S,D) < work_i(i,adds(r1,r2,S),sat(i,r1,D))$$

Axiom 9: There must exist different designs that satisfy the same specification but which have different work metric values. In other words design size is not only a function of the specification but also the intrinsic organisation of the design. The consequence of this axiom is that for a given specification choice of architecture can influence size.

$$\exists\, D_1, D_2 : des;\ S : spec \cdot satspec(D_1, S) \wedge satspec(D_2, S) \Rightarrow work(S, D_1) <> work(S, D_2)$$

It is noteworthy that the model of design size has fewer axioms associated, than the model of design structure presented in the previous chapter. The likely explanation is that size is a somewhat simpler concept than that of structure and flows of information coupling modules together. Nevertheless, there remains the question of whether the axioms listed above are sufficient. To some extent the question is unanswerable, since the desired behaviour of the model is merely the collection of intuitions and hypotheses floating around within the head of its progenitor! The method of metric development outlined in this thesis does offer some safeguards, in the form of axioms that must be true of all metrics and categories for the generation of axioms specific to the model. For example, axioms may be required for the level of resolution of the metric. Consequently, one can have a reasonable degree of confidence that the axioms presented at least describe some minimal set of model behaviour characteristics.

6.3.1.3 Proofs of model axioms

The next step is to show that the nine axioms described in the previous section hold for the model of design size.

Axiom 1: It must be possible to describe, even if not formally, the rules governing the measurement. This is satisfied by the existence of an algebraic definition of the work metric[6].

Axiom 2: The measure must generate at least two equivalence classes. This axiom demands an existence type proof, that is it is only necessary to postulate two structures that yield different metrics in order to establish the validity of this proposition. This obligation is satisfied by the a null or empty design that has a measurement value of work equal to zero, whilst a design that satisfies a single functional requirement and comprising of a single module will have a value of work approximately equal to 1.67. The formal proof is given in Appendix D.

[6]Furthermore this definition has been shown to be fully operational by transforming the algebra into OBJ and executing the program.

Axiom 3: An equality relation is required. No proof is offered as the relation is axiomatic to our algebraic system, along with propositional logic and natural numbers.

Axiom 4: There must exist two or more structures that will be assigned to the same equivalence class. To satisfy this all that is required is to find two different design structures and specifications that they fulfil, which yield the same measurement. Two such structures are shown in Figure 6.1. In this slightly contrived example, both designs are based upon the same specification but the second design comprises an additional module m_2, but one which satisfies no functional requirement. Both designs have identical work metric values, as is formally proved in Appendix D and consequently, the axiom is shown to be true, because examples of different designs have been given that are members of the same equivalence class - in this case the class with a value of 1.67 - for the work metric.

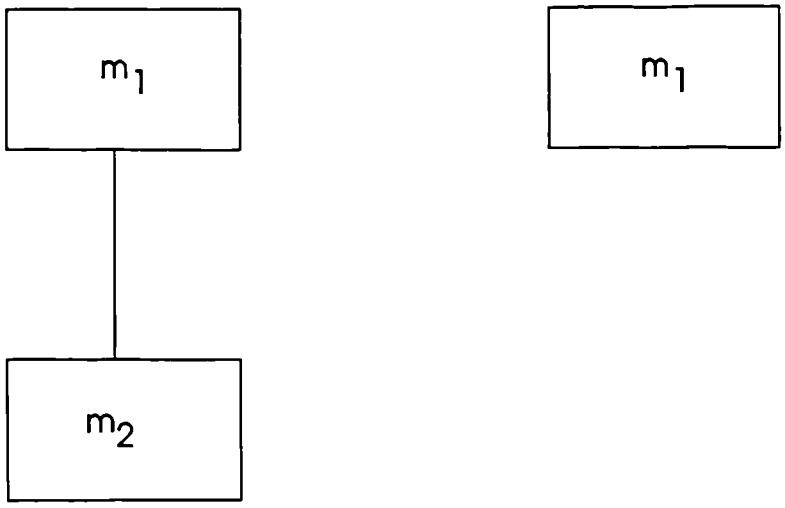


Figure 6.1: An example of two designs that satisfy Axiom 4

Axiom 5: The metric must not produce anomalies (i.e. the metric must preserve empirical orderings). Since the measurement system is based upon the number system of natural numbers this is satisfied[7]. For the proof refer to Krantz *et al* [Kran71].

---

[7]In practice this axiom only becomes non-trivial when the measurement system is based upon such mathematical exotica as vectors, when it is not at all obvious that the Representation Theorem holds (e.g. Hansen's modification to the cyclomatic measure [Hans78]). This axiom assumes away the problem of errors

Axiom 6: The Uniqueness Theorem must hold [Supp71] for all permissible transformations for the particular scale type (i.e. there is a homomorphism between the transformed and the measurement structures). The underlying question is whether the measurement system is adequate for the type of measurement scale selected. For an ordinal scale the measurement structure must be order preserving for any monotonically increasing transformation function. This is trivially true [Stev59, Supp71, Kran71].

Axiom 7: Adding an additional requirement to the system specification must increase the work metric.

$$\forall \text{ S:spec; D:des r:req; m:mod} \cdot$$
$$\text{SATSPEC(D,S)} \Rightarrow \text{work(S,D)} < \text{work(addr(S),sat(m,r,D))}$$

Essentially there are two parts to this proof. First, it must shown that if specification S is implemented by design D, then adding a new requirement to S implies adding a requirement satisfaction operation *sat* to D. The design may also be augmented by an additional module, but this is incidental to our argument as will become apparent[8]. Second, it must be shown that if the number of *sat* operations for design D is increased then the work metric must also increase for D. Since the required proof is universal in nature, induction will be employed. The base case is a null specification S and design D, whilst the n+1 case is a requirement concatenated to S and a module to D. Again the full proof is presented in Appendix D where it is shown that the axiom holds for the base case and for the n^{th}+1 case, and therefore by inductive reasoning, for all cases.

Axiom 8: Related to this axiom is that the metric $work_i$ must increase when the ith module satisfies an additional requirement, in other words the module specification has increased.

$$\forall \text{ i:mod; } r_1, r_2\text{:req; S:spec; D:des} \cdot \text{work}_i\text{(i,S,D)} < \text{work}_i\text{(i,adds(r1,r2,S),sat(i,r1,D))}$$

Since this axiom is extremely similar to the previous axiom, the proof will be omitted. The work metric for the system is the sum of the metric values for individual modules, and this can be seen in the previous proof in that the *work* operation reduces to the sum of the $work_i$ operations for each *addm* operation. Thus, we have already considered the more general case and Axiom 8 may be conceived as a specific instance of Axiom 7.

---

in the measurement process, however, with the application of software measurement tools any errors will at least be systematic!

[8] It is the sat and not the addm operation that increases the value of $work_i$.

Axiom 9: There must exist different designs that satisfy the same specification but which have different work metric values. In other words design size is not only a function of the specification but also the intrinsic organisation of the design. The consequence of this axiom is that for a given specification choice of architecture can influence size.

$$\exists D_1,D_2:des; S:spec \cdot satspec(D_1,S) \wedge satspec(D_2,S) \Rightarrow work(S,D_1) <> work(S,D_2)$$

Doubtless, by this stage the reader will be grateful to note that proof of the validity of this axiom for the model is an existential one; we merely have to find two designs for which the axiom holds in order to establish its validity. In order to further reduce the length of the proof, the last example employed to establish axiom 7 will be re-utilised along with a different design to implement the same specification (refer to Figure 6.2). The specification common to both designs, S is:

addr(r1,",addr(r,",newspec))

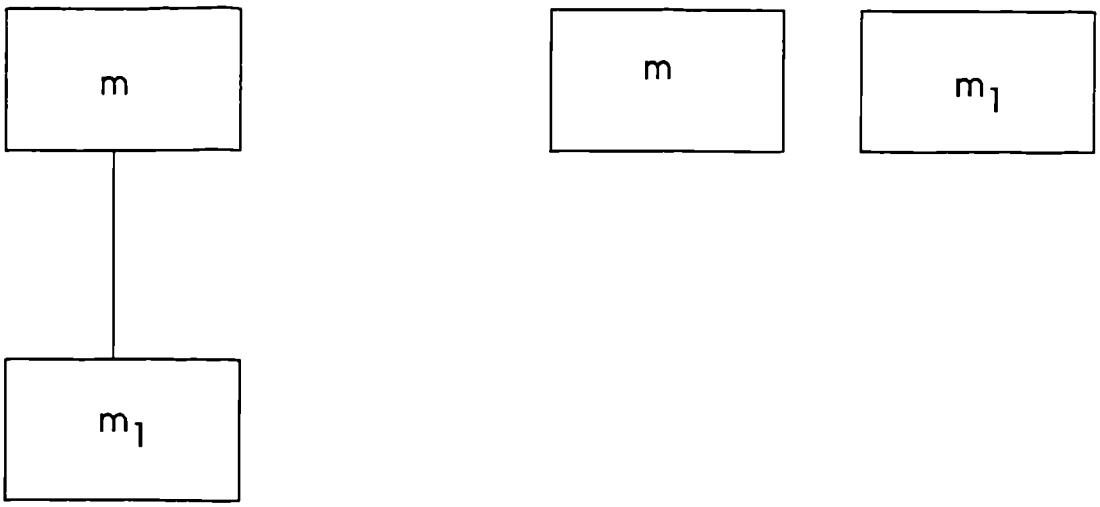


Figure 6.2: An example of a differing design that satisfies the same specification

The first design, D1, with a known work metric value of 3.33 is:

sat(m1,r1,addm(m1,",sat(m,r,addm(m,",newdes))))

whilst the second design, D2, is formally described as:

sat(m1,r1,addm(m1,m,sat(m,r,addm(m,",newdes))))

The difference between the designs lies in the fact that in the second design D2, module m1 is a subordinate of module m and as a consequence m inherits the requirement satisfaction of module m and therefore has to do scheduling work, the result of which is a higher work metric value. This is shown more formally in Appendix D where it is demonstrated that the two designs have values of 4.0 and 3.33 respectively. Consequently, Axiom 9 stands.

To summarise the progress from our theoretical treatment of the model: it has been formally defined by means of an algebra, the desired properties of the model have been given as a set of nine axioms and then each axiom demonstrated to be true by means of a formal proof. This suggests that there are some grounds for confidence in the model and that it is appropriate to progress to an empirical evaluation. It must be emphasised though, that a theoretical treatment is in no way seen as a substitute for - but rather as complementary to - empirical validation. There are two reasons why this is so. First, the proofs are in many cases lengthy and so the possibility that they may contain flaws cannot be ignored. Second, the axiom set that describes the desired model behaviour may not cover all significant model characteristics. Third, the model may not capture all relevant factors, for example the 'work' metric does not incorporate any notion of application area yet this might be highly influential in terms of accurately capturing module size at design time. Unfortunately, since a formal system is by definition a closed one, our theoretical evaluation has nothing to say concerning factors external to the model. Hence the need for empirical evaluation.

6.3.2 Empirical validation

There are two areas of the model that need to be subject to empirical investigation. First, there is the relationship between the "work" metric and size related factors, such as traditional measures like LOC and also other factors such as the number of variables referenced. Second, there is the question of the extent to which the "work" metric may be used to improve the basic model of system architecture outlined in the previous chapter. I will postpone dealing with the second area until after techniques for combining dimensions have been discussed later in this chapter.

Since the need to be able to crudely measure module size during high level design, was prompted by the discovery of a small number of "rogue" architectures, it is appropriate to attempt to validate the "work" metric against this data. The data is derived from the

experiment to relate information flow based metrics of design structure to development effort and reliability. The reader will recall that it is based upon 13 adventure game systems, that had been developed by teams of three or four BSc Computer Science students. These ranged in size between 14 and 33 modules and approximately 350 to 1000 ELOC and were implemented in Pascal. Further details may be found in the previous chapter and [Shep90a]. Certain system architectures were observed to exploit the trade-off between component size and structural complexity by embedding structural complexity within components, but at the cost of having a small number of abnormally large components.

	work	ELOC	decs	vars
ELOC	0.927			
decs	0.895	0.963		
vars	0.694	0.692	0.679	
C&A	-0.141	0.067	0.018	0.205

decs = no. of decisions
vars = no. of variables referenced
C&A = Card and Agresti measure [Card88]

Table 6.1: Cross correlations of module size measures based on rank

Table 6.1 presents the cross correlations between the "work" metric and other indicators of module size. We also used the upper tail of boxplots to identify abnormally large modules. Where a module was classified as abnormal for at least two out of three size indicators (ELOC, vars and decs) the module was deemed to be a "large" module. The "work" metric identified eight out of nine such modules. Furthermore, only one "non-large" module was incorrectly pin-pointed. This suggests that "work" is a good indicator of design component size with the *caveat* that it has a low resolution, in other words it is not an effective discriminator between modules of roughly equal size. This is hardly surprising given the subjective nature of tracing functional requirements to module hierarchies and the lack of homogeneity amongst the primitive requirements.

The above data suggests that there are good grounds for employing the "work" metric as a crude indicator of module size at design time. The "work" metric shows significant correlations with all three module size indicators at the 1% confidence level. In order to provide a comparison, the empirical study also investigated the Card and Agresti intra-modular metric [Card88]. Unlike the work metric, it revealed no significant correlations

and does not appear to be a fruitful method of identifying abnormally large modules at design time[9]. This can be partly explained by the fact the Card and Agresti approach ignores the effect of scheduling work upon size. There model suggests that module size decreases as the number of subordinate modules increases. Our study found that to the contrary, module size increases because additional code is required to control, or schedule these extra subordinate modules. Clearly, without further empirical investigation - preferably industrially based - one has reservations about making extravagant claims for the work metric. Nevertheless, the model appears to have some merit, and in the absence of a better approach it will be adopted into the multidimensional model of high level design or system architecture.

6.4 The Multidimensional Model

The next step is to integrate the models of module size and design structure, so as to provide a more comprehensive picture of software design. Thus, we have two concerns. First, to explore techniques for the integration of simple or uni-dimensional models, and their associated metrics, into more complex or multidimensional models. Despite, increasing recognition of the need for more sophisticated approaches, for example Basili and Rombach [Basi88], there is limited work in this area. Second, a much more specific concern, namely the development of a model based upon the dimensions of size and structure to aid the software engineer at design time in the production of system architectures that are easier to implement and more reliable.

The most typical approach to the integration of more than one model might be characterised as an abrogative method. This is based on the recognition that a single metric is an inadequate means to represent such a complex object as a software system consequently the software engineer is presented with many metrics. However, the responsibility to integrate the metrics into a model is abrogated. Instead, specious analogies are drawn with car dashboards[10]. Essentially the user of the metrics is being asked to build implicit and informal models because the researchers are unable to do so. It is arguable that this might sometimes unfortunately be a necessity, but it is hardly meritorious or worthy of the epithet science. Examples of this method include the work of Hansen [Hans78], Bache and Tinker [Bach88] and also that of Kaposi and Myers [Kapo90] The overwhelming problem of this type of approach lies in the problem of

[9]This is does not necessarily mean that the Card and Agresti approach is without merit; merely that it is unsuitable for this particular application.

[10]For example, Whitty has argued that it is the role of the user to combine, possibly disregard, and interpret the set of 18 metrics derived from their flow graph based metrics tool QUALMS.

comparison. What, for example, is one to make of the two pairs <*a*,*b*> and <*c*,*d*> where *a*>*c* and *d*>*b*?[11]

The second approach might be regarded as the additive method. Again, the researchers recognise that an adequate model cannot be devised based upon a single factor, but in contrast to the previous approach, attempt to additively combine in a numeric fashion, n factors, normally by means of weightings. This is legitimate only, where factors are essentially measures of the same dimension but differ in unit[12] Additive combination is unsound when the factors are not related to the same dimension, that is orthogonal, because, this leads to the apples and oranges type of problem where a concatenation operation cannot be defined. An example of this approach is the complexity metric of Oviedo [Ovie80] which attempts to add control and data flow complexity.

The third approach is to map n factors onto n-dimensional space. For example, two factors can be modelled by the familiar scatter diagram. The space may then be partitioned in order to generate a classification system. The simplest method is merely to characterise values for each dimension, or factor, as outlier or non-outlier[13]. These are then combined with all other dimensions to yield a 2^n classification scheme, where n is the number of dimensions. This method offers two advantages. First, the model deals with the product of the dimensions, which overcomes both problems of units and orthogonality [Kran71]. Second, it allows for a more sophisticated response by the user of the metrics. By way of illustration, one might view an abnormally large module differently if it is also known that it has a very simple interface with the remainder of the system architecture, than if the interface were complex. In the first instance the module might be considered as a strong candidate for partitioning. In the second instance a complete re-design might be more appropriate! This approach has been successfully exploited by Kitchenham *et al* [Kitc89, Kitc90].

---

[11]A further problem emerges from the absence of any well defined relational operator, other than equality, in that the metric fails to satisfy the Uniqueness Theorem for any scale other than nominal. Consequently, the metric would fail to satisfy the majority of axioms previously outlined in this thesis.

[12]An example might be where length is in part measured in metres and part by inches. The use of weightings would allow one to arrive at an integrated measure of length, in this case by multiplying inches by approximately 39. A software engineering example is the different weightings assigned to the constituent counts of Albrecht's function points [Albr83].

[13]In other words, abnormal or normal. Refer to [Kitc87b, Kitc89, Link91] for a discussion of outlier analysis techniques.

Using this third approach, we combined both size (work) and structure ($IF4_m$) metrics using a two dimensional scatter plot diagram and simply used the upper quartile of ranked values to identify outlier or abnormal modules (see Figure 6.3).

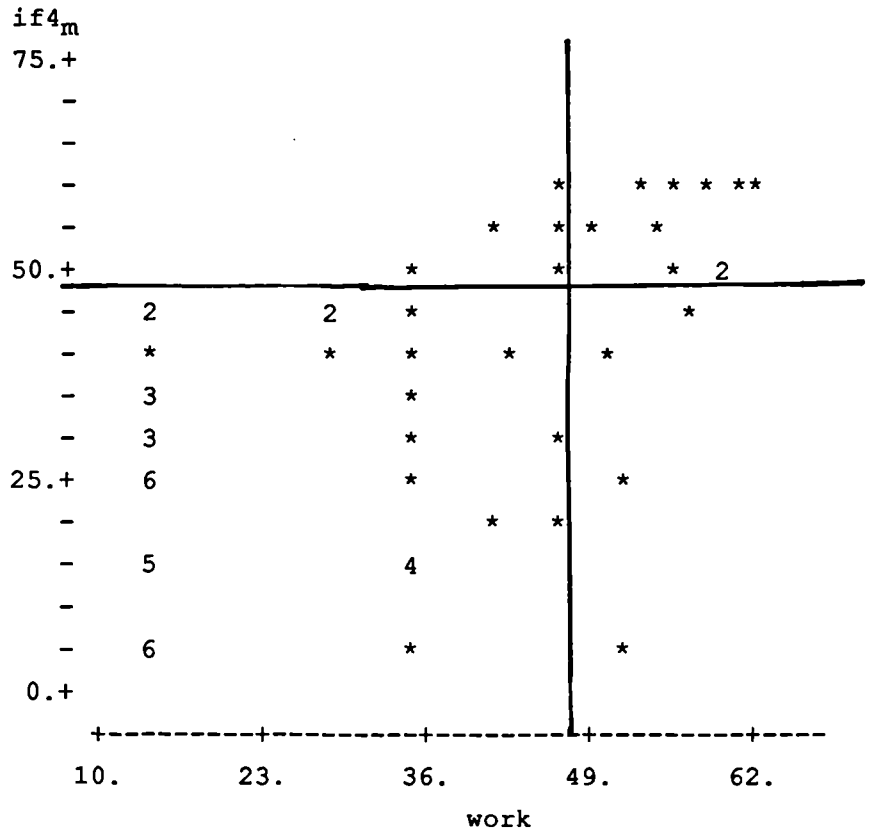


Figure 6.3: Structure vs. Size (By Rank)

This gives a fourfold classification of modules:

low $IF4_m$-low work (43 modules)
low $IF4_m$-high work (4 modules)
high $IF4_m$-low work (5 modules)
high $IF4_m$-high work (10 modules).

The low $IF4_m$-low work modules are not a cause for concern to the project manager. In this study the 43 modules in this class had an average error rate of 0.07 known errors per module. This contrasts with an average error rate of 0.58 for modules drawn from the

other three categories. It is important to stress that a single metric would only have identified two out of the three classes of problem module.

The problem modules vary in type according to their class. The high $IF4_m$ - high work modules manifest the most serious design faults in that they are both have a high workload and have a complex interface with the remainder of the system. Such metrics would be identified as outliers by either metric in isolation. The low $IF4_m$ - high work modules are those modules that initially motivated this work, since they exhibit the trading of structural complexity for size, in other words much of the complexity is embedded within the module. This suggests the need for further partitioning in order to generate a maintainable and re-usable system architecture. The final class are those modules that do not have a high workload yet have a complex interface with the remainder of the system. Typically these modules arose when a single function was split between several modules or as a consequence of a lack of data structure isolation.

The final issue that this empirical analysis raises is why do system architectures potentially contain such abnormal components? The systems under study were intended as serious pieces of software and not merely as intellectual curios. The developers were unaware that a study was being conducted and we believe it unlikely that they were consciously exploiting the size / structural complexity trade-off. Our investigation indicates that there are three contributory causes.

First, in many instances functions were split across two levels of the module hierarchy. Thus, the higher level, or calling module may contain many partial functions in addition to any scheduling that it performs.

Second, all four architectures exhibited very broad hierarchies, in one case a module had 17 children. This adds greatly to the scheduling workload of the parent module.

Third, many functions are misplaced within the module hierarchy, thereby reducing the scope for procedural abstractions. A higher level requirement cannot be satisfied until all the sub-requirements are satisfied. This is delayed until higher up the module hierarchy, if one or more sub-requirements are not within the scope of a module, and therefore cannot be inherited. The scheduling workload of calling modules is increased as the module must have knowledge concerning the details of the partial satisfaction of the requirement. Where a higher level requirement is fully satisfied, then the calling module may regard it in a more abstract vein without the need to know about the more detailed functions that go to make up the requirement. This is potentially a most serious design fault, as it will have a negative impact upon, ease of comprehension, maintainability, and re-usability of the resultant software.

6.5 Summary

In this chapter we have revised and extended the simple uni-dimensional model presented in the previous chapter. In particular, we have addressed the problems uncovered by the application of the validation techniques encompassed within our metrics development method. The theoretical and empirical evaluation of the enhanced multidimensional model indicates that this new model represents an improvement upon the earlier model. It also illustrates the iterative nature of model building and highlights the probability that there still remains scope for further improvement.

Module "work" has been shown to be a useful metric for the identification of abnormally large design components during the design phase of the software life cycle. Taken on its own, the "work" metric has a relatively low resolution. It does not effectively distinguish between components of approximately the same size. On the other hand, our empirical findings suggest that it is reliable in identifying the outlier or very large system components. This is extremely valuable when combined with other measures, such as the information flow metrics of structural complexity [Ince89a, Shep90a,b] as we can now construct a multidimensional model of a software design.

The analysis of system structures using scatterplots of two design metrics illustrates that a system architecture may contain more than one - in this case three - distinct classes of outlier module. These classes of outliers can be used as profiles to facilitate the comparison of software designs. Furthermore, the project manager may wish to adopt different remedial actions for the different classes of outlier module. Perhaps most significantly, however, no one metric could detect modules from all three outlier classes. We conclude, therefore, that multidimensional modelling is a technique of some significance for the software metrologist and the software engineer alike.

The empirical study is based upon relatively small scale software systems for one particular domain, that of interactive computer games. It is not clear to what extent these results may be translated to larger systems and other application domains. For the study of larger systems, the development of an automated software tool is an imperative, as the measuring process can become extremely onerous.

Finally, to summarise, the tracing of functional software requirements to design components provides a new perspective for understanding and measuring system architecture. Metrics derived from this mapping provide reliable insights into design component size and when these are used in conjunction with other design metrics such as the structure metrics of information flow, provide valuable help for the software designer's decision making.

7. CONCLUSIONS

Non modo ... sed etiam ...

"Now hauing perorated (as he thinkes) sufficiently, he beginnes to growe to a conclusion."

Attributed to Francis Bacon

Synopsis

In which we review the scope of the research described within this thesis, and summarise its major findings. We go onto discuss some of the weaknesses of this research, and highlight the areas of potential significance that have been uncovered, but which have yet to be explored.

7.1 Research Coverage

To recap; this research has been concerned with the development and even more importantly, the evaluation of a set of system architecture metrics that might provide the software designer with feedback whilst making design decisions. This aim has in turn generated a secondary aim, which has been the derivation of a method to support the development and evaluation of software metrics in general.

The research has been restricted to product metrics only, and has concentrated upon "function strong" as opposed to "data strong" systems[1]. Nevertheless, it is a contention of this thesis that many of the principles have wider application - a point that will be expanded in the next section.

The work has commences with a detailed review and critique of accomplishments, to date, in the area of software engineering product metrics, with particular emphasis upon three of the best known metrics, one of which might reasonably be regarded as the classic design, or system architecture metric. Out of this analysis, recurring patterns and behaviours are observed. These suggest a method that is generally applicable for

[1]"Function strong" and "data strong" systems is a distinction suggested by deMarco [deMa82] in order to distinguish between those systems characterised by procedural or functional complexity, and those characterised by data complexity.

the development and evaluation of software metrics and their underlying models. The method is then used to try to solve the primary objective of this research: that of developing metrics for the designers of software architecture. It also offers a two pronged approach of both theoretical and empirical evaluation of the metrics which in turn has suggested further enhancements to the model.

The outcome of this work has been the evolution of a model to support the decision making process of a system architect, that has been both formally verified and empirically examined including against an industrial project. A second outcome, is a method which has wider implications for the development, verification and validation of software engineering metrics in general, and clearly could include both process metrics and metrics related to data architectures[2] - areas that have been excluded from this research.

7.2 Summary of Findings

What then has the above research established? First, that despite the extraordinarily large number of different metrics and the inventiveness of workers in the field, there has been little progress in terms of metrics achieving widespread acceptance. This is the direct consequence of unsatisfactory validations. Unfortunately it is rather easier to propose metrics than it is to validate them. Many of the metrics in the literature remain completely unvalidated, other than by vague conjectures and appeals by their creators, to our collective intuitions[3]. Plainly, this not the result of some communal conspiracy amongst the computer science fraternity, but due to the absence of any agreed framework for the validation of metrics, poorly articulated underlying models, a lack of definitions - most notably of what exactly it is that is being measured, questionable experimental design and the frequent misapplication of statistics.

---

[2]Recent research indicates that many of the ideas derived from the modelling and measurement of functionally oriented architectures are indeed relevant to data architecture [Ince90b,c]. For example, connections between entities in the form of relationships appear to have a significant bearing upon implementation difficulties of the resultant system.

[3]A few examples include, the Myers' and the Hansen modification [Hans78]to the McCabe metric [Myer77]. Other variations on a theme include [Iyen82, Negr83, Stet84, Sinh86] all without empirical validations. Yet another example is Harrison and Magel's attempt to combine Halstead's metric with a development of McCabe's metric based on nesting level [Harr81]. They argue that their metric is "more intuitively satisfying". No further validation is offered. A more recent design metric example is McCabe's family of design metrics [McCa89], again proposed without any published empirical support. One could go on, instead however, the unconvinced reader is referred back to chapters two and three.

Careful analysis of Halstead's software science [Hals77], McCabe's cyclomatic complexity measure [McCa76] and the information flow metric of Henry and Kafura [Henr79, 81a], has revealed a number of recurring patterns. In each case it is unclear exactly what is being measured, and terms such as "complexity" and "quality" predominate. Since these can hardly be regarded as operational definitions, empirical validation becomes a rather more difficult undertaking. Thus, the same metric is validated against a whole range of different quality factors by different investigators; for instance the information flow metric has been applied to estimating implementation effort [Ince89a], coding time and changes [KafC85], maintainability [KafR87, Romb87a, Shep90e], comprehensibility [Romb87a] and error incidence [Kitc88]. Although, these factors are doubtless related, one cannot help wondering whether slightly more focus might expedite validation. Also, whether it is expecting rather a lot from what are after all comparatively naive models that it should be able to accurately predict such a range of software characteristics in an equally diverse range of environments and applications.

Related to the problem of unclear measurement goals are poorly articulated models. Again, using information flow as an example, it is unclear what is meant by a global data structure, there are conflicting definitions of global flows [Henr81a p512 Definition 1] compared with the formula given by [Henr81a p514]. An outcome of the scant regard paid to the model underlying a metric, is anomalous behaviour such as the unintentional bias - of the Henry and Kafura metric - in the counting of the parameterised or local information flows against the under-counting of global flows [Ince89a, Shep90a].

Returning to the empirical validations of these metrics; closer inspection reveals much of the claimed support to be illusory. This is well illustrated by examining the alleged support for the information flow metric derived from an analysis of UNIX 'error' data [Henr81a,b]. The first problem is that in actual fact the authors used change data from a new release of the UNIX operating system, not error data. There are many reasons why code may be changed between versions of a system other than error correction. Second, their analysis is based upon logarithmic class intervals and even then Henry and Kafura still had to make the arbitrary decision of merging the highest three classes in order to transform a correlation coefficient of $r_s=0.21$ into $r_s=0.94$. Thus important an empirical evaluation of the information flow metric rests upon a mixture of arbitrary decisions and a mere six data points.

To summarise then, there are ill defined measurement goals, underlying models that are, in the main, implicit, and questionable empirical validations of the software metrics. These problems are common to all three metrics, despite the high degree of attention that they all received, from the software engineering community over the past few years. Why should this be so? The research described in this thesis suggests two answers. The fundamental reason has been the concentration upon the *minutiae* of

measurement without an equal concern for the higher level factors, such as goals and methods. The second explanation, which in many ways arises out of the first, is the lowly role accorded the measurement model. A model captures a theory concerning the measurement application and it is this model that provides a meaning and a context for a measurement. And this is true as much for software metrics as for any other area of measurement. Given this back drop it is hardly surprising that the majority of metrics work has been *ad hoc* and poorly validated.

This then was the background for the research in thesis. Thus the agenda became one of:

- finding a framework to describe models that underlie software metrics;
- devising more effective techniques for the evaluation of metrics;
- developing a method to guide the would-be software metrologist in the creation, selection and validation of software metrics.

We will discuss our findings in each area in turn.

First, the framework for describing a model. As has already been noted, the vast majority of measurement models are entirely implicit. Even when some attempt has been made to present the model behind the metric these are usually incomplete in one, or more respects. This research has suggested that models may be informally described under seven headings, namely:

- inputs
- outputs
- parameters
- relationships
- mappings from and onto the "real world"
- model limitations
- model reliability

Probably the last three headings are the most widely ignored yet they are all vital if the model is to have any real software engineering application and also to enable meaningful evaluation. This thesis then shows how, this relatively informal approach may be further refined and the behaviour of the model formally defined in terms of inputs, outputs, parameters and relationships between these model components. This can be accomplished by the algebraic approach of Guttag [Gutt77] and Liskov [Lisk86] where the model is defined as an equational rewrite system, each equation representing an axiom or property of the model. Such formality would seem to be indispensable if a model is to be adequately scrutinised. The precision that it offers would also seem to be valuable for anybody proposing to build software tools to

automate the measurement process. Algebraic specifications offer another advantage, in addition to unambiguity, which is that they can easily be animated using such executable specification languages as OBJ[4].

The second item on the research agenda has been that of metric evaluation. As the thesis title implies, it has been argued that this lies at the heart of the research. There is little point in defining measurement models, however formally, if there is no prospect of being able to evaluate (and if appropriate refine) them. Given the relatively new state of the whole subject area of software engineering, it would not be surprising if our model building is a highly iterative process of postulating, evaluation and refinement. Consequently, it would seem entirely appropriate that we consider the tools and mechanisms at our disposal to carry out this evaluation. This thesis has made three contributions in this direction.

First, evaluation is greatly facilitated if the metric is carefully defined, when it is clear what is being measured, that any model limitations or assumptions are made explicit and some indications are given of the required level accuracy.

Next, we have proposed an entirely novel approach to the problem of validation of the theory or model behind the metric. This we term theoretical metric evaluation which makes use of algebraic model specifications. A flexible framework has been described which enables those carrying out the evaluation to evolve a set of model invariants, or properties of the model that they wish always to be true. An example of such an invariant would be that a metric should always increase if a further module is added to an arbitrary system architecture. It is then possible, as this thesis has demonstrated, to prove that the invariant holds, or that in certain circumstances it is violated. It is then up to the validators to determine whether they wish to relax the invariant or modify the model. This method has been successfully used to reveal a number of problems with the unidimensional model of system architecture described within this thesis.

One benefit of formal proofs, is an extremely pragmatic one, and that is that they are normally far quicker and therefore cheaper to conduct than industrial scale empirical validations. For this reason alone it is suggested that they be considered prior to empirical analysis so that if model refinements are found to be necessary these may be carried out prior to the potentially more costly empirical work. Another benefit is that it is possible to reason about all possible measurement objects, whereas empirical

---

[4]The IF4 metrics and their underlying model have been implemented by the author, using the OBJ EX environment, thereby allowing "what if" style investigation of the model. The main disadvantage of such implementations is that they require the user to be familiar with equational rewriting and are consequently not very user friendly! Nevertheless, they are potentially useful given the very small amount of effort to transform the style of notation used in this thesis into OBJ.

analyses tend to adopt a sampling approach, where hopefully the sample is representative of the larger population. This means that the theoretical approach might uncover a problem with a rare but pathological object that an empirical approach might not detect. We argue that this theoretical validation technique offers several distinctive advantages over more conventional methods of model evaluation. We do not argue that it should supplant empirical validation, but that it should be regarded as complementary.

The usual approach to the problem of evaluating a measurement model is empirical. Clearly, this is an equally important method of model evaluation as the theoretical approach, but one which is likely to uncover *different* types of problems. A weakness with formal systems is that they must necessarily describe closed systems. However, it is often precisely that part of a system that is excluded from a model that can cause most modelling difficulties. Thus, formal approaches are not effective and detecting what might be thought of as "sins of omission". Fortunately, it is precisely these types of problems that an empirical analysis can be successful at uncovering. This thesis has made no special contribution in this area other than to suggest the characteristics whereby an empirical analysis may be judged. These are:

- the hypothesis under investigation;
- the artificiality of the data used;
- the validity of the statistics employed.

It is recognised that in a world of finite resources and complex interacting systems and processes the ideal cannot be attained, but at least it enables workers to decide how much importance to attach to a particular empirical result. Even in this sub-optimal world that we inhabit, one would still expect an empirical validation to be capable of refuting its hypothesis and of employing meaningful statistics.

The third item on the research agenda was the evolution of a method to guide workers attempting to develop and evaluate software metrics. The method proposed builds upon the pioneering work of Basili and Rombach with their GQM method [Romb87b, BasR88]. However, the method suggested within this thesis emphasises the need to make the underlying model explicit, to evaluate and to refine. The method has six stages:

- problem identification
- construction of an informal model
- transformation into a formal model
- derivation of the model axioms
- theoretical model evaluation
- empirical model evaluation

We have also emphasised the iterative nature of the method, and therefore the likelihood of backtracking.

In order to consider the validity of this method, it has been applied to the problem of developing design metrics for a system architect. Two models have been proposed. A simple model based upon the IF4 metric [Shep90a] was found to be wanting in a number of respects, and a more complex model was evolved, based upon the combining of two metrics: the multidimensional model [Shep90b]. The latter was found to satisfy all its axioms of desired model behaviour (i.e. its invariants). Furthermore, two separate empirical investigations (described earlier in the thesis) have shown the metrics to yield useful engineering approximations, in relating the model to development effort and maintenance problems. Statistically significant relationships have been found. Consequently, it has been argued that the design metrics of IF4 and "work" coupled with their supporting model have utility in supporting the engineering of system architectures. Practical applications include selecting architectures that will require less development effort, identifying potential maintenance problem areas prior to implementation, using the model to guide and focus restructuring and re-engineering activities for "geriatric" software systems[5] and to provide a suggested order of priority for modules undergoing design inspection or review[6].

In conclusion, the areas of progress from this doctoral research have been threefold. A new framework for describing measurement models has been proposed. Second, a new technique for model evaluation has been devised, based upon the application of algebras and term rewriting. Last, a method has been given to guide future metrics workers through the problematic tasks of metric development and its frequently overlooked evaluation. *Not only* have these ideas been proposed, *but also*, they have tried on the non-trivial task of developing system architecture metrics to guide the designer in the task of selecting structures that easy to develop and maintain.

7.3 Suggestions for Further Work

[5]For example, the costs of completely restructuring a system causing maintenance difficulties may be prohibitive, but use of the IF4 and "work" metrics can suggest those parts of an architecture from which most restructuring benefit would be obtained. This capitalises upon the idea that very often much of the benefit may be obtained from a small proportion of the work.

[6]If a fixed time has been allocated for a design inspection then it is appropriate to concentrate upon the potentially most critical components first. The author has sat in a number of inspection meetings where the ordering of the review has been driven by execution order, from working left to right and top to bottom from a module hierarchy chart or even, on one occasion, in alphabetical order!

Although it has been argued that this research has made some small contribution in the software metrics arena, there are clearly weaknesses and avenues that have yet to be explored.

There are two weaknesses that need to be brought to light. One is that the model based metric development method is relatively untested - other than by its progenitor - so its worth for other types of measurement application has yet to be demonstrated. What this research has revealed, and what is incontrovertible, is that there is a pressing need for methods to guide the developers and users of software metrics. Doubtless the method outlined in this thesis can, and hopefully will be in due course, modified, adapted and improved upon. What is important, is to appreciate the need for method. The second weakness concerns the model of system architecture evolved during the course of this research. Adherence to the model based development method has lead to one cycle of revision, however, it may be that other empirical evaluations (particularly by evaluations that conform more closely to our *desiderata* for empirical studies) might highlight other difficulties that call for further refinements. Again, though this is likely to be the case, it has been argued that the model developed, has been the subject of methodical evaluation and that there is at least some basis for confidence in its utility. It is not the contention of this thesis that the model is in some sense a definitive model of system architecture. Other work such as [BasR88] and [Shep90f] suggest that there are no definitive metrics or models but rather one needs a whole multiplicity of metrics each suited to different measurement goals and applications.

Avenues that have remained unexplored include the design of data architectures. Plainly, this is an issue of some importance given the large number of database systems in existence. May be even more significant is the lack of concern with the application of metrics and models *once* they have been satisfactorily evaluated. In other words, metrics researchers must also consider the issue of how are these metrics to be used by software engineers. One reason why this may become important is that by focusing solely upon a product, say a system architecture, one loses sight of the fact that the same product may enter into a number of different processes, and therefore take on a number of different meanings. For example, an engineer may be designing an architecture from scratch, doing some maintenance work, optimising the architecture to enhance performance, searching for reusable components or restructuring an ageing software system. In each case the architecture takes on a different meaning, a point not captured by our product oriented view of measurement and modelling. Again, it may be more fruitful to look for a general framework rather than focus too strongly upon individual metrics and applications. A possibility might be to apply some of the concepts and formalisms from the area of software engineering research known as

process modelling[7]. The outcome might be quantitative process models that more precisely describe how metrics are to be integrated into software engineering projects.

7.4 Postscript

To conclude on a slightly more personal note. Has this thesis over-emphasised the role of methods? Is this really the manner in which progress is made? After all if one were to be honest - and for a brief moment I shall try to be - this research has not be conducted in an entirely systematic fashion, driven by a single goal since the first day of its inception. Obviously not! On occasions I have acted in an opportunistic, even spontaneous fashion, with little regard to master plans, methods or objectives. The original goal of seeking to provide useful metrics for the software designer, particularly with respect to the quality factors of ease of implementation, reliability and maintainability, has been considerably enlarged in scope, so as to encompass broader goals such as a framework for the development and evaluation of metrics and their models in general. Also, the quality factor of reliability has not been pursued.

Is this hypocritical? Hopefully not. In the first instance progress in the absence of methods has hardly been one uninterrupted success story. Second, methods as a means of guidance, as opposed to some procedural strait-jacket, are valuable for other types of human endeavour *vide* more conventional forms of architecture. Lastly, one can justify and describe activities *as if* one had followed a method. This at least provides structure and therefore a basis for comparison with other activities doubtless the outcome of equally anarchic processes!

So, my case rests.

[7]For an example of the coverage of this type of work the reader is referred to the Proceedings of ACM International Process Modelling Workshops, alternatively Humphrey gives a good review [Hump89].

References

[Adam66] Adams, E.W. 'On the nature and purpose of measurement'. *Synthese* 16 pp125-169. 1966.

[Adam88] Adam, M.F. LeGall, G. Moreau, B. Vallete, B. 'Towards an observatory aiming at controlling the software quality'. *Proc. IEE/BCS Conf. Software Eng. '88*, pp50-54, Liverpool, England. 1988.

[Akiy71] Akiyama, F. 'An example of system software debugging'. *Proc. IFIP Congress*, pp353-358. 1971.

[Albr83] Albrecht, A.J. Gaffney, J.R. 'Software function, source lines of code, and development effort prediction: a software science validation'. *IEEE Trans. on Softw. Eng.* 9(6) pp639-648. 1983.

[Alex64] Alexander, C. *'Notes on the synthesis of form'* Harvard University Press, Cambridge MA, 1964.

[Arth85] Arthur, L.J. *'Measuring programmer productivity and software quality'.* Wiley-Interscience. 1985.

[Ashw90] Ashworth, C Goodland, M *SSADM, a practical approach* McGraw-Hill, 1990.

[Bach88] Bache, R. Tinker, R. 'A rigorous approach to metrication: a field trial using KINDRA' *Proc. BCS / IEE Software Engineering '88* July 12- 15, Liverpool University, pp28-32. 1988.

[Bail81a] Bailey, C. Dingee, W. 'A software study using Halstead metrics'. *ACM SIGMETRICS P.E.R.* 10, Spring, pp189-197. 1981.

[Bake79] Baker, A.L. 'The use of software science in evaluating modularity concepts'. *IEEE Trans. on Softw. Eng.* 5(2) pp110-120. 1979.

[Bake80] Baker, A.L. Zweben, S.H. 'A comparison of measures of control flow complexity'. *IEEE Trans. on Softw. Eng.* 6(6) pp506-511. 1980.

[Balu74] Balut, N. Halstead, M.H. Bayer, R. 'Experimental validation of a structural property of FORTRAN programs'. *Proc. ACM Natl. Conf.*, pp207-211. 1974.

[Basi81] Basili, V.R. 'Evaluating software development characteristics: an assessment of software measures in the Software Engineering Laboratory'. *Proc. 6th Annu. Softw. Eng. Workshop, NASA/GSFC.* 1981.

[BasH83] Basili, V.R. Hutchens, D.H. 'An empirical study of a syntactic complexity family'. *IEEE Trans. on Softw. Eng.* 9(6) pp664-672. 1983.

[BasP81] Basili, V.R. Phillips, T. 'Evaluating and comparing the software metrics in the Software Engineering Laboratory'. *ACM SIGMETRICS P.E.R.* 10 Spring pp95-106. 1981.

[BasP84] Basili, V.R. Perricone, B.T. 'Software errors and complexity: an empirical investigation' *CACM* 27(1) pp42-52. 1984.

[BasR79] Basili, V.R. Reiter, R.W. 'Evaluating automatable measures of software development'. *Proc. IEEE Workshop on Quant. Softw. Models,* pp107-116. 1979.

[BasR87] Basili, V.R. Rombach, H.D. 'Tailoring the software process to project goals and environments'. *Proc. 9th Int. Softw. Eng. Conf.* Monterey, CA.. pp345-357. 1987.

[BasR88] Basili, V.R. Rombach, H.D. 'The TAME Project: Towards Improvement-Oriented Software Environments'. *IEEE Trans. on Softw. Eng.* 14(6) pp758-773. 1988.

[BasS83] Basili, V.R. Selby, R.W. Phillips, T. 'Metric analysis and data validation across FORTRAN projects'. *IEEE Trans. on Softw. Eng.* 9(6) pp652-663. 1983.

[Bean84] Beane, J. Giddings, N. Silverman, J. 'Quantifying software designs'. *Proc. 7th Int Softw. Eng. Conf.* pp314-322. 1984.

[Behr83] Behrens, C.A. 'Measuring the Productivity of Computer Systems Development Activities with Function Points'. *IEEE Trans. on Softw. Eng.* 9(6) pp649-658. 1983.

[Bela79] Belady, L.A. 'On software complexity'. *IEEE Proc. Workshop on Quant. Softw. Models for Reliability, Complexity and Cost,* Oct. 1979 pp90-94. 1979.

[Bela81] Belady, L.A. Evangelisti, C.J. 'System partitioning and its measure'. *J. of Sys. & Softw.* 1(2), pp23-29. 1981.

[Beny79] Benyon-Tinker, G. 'Complexity measures in an evolving large system'. *Proc. ACM Workshop on Quant. Software Models.* pp117-127. 1979.

[Bese82] Beser, N. 'Foundations and experiments in Software Science'. *ACM SIGMETRICS PER* 11(3) pp48-72. 1982.

[Boeh81] Boehm, B. *'Software engineering economics'* Prentice-Hall, Englewood Cliffs, N.J.. 1981.

[Boeh84] Boehm, B.W. 'Software engineering economics'. *IEEE Trans. on Softw. Eng.* 10(1) pp4-21. 1984.

[Booc86] Booch, G. 'Object-oriented design' *IEEE Trans.on Softw. Eng.* 12(2) pp211-221. 1986.

[Bowe78] Bowen, J.B. 'Are current approaches sufficient for measuring Software quality?'. *Proc SIGMETRICS / SIGSOFT Software Qual. Ass. Workshop.* 1978.

[Bowe84] Bowen, J.B. 'Module size: a standard or heuristic?' *J. of Sys. & Softw.* 4, pp327-332. 1984.

[Bowl83] Bowles, A.J. 'Effects of design complexity on software maintenance'. Doctoral thesis Northwestern Uni. Evanston IL. 1983.

[Broo80] Brooks, R.E. 'Studying programmer behaviour experimentally: the problems of proper methodology'. *CACM* 23(4) pp207-213. 1980.

[Brow78] Brown, J.R. Fischer, K.F. 'A graph theoretic approach to the verification of program structures". *Proc. 3rd IEEE Intl. Conf. on Softw. Eng.*, pp136-141. 1978.

[Card86] Card, D.N. Church, V.E. Agresti, W.W. 'An Empirical Study of Software Design Practices'. *IEEE Trans. on Softw. Eng.* 12 pp264-271. 1986.

[Card87] Card, D.N. Agresti, W.W. 'Resolving the software science anomaly' *J. of Sys. & Softw.* 7, pp29-35. 1987.

[Card88] Card, D.N. Agresti, W.W. 'Measuring software design complexity' *J. of Sys. & Softw.* 8, pp185-197. 1988.

[Chan74] Channon, R.N. 'On a Measure of Program Structure', Doctoral Dissertation, Carnegie-Mellon Univ. 1974.

[Chap79] Chapin, N. 'A measure of software complexity' *Proc. NCC '79* pp995-1002. 1979.

[Chen78] Chen, E.T. 'Programmer complexity and programmer productivity'. *IEEE Trans. on Softw. Eng.* 4(3), pp187-194. 1978.

[Chri81] Christensen, K. Fitsos, G.P. Smith, C. 'A perspective on software science'. *IBM Sys. J.* 20(4) pp372-387. 1981.

[Chur59] Churchman, C.W. 'Why measure?' in Churchman, C.W. Ratoosh, P (eds.) *'Measurement: definitions and theories'*. Wiley, N.Y.. 1959.

[Come79] Comer, D. Halstead, M.H. 'A simple experiment in top-down design'. *IEEE Trans. on Softw. Eng.* 5(2) pp105-109. 1979.

[Cont82] Conte, S.D. Shen, V.Y. Dickey, K. 'On the effect of different counting rules for control flow operators on software science metrics in FORTRAN'. *ACM SIGMETRICS P.E.R.* 11(2) pp118-126. 1982.

[Cont86] Conte, S.D. Dunsmore, H.E. Shen, V.Y. *'Software engineering metrics and models'* Benjamin Cummings. 1986.

[Coul81] Coulter, N.S. 'Applications of psychology in software science'. Proc. COMPSAC '81 pp50-51. 1981.

[Coul83] Coulter, N.S. 'Software Science and cognitive pyschology '. *IEEE Trans. on Softw. Eng.* 9(2) pp166-171. 1983.

[Craw85] Crawford, S. McIntosh, A. Pregibon, D. 'An analysis of static metrics and faults in C software'. *J. of Syst.& Softw.* 5(1). 1985.

[Curt79a] Curtis, B. Sheppard, S. Milliman, P. Borst, M. Love, T. 'Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe Metrics'. *IEEE Trans. on Softw. Eng.* 5(2) pp96-104. 1979.

[Curt79b] Curtis, B. 'In search of software complexity'. *Proc. of Workshop on Quant. Softw. Complexity Models.* pp95-106. 1979.

[Curt79c] Curtis, B. Sheppard, S. Milliman, P. 'Third time charm; stronger prediction of programmer performance by software complexity metrics'. *Proc. 4th IEEE Intl. Conf. on Softw. Eng.* pp356-360. 1979.

[Curt80] Curtis, B. 'Measurement and experimentation in software engineering'. *Proc. IEEE* 68(9) pp1144-1157, 1980.

[Curt83] Curtis, B. 'Software metrics: guest editor's introduction". *IEEE Trans. on Softw. Eng.* 9(6) pp637-638. 1983.

[Dahl72] Dahl, O.J. Dijkstra, E.W. Hoare, C.A.R. *Structured Programming,* Academic Press, 1972.

[Davi88] Davis, J.S. LeBlanc, R.J. 'A study of the applicability of complexity measures'. *IEEE Trans. on Softw. Eng.* 14(9) pp1366-1372. 1988.

[deMa78] deMarco, T. *'Structured analysis and system specification'.* Yourdon Press. NY. 1978.

[deMa82] deMarco, T. *'Controlling software projects. Management, measurement and estimation'.* Yourdon Press. NY. 1982.

[Dijk68] Dijkstra, E.W. 'Goto statement considered harmful', *CACM,* 18(8) pp453-457, 1968.

[Dorf84] Dorfman, M. Flynn, R.F. 'ARTS - An automated requirements traceability system'. *J.of Syst. & Softw.* 4(4) pp63-74. 1984.

[Duns77] Dunsmore, H.E. Gannon,J.D. 'Experimental investigation of programming complexity'. *Proc. ACM/NBS 16th Annu. Tech. Symp. on Syst. Softw.,* Washington,DC. pp117-125. 1977.

[Duns79] Dunsmore, H.E. Gannon,J.D. 'An analysis of the effects of programming factors on programming effort'. *J. of Syst. & Softw.* 1(2) pp141-153. 1980.

[Elsh76] Elshoff, J.L. 'Measuring commercial programs using Halstead's criteria'. *ACM SIGPLAN Notices* 11(5). pp38-46. 1976.

[Elsh78] Elshoff, J.L. 'An investigation into the effects of the counting method used on software science measurements'. *ACM SIGPLAN Notices* 13(2) pp30-45. 1978.

[Emer84] Emerson, T.J. 'A discriminant metric for module cohesion'. *Proc. 7th Int. Conf. on Softw. Eng.* ,pp294-303. 1984.

[Endr75] Endres, A. 'An analysis of errors and their causes in system programs'. *IEEE Trans.on Softw.Eng.* 1(2). 1975.

[Evan83] Evangelist, W.M. 'Software complexity metric sensitivity to program structuring rules'. *J. Syst. & Softw.* 3(3) pp231-243. 1983.

[Evan84] Evangelist, W.M. 'Program complexity and programming style'. *Proc. IEEE Intl. Conf. on Data Eng.*, L.A.,CA, pp534-541. 1984.

[Fent85] Fenton, N.E. Whitty, R.W. Kaposi, A.A. 'A generalised mathematical theory of structured programming' *Theor. Comput. Sci.* 36, pp145-171, 1985.

[Fent86] Fenton, N.E. Whitty, R.W. 'Axiomatic approach to Software metrification through program decomposition'. *Computer J.* 29(4) pp330-340. 1986.

[Fent87a] Fenton, N.E. Kaposi, A.A. 'An engineering theory of structure and measurement'. *Proc. Centre for Softw.Reliability, Conf.* Bristol. England. Sept.1987.

[Fent87b] Fenton, N.E. Kaposi, A.A. 'Metrics and software structure'. *Info. & Softw. Technol.* 29(6) pp301-320, 1987.

[Fent90] Fenton, N.E. 'Software measurement: theory, tools and validation' *Softw. Eng. J.* 6(1), 1990.

[Feue79] Feuer, A.R. Fowlkes, E.B. 'Some results from an empirical study of computer software'. *Proc. 4th IEEE Intl. Conf. on Softw. Eng.*. pp351-355. 1979.

[Fink84] Finkelstein, L. Leaning, M.S. 'A review of the fundamental concepts of measurement'. *Measurement*, 2(1) pp25-34. 1984.

[Fits80] Fitsos,G. 'Vocabulary effects in software science'. *Proc. COMPSAC 80*, pp751-756. 1980.

[Fitz78a] Fitzsimmons, A. 'Relating the presence of Software errors to the theory of Software Science'. *11th Hawaii Int. Conf. on Systems Sci.*. vol. 1 pp40-46. 1978.

[Fitz78b] Fitzsimmons, A. Love, T. 'A review and evaluation of software sciences' ACM Computing Surveys, 10, pp3-18. 1978.

[Funa76] Funami, Y. Halstead, M.H. 'A software physics analysis of Akiyama's debugging data'. *Proc. Symp. on Computer Softw. Eng.* pp133-138. N.Y. Poly. Inst.. 1976.

[Gaff79] Gaffney, J.E. 'Program control, complexity and productivity'. *Proc. IEEE Workshop on Quant. Softw. Models for Reliability*, pp140-142. 1979.

[Gaff84] Gaffney, J.E. 'Estimating the number of faults in code' *IEEE Trans. on Softw. Eng.* 10(4) pp459-464, 1984.

[Gann86] Gannon, J.D. Katz, E.E. Basili, V.R. 'Metrics for Ada packages: an initial study'. *CACM* 29(7), 1986.

[Geha82] Gehani, N.H. 'Specifications formal and informal - a case study', *Softw. Pract. & Experience*, **12**, pp433-444, 1982.

[Geri77] Geritsen, R. Morgan, H. Zisman, M. 'On some metrics for databases or what is a very large database?' *ACM SIGMOD Record* pp50-74, June 1977.

[Gibs89] Gibson, V.R. Senn, J.A. 'System structure and software maintenance performance'. *CACM* 32(3), pp347-358, 1989.

[Gidd84] Giddings, N. Colburn, T. 'An automated software design evaluater(sic)'. *Proc. Annu. Conf. - ACM 84*, San Francisco, CA. 1984.

[Gilb88] Gilb, T. *Principles of software engineering management*, Addison-Wesley, 1988.

[Gord76] Gordon, R.D. Halstead, M.H. 'An experiment comparing Fortran programming times with the Software Physics Hypothesis'. *Proc. AFIPS* pp935-937. 1976.

[Gord79] Gordon, R.D. 'Measuring improvements in program clarity'. *IEEE Trans.on Softw. Eng.* 5(2) pp79-90. 1979.

[Goul75] Gould, J.D. 'Some psychological evidence on how people debug computer programs'. *Int. J. of Man-Machine Studies* 7. 1975.

[Gutt77] Guttag, J.V. 'Abstract data types and the development of data structures'. *CACM* 20(6) pp397-404. 1977.

[Hall84] Hall, N.R. Preiser, S. 'Combined network complexity measures'. *IBM J. of R. & D.* 23(1) pp15-27. 1984.

[Hals72] Halstead, M.H. Natural laws controlling algorithmic structure' *ACM SIGPLAN Notices* 7(2) pp19-26.1972.

[Hals77] Halstead, M.H. *'Elements of Software science'.* Elsevier North-Holland NY. 1977.

[Hals79a] Halstead, M.H. 'Advances in software science' in *Advances in Computers*, Vol. 18. Ed. Yovits, M.. Academic Press. NY. 1979.

[Hals79b] Halstead, M.H. 'Guest editorial on software science'. *IEEE Trans.on Softw. Eng.* 5(2) pp74-75. 1979.

[Hame82] Hamer, P.G. Frewin, G.D. 'M.H. Halstead's Software Science - A Critical Examination'. *Proc. IEEE 6th Int. Conf on Softw. Eng.* pp197-206. 1982.

[Hane72] Haney, F.M. 'Module connection analysis - a tool for scheduling software debugging activities'. *Proc. AFIP Fall Joint Conf.* pp173-179. 1972.

[Hans78] Hansen, W.J. 'Measurement of program complexity by the pair (Cylomatic Number, Operator Count)'. *ACM SIGPLAN Notices* 13(3) pp29-33. 1978.

[Harr81] Harrison, W. Magel, K. 'A complexity measure based on nesting level'. *ACM SIGPLAN Notices* 16(3) pp63-74. 1981.

[Hart82] Hartman, S. 'A counting tool for RPG'. *ACM SIGMETRICS P.E.R.* 11,Fall, pp86-100. 1982.

[Henr79] Henry, S. 'Information flow metrics for the evaluation of operating systems' structure'. PhD thesis, Iowa State Univ.. 1979.

[Henr81a] Henry, S. Kafura, D. 'Software metrics based on information flow.' *IEEE Trans. on Softw. Eng.* 7(5) pp510-518. 1981.

[Henr81b] Henry, S. Kafura, D. Harris, K. 'On the relationship among three software metrics' *ACM SIGMETRICS Performance Evaluation Review* 10, Spring pp81-88. 1981.

[Henr84] Henry, S. Kafura, D. 'The evaluation of software systems' structure using quantitative software metrics'. *Softw. Pract & Expr.* 14(6) pp561-573. 1984.

[Hoar78] Hoare, C.A.R. 'Communicating Sequential Processes', *C.A.C.M.*, 1978.

[Hump89] Humphrey, W.S. *Managing the Software Process*, Addison-Wesley, 1989.

[Hutc85] Hutchens, D.H. Basili, V.R. 'System structure analysis: clustering with data bindings'. *IEEE Trans. on Softw. Eng.* 1198) pp749-757. 1985.

[Ince84] Ince, D.C. 'Module interconnection language and Prolog'. *ACM SIGPLAN Notices* 19(8) pp89-93. 1984.

[Ince86] Ince, D.C. Hekmatpour, S. 'The peturbational analysis of software designs.' *Proc. Phoenix Conf. on Comp. & Comms.* pp462-464. 1986.

[Ince88a] Ince, D.C. Hekmatpour, S. 'An approach to automated software design based on product metrics'. *Softw. Eng. J.* 3(2) pp53-56. 1988.

[Ince88b] Ince, D.C. Shepperd, M.J . 'System design metrics: a review and perspective.' *Proc. IEE / BCS Conf. Software Engineering '88* July 12- 15, Liverpool University, pp23-27. 1988.

[Ince89a] Ince, D.C. Shepperd, M.J. 'An empirical and theoretical analysis of an information flow based design metric'. *Proc. European Software Eng. Conf.*, Warwick, England. Sept. 12-15, 1989.

[Ince89b] Ince, D.C. Shepperd, M.J. 'Quality control of software designs using cluster analysis'. *Proc. EOQC/SQA Conf. Management of quality: key to the nineties.* Vienna,

[Ince90a] Ince, D.C. Shepperd, M.J. 'The use of cluster techniques and system design metrics in software maintenance'. *Proc. IEE/DTI UK IT'90 Conf.*, Southampton, UK, March 1990.

[Ince90b] Ince, D.C. Shepperd, M.J. 'Metricating data oriented notations'. Open University Tech. Rep. May 1990.

[Ince90c] Ince, D.C. Shepperd, M.J. 'The measurement of data design', *Tech. Rep. 90/05*, School of Computing & IT, Wolverhampton Polytechnic, May 1990.

[Iyen82] Iyengar, S.S. Parameswaran, N. Fuller, J. 'A measure of logical complexity of programs'. *Computer Langs.* 7 pp147-160. 1982.

[Jack75] Jackson, M.A. *'Principles of program design.'* Academic Press, New York. 1975.

[Jack82] Jackson, M.A. *'System development'*. Prentice-Hall, Englewood-Cliffs, NJ. 1982.

[John81] Johnston,D.B. Lister, A.M. 'A note on the software science length equation'. *Softw. Pract.& Experience*, 11(8). 1981.

[Jone86] Jones, C.B. *Systematic Software Development using VDM*, Prentice-Hall, 1986.

[Kafu84] Kafura, D. 'The independence of software metrics taken at different life-cycle stages'. *Proc. 9th Annu. Softw.Eng. Workshop NASA/GSFC.* 1984.

[KafC84] Kafura, D. Canning, J.T. Reddy, G.R. 'The independence of software metrics taken at different life-cycle stages'. *Proc. 9th Ann. Softw. Eng. Workshop,* Chicago,IL. pp213-222. 1984.

[KafC85] Kafura, D. Canning, J.T. 'A validation of software metrics using many metrics and two resources.' *Proc. 8th Int. Conf. Softw. Eng.* London, England pp378-385. 1985.

[KafH81] Kafura, D. Henry, S. 'Software quality metrics based on interconnectivity', *J. of Sys. & Softw.* 2 pp121-131. 1981.

[KafR87] Kafura, D. Reddy, G.R. 'The use of Software Complexity Metrics in Software Maintenance'. *IEEE Trans. on Softw. Eng.* 13(3) pp335-343. 1987.

[Kapo90] Kaposi, A.A. Myers, M. 'Quality assuring specification and design'. *Softw. Eng. J.* 5(1) pp11-26, 1990.

[Kari88] Karimi, J. Konsynski, B.R. 'An automated software design assistant'. *IEEE Trans. on Softw. Eng.* 14(2) pp194-210, 1988.

[Kear86] Kearney, J.K. Sedlmeyer, R.L. Thompson, W.B. Gray, M.A. Adler, M.A. 'Software complexity measurement'. *CACM* 29(11) pp1044-1050. 1986.

[Keme87] Kemerer, C.F. 'An empirical validation of software cost estimation models'. *CACM* 30(5) pp416-429. 1987.

[Kern78] Kernighan, B. Plauger, P. *'The Elements of Programming Style'.* 2nd ed., McGraw-Hill. 1978.

[Kitc81] Kitchenham, B.A. 'Measures of programming complexity'. *ICL Tech. J.* May '81 pp298-316. 1981.

[Kitc86] Kitchenham, B.A. McDermid, J.A. 'Software metrics and integrated project support environments'. *Softw. Eng. J.* 1(1) pp58-64. 1986.

[Kitc87a] Kitchenham, B.A. 'Towards a constructive quality model. Part I: Software quality modelling, measurement and prediction'. *Softw. Eng. J.* 2(4) pp105-113. 1987.

[Kitc87b] Kitchenham, B.A. Pickard, L 'Towards a constructive quality model. Part II: Statistical techniques for modelling software quality in the ESPRIT REQUEST project.'. *Softw. Eng. J.* 2(4) pp114-126. 1987.

[Kitc88] Kitchenham, B.A. 'An evaluation of software structure metrics.' *Proc. COMPSAC '88*, Chicago, IL. 1988.

[Kitc89] Kitchenham, B.A. Linkman, S.J. 'Design metrics in practice', *Proc. NDISD'89*, pp12-39 School of Computing & I.T. Wolverhampton Polytechnic, England (also to be published *Info. & Softw. Tech.* May, 1990).

[Kitc90] Kitchenham, B.A. Pickard, L.M. Linkman, S.J. 'An evaluation of some design metrics'. *Softw. Eng. J.* 5(1) pp50-58, 1990.

[Knij78] van der Knijff, D.J.J. 'Software physics and program analysis'. *Australian Computer J.* 10 pp82-86. 1978.

[Kran71] Krantz, D.H. Luce, R.D. Suppes, P. Tversky, A. *'Foundations of measurement'*. Academic Press, London. 1971.

[Koko89] Kokol, P. 'Using spreadsheet software to support metric life cycle activities', *ACM SIGPLAN Notices.* 24(5) pp27-37, 1989.

[Kons85] Konstam, A.H. Wood, D.E. 'Software science applied to APL'. *IEEE Trans.on Softw.Eng.* 11(10) pp994-1000. 1985.

[Kott83] Kottemann, J.E. Konsynski, B.R. 'Complexity assessment: A design and management tool for information system development'. *Inform. Syst.* 8(3) pp195-206. 1983.

[Kybe84] Kyburg, H.E. *'Theory and measurement'*. Cambridge Univ. Press, Cambridge, England. 1984.

[Laka70] Lakatos, I. 'Falsification and the methodology of scientific research programs', in Lakatos, I Musgrave, A. (eds.) *'Criticism and the growth of knowledge'*. C.U.P. Cambridge, England. 1970.

[Lass81] Lassez, J-L. van der Knijff, D.J.J. Shepherd, J. Lassez, C. 'A critical examination of software science'. *J. of Syst. & Softw.* 2, pp105-112. 1981.

[Laur82] Laurmaa, T. Syrjanen, M. 'APL and Halstead's theory: a measuring tool and some experiments'. *ACM SIGMETRICS P.E.R.* 11, Fall, pp32-47. 1982.

[Lehm76] Lehman, M.M. Belady, L.A. 'A model of large system development' *IBM Sys. J.* 15(3) pp225-252. 1976.

[Lind89] Lind, R.K. Vairavan, K. 'An experimental investigation of software metrics and their relationship to software development effort'. *IEEE Trans.on Softw.Eng.* 15(5) pp649-653. 1989.

[Link91] Linkman, S. Walker, J. 'Maintenance Metrics (And how to avoid using them by controlling development programmes through measurement)', *Info & Softw. Tech.* (To be published Jan/Feb 1991).

[Lisk86] Liskov, B. Guttag, J. *'Abstraction and specification in program development'.* MIT Press, MA.. 1986.

[List82] Lister, A.M. 'Software science - The emperor's new clothes?' *Australian Comput. J.* 14(2) pp66-71. 1982.

[Lohs84] Lohse, J.B. Zweben, S.H. 'Experimental evaluation of software design principles: an investigation into the effect of modular coupling on system and modifiability'. *J.of Sys. & Softw.* 4(4) pp301-308. 1984.

[Lond87] Londeix, B. *'Cost estimation for software development'* Addison-Wesley. 1987.

[Love76] Love, L.T. Bowman, A.B. 'An independent test of the theory of Software Physics'. *ACM SIGPLAN Notices* 12(11) pp42-49. 1976.

[Love77] Love, L.T. 'An experimental investigation of the effect of program structure on program understanding'. *ACM SIGPLAN Notices* 12(3) pp105-113. 1977.

[Low90a] Low, G.C. Jeffery, D.R. 'Function points in the estimation and evaluation of the software process'. *IEEE Trans.on Softw.Eng.* 16(1) pp64-71. 1990.

[Low90b] Low, G.C. Jeffery, D.R. 'Calibrating estimation tools for software development'. *Softw. Eng. J.* 5(14) pp215-221, 1990.

[Luce56] Luce, R.D. 'Semi-orders and a theory of utility discrimination' *Econometrica* 24, pp178-191. 1956.

[Luce69] Luce, R.D. Marley, A.A.J. 'Extensive measurement when concatenation is restricted and maximal elements may exist'. In Morgenbeser, S. Suppes, P. White, M.G. (eds.), *Philosophy, science and method: essays in honour of Ernest Nagel.* St.Martin's Press, N.Y.. 1969.

[Mage81] Magel, K. 'Regular expressions in a program complexity metric'. *ACM SIGPLAN Notices* 16(7) pp61-65. 1981.

[McCa76] McCabe, T.J. 'A complexity measure' *IEEE Trans. on Softw. Eng.* 2(4) pp308-320. 1976.

[McCa82] McCabe, T.J. 'Structured testing: a testing methodology using the McCabe complexity metric', *Natl. Bureau of Standards,* NB82NAAK55181, 1982.

[McCa89] McCabe, T.J. Butler, C.W. 'Design complexity Measurement and testing' *CACM* 32(12) pp1415-1425. 1989.

[McCl78] McClure, C.L. 'A model for program complexity analysis', *Proc. 3rd Intl. Conf. on Softw. Eng.,* pp149-157, 1978.

[Melt90] Melton, A.C. Gustafson, D.A. Bieman, J.A. Baker, J.A. 'A mathematical perspective for software measures research', *Softw. Eng. J.* 5(4) pp246-254, 1990.

[Miln89] Milner, R. *Communication and Concurrency,* Prentice-Hall, 1989.

[Moha81] Mohanty, S.N. 'Software cost estimation: present and future'. *Softw. Pract. & Experience* 11, pp103-121. 1981.

[Muns90] Munson, J.C. Khoshgoftaar, T.M. 'The relative software complexity metric: a validation study', *Proc. BCS/IEE Software Engineering '90 Conf.,* Brighton, UK, 1990.

[Myer75] Myers, G.J. *'Reliable Software Through Composite Design'.* Van Nostrand Reinhold NY. 1975.

[Myer77] Myers, G.J. 'An extension to the the cyclomatic measure of program complexity'. *ACM SIGPLAN Notices* 12(10) pp61-64. 1977.

[Naur69] Naur, P. Randell, B. (Eds.) *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee,* NATO, 1969.

[Navl87] Navlakha, J.K. 'A survey of system complexity metrics', *The Comp. J.* 30(3), pp233-238, 1987.

[NCC86] *SSADM Manual,* NCC, Manchester, England, 1986.

[Otte76] Ottenstein, K. J. 'An algorithmic approach to the detection and prevention of plagiarism'. *ACM SIGCSE Bull.* 8(4) pp30-41. 1976.

[Otte79] Ottenstein, L.M. 'Quantitative estimates of debugging requirements'. *IEEE Trans. on Softw. Eng.* 5(5) pp504-514. 1979.

[Ouls79] Oulsnam, G. 'Cyclomatic numbers do not measure complexity of unstructured programs'. *Info. Proc. Letts.* Dec. 1979 pp207-211. 1979.

[Ovie80] Oviedo, E. 'Control flow, data flow and program complexity'. *Proc. COMPSAC 80* pp146-152. 1980.

[Ovie83] Oviedo, E. Ralston, A. 'An environment to develop and validate program complexity measures'. Proc. IEEE Natl. Educational Computing Conf., pp115-121, Baltimore, June 6-8, 1983.

[Paig80] Paige, M.R. 'A metric for software test planning'. *Proc. COMPSAC '80* pp499-504. 1980.

[Parn72] Parnas, D.L. 'On the criteria to be used in decomposing systems into modules'. CACM 15(2) pp1053-1058.

[Parn79] Parnas, D.L. 'Designing software for ease of extension and contraction'. *IEEE Trans. on Softw. Eng.* 5(2) pp128-138. 1979.

[Pfan82] Pfanzagl, J. *'Theory of measurement'*. Physica-Verlag, Wurzburg-Vienna. 1968.

[Piow82] Piowarski, P. 'A nesting level complexity measure'. *ACM SIGPLAN Notices* 17(9) pp40-50. 1982.

[Poti81] Potier, D. Ferreol, A.R. Bilodeau, A. 'Experiments with computer software complexity and reliability'. *Proc. 6th IEEE Intl. Conf.on Softw. Eng.*, pp94-103. 1981.

[Prat84] Prather, R.E. 'An axiomatic theory of software complexity metrics'. *The Comp. J.* 27(4) pp340-347. 1984.

[Prat87] Prather, R.E. 'On hierarchical software metrics'. *Softw. Eng. J.* 2(2) pp42-45. 1987.

[Prat88] Prather, R.E. 'Comparison and extension of theories of Zipf and Halstead'. *The Comp. J.* 31(3) pp248-252. 1988.

[Pres87] Pressman,R.S. *'Software engineering. A practitioner's approach'*. McGraw-Hill, 2nd. Edn.. 1987.

[Reyn84] Reynolds, R.G. 'Metrics to measure the complexity of partial programs'. *J. of Syst. & Softw* 4(1) pp75-92. 1984.

[Rodr87] Rodriguez, V. Tsai, W.T. 'A tool for discriminant analysis and classification of software metrics'. *Info. & Softw. Tech.* 29(3) pp137-150. 1987.

[Romb87a] Rombach, H.D. 'A controlled experiment on the impact of software structure on maintainability.' *IEEE Trans. on Softw. Eng.* 7(5) pp510-518. 1987.

[Romb87b] Rombach, H.D. Basili, V.R. 'A quantitative assessment of software maintenance'. *Proc. Conf. on Softw. Maint., Austin,TX* Sept.'87 pp134-144. 1987.

[Romb89] Rombach, H.D. Ulery, B.T. 'Improving software maintenance through measurement', *IEEE Proc*, 1989.

[Royc70] Royce, W.W. 'Managing the development of large software systems', *Proc. WESTCON*, Calif., USA, 1970.

[Salt82] Salt, N. 'Defining software science counting strategies'. *ACM SIGPLAN Notices* 17(3) pp58-67. 1982.

[Sams87] Samson, W.B. Nevill, D.G. Dugard, P.I. 'Predictive software metrics based on a formal specification'. *Information & Software Technology* 29(5) pp242-248. 1987.

[Schn88] Schneider, V. 'Approximations for the Halstead software science error rate and project estimators'. *ACM SIGPLAN Notices* 23(1) pp40-47. 1988.

[Selb88] Selby, R.W. Basili, V.R. 'Error localization during maintenance: generating hierarchical system descriptions from source code alone', *Proc. IEEE Conf. on Softw. Maint.*, 1988.

[Shaw89] Shaw, W.H. Howatt, J.W. Maness,R.S. Miller, D.M. 'A software science model of compile time'. *IEEE Trans. on Softw. Eng.* 15(5) pp543-551. 1989.

[Shen79] Shen, V.Y. 'The relationship between student grades and software science parameters'. *Proc. COMPSAC '79.* pp783-787. 1979.

[Shen83] Shen, V.Y. Conte, S.D. Dunsmore, H.E. 'Software science revisited: a critical analysis of the theory and its empirical support'. *IEEE Trans. on Softw. Eng.* 9(2) pp155-165. 1983.

[Shen85] Shen, V.Y. Yu, T.J. Thebaut, S.M. Paulsen, L.R. 'Identifying error-prone software - an empirical study'. *IEEE Trans. on Softw. Eng.* 11(4) pp317-324. 1985.

[Shep88a] Shepperd, M.J. 'A critique of cyclomatic complexity as a software metric' *Softw. Eng. J.* 3(2) pp30-36. 1988.

[Shep88b] Shepperd, M.J. 'An evaluation of software product metrics.' *Information & Softw. Tech.* 30(3) pp177-188. 1988.

[Shep88c] Shepperd, M.J. 'A preliminary investigation into the relationship between software maintainability and design metrics'. *The Polytechnic: Wolverhampton, School of Computing and Information Technology, Technical Report 88/09.* 1988.

[Shep88d] Shepperd, M.J. 'An empirical study of design measurement: an interim report'.*The Polytechnic: Wolverhampton, School of Computing and Information Technology, Technical Report 88/08.* 1988 (To be published SEJ Jan. 1990).

[Shep89a] Shepperd, M.J. Ince, D.C. 'Metrics, outlier analysis and the software design process'. *Information & Softw. Tech.* 31(2) pp91-98.1989.

[Shep89c] Shepperd, M.J. 'A metrics based tool for software design' *Proc. 2nd Int. Conf. on Softw. Eng. for Real Time Systems*, The Royal Agriculture College, Cirencester, UK. Sept. 18-20, 1989.

[Shep89d] Shepperd, M.J. 'Specification: a new perspective on design metrics'.*The Polytechnic: Wolverhampton, School of Computing and Information Technology, Technical Report 89/01.* (Also accepted for publication *The Computer J.*). 1989.

[Shep90a] Shepperd, M.J. 'An empirical study of design measurement'. *The Softw. Eng. J.* Jan. 1990.

[Shep90b] Shepperd, M.J. Ince, D.C. 'The multi-dimensional modelling and measurement of software designs'. *Proc. Annu. ACM Comp. Sci. Conf.*, Washington DC, Feb.20-22, 1990.

[Shep90c] Shepperd, M.J. 'Early life cycle metrics and software quality models', *Information & Softw. Tech.* 32(4) pp311-316, 1989.

[Shep90d] Shepperd, M.J. 'Metrics for the software engineer: a review' In *'The Software Life Cycle'* Eds. Ince, D.C. Andrews, D. Butterworth Scientific, 1990.

[Shep90e] Shepperd, M.J. Ince, D.C. 'Controlling software maintainability', *Proc. 2nd European Conf. on Softw. Quality Assurance*, Oslo, Norway, 1990.

[Shep90f] Shepperd, M.J. 'The use of metrics for the early detection of software design errors'. *Proc. BCS/IEE Software Engineering 1990*, Brighton, July 24-27, 1990.

[Silv83] Silverman, J. Giddings, N. Beane, J. 'An approach to design-for-maintenance'. *Proc. Softw. Maint. Workshop*, Monterey,CA. 1983.

[Simo80] Simon, H. *'Science of the artificial'* MIT Press. 1980.

[Sinh86] Sinha, P.K. Jayaprakash, S. Lakshmanan, K.B. 'A new look at the control flow complexity of computer programs'. *Proc. BCS/IEE Conf. SE '86* 10-12 Sept. pp88-102. 1986.

[Smit79] Smith, C.P. 'Practical applications of software science'. *IBM Santa Teresa Lab., Tech. Rep.* 03.067, June 1979.

[Smit80] Smith, C.P. 'A software science analysis of programming size'. *Proc. ACM. Nat. Comput. Conf.* pp179-185. October 1980.

[Somm89] Sommerville, I. *'Software engineering'*. Addison-Wesley. 3rd Edn. 1989.

[Soon77] Soong, N.L. 'A program stability measure'. *Proc. ACM Annual Conf.* pp163-173. 1977.

[Stet84] Stetter,F. 'A measure of program complexity'. *Computer Langs.* 9(3) pp203-210. 1984.

[Stev46] Stevens, S.S. 'On the theory of scales of measurement'. *Science* 103, pp677-680. 1946.

[Stev59] Stevens, S.S. 'Measurement, psychophysics and utility' in Churchman, C.W. Ratoosh, P (eds.) *'Measurement: definitions and theories'*. Wiley, N.Y.. 1959.

[Stev74] Stevens, W.P. Myers, G.J. Constantine, L.L. 'Structured design' *IBM Sys. J.* 13(2) pp115-139. 1974.

[Stev80] Stevens, W.P. *'Structured design'*. Academic Press. 1980.

[Stro66] Stroud, J.M. 'The Fine Structure of Psychological Time'. *Annals of the New York Academy of Sciences*, pp623-631. 1966

[Suno81] Sunohara, T. Takano, A. Vehara, K. Ohkawa, T. 'Program complexity measure for software development management'. *Proc. 5th IEEE Intl. Conf. on Softw. Eng.*, pp100-106. 1981.

[Supp59] Suppes, P. 'Measurement,empirical meaningfulness and three-valued logic'. in Churchman, C.W. Ratoosh, P (eds.) *'Measurement: definitions and theories'*. Wiley, N.Y.. 1959.

[Supp71] Suppes, P. Zinnes, J.L. 'Basic measurement theory'. In Lieberman, B. (ed.) *'Contemporary problems in statistics'* O.U.P. 1971.

[Symo88] Symons, C.R. 'Function point analysis: difficulties and improvements'. *IEEE Trans.on Softw. Eng.* 14(1) pp2-11. 1988.

[Szul81] Szulewski, P.A. Whitworth, M.H. Buchan, P. DeWolf, J.B. 'The measurement of software science parameters in software designs'. *ACM SIGMETRICS PER* Spring '81, pp89-94. 1981.

[Taus81] Tausworthe, R.C. 'Deep space network software cost estimation model'. *Tech. Rep. 81-7, Jet Propulsion Lab.*, Pasadena, CA. 1981.

[Thay78] Thayer, T.A. Lipow, M. Nelson, E.C. *'Software reliability'*. North-Holland, NY. 1978.

[Troy81] Troy, D.A. Zweben, S.H. 'Measuring the quality of structured designs'. *J. of Syst. & Softw.* 2(2) pp113-120. 1981.

[Wals79] Walsh, T.A. 'A software reliability study using a complexity measure'. *Proc. National Computer Conf.*,pp761-768. 1979.

[Wang84a] Wang, A.S. Dunsmore, H.E. 'Back-to-front programming effort prediction'. *Info Proc & Mngt* 20(1-2) pp139-149. 1984.

[Wang84b] Wang, A.S. *The estimation of software size and effort: an approach based on the evolution of software metrics*, PhD Thesis, Dept. of Comp. Sci., Purdue University, 1984.

[Weis74] Weissman, L. 'Pyschological complexity of computer programs: an experimental methodology'. *ACM SIGPLAN Notices* 9(6) pp25-36. 1974.

[Weyu88] Weyuker, E.J. 'Evaluating software complexity measures'. *IEEE Trans. on Softw. Eng.* 14(9) pp1357-1365. 1988.

[Whit85] Whitty, R.W. Fenton, N.E. Kaposi, A.A. 'A rigorous approach to structural analysis and metrication of software' *IEE Softw. & Microsystems* 4(1) pp2-16. 1985.

[Wien84] Wiener,R. Sincovec,R. *'Software engineering with Modula-2 and Ada'.* Wiley. 1984.

[Will78] Willis, R.R. 'DAS - an automated system to support design analysis', *Proc. 3rd Int. Conf. on Softw. Eng.* Atlanta GA, pp109-115. 1978.

[Will79] Willis, R.R. Jensen, E.P. 'Computer aided design of software systems'. *Proc. 4th Int. Conf. on Softw. Eng.* Munich. 1979.

[Wirt76] Wirth, N. *Systematic Programming, An Introduction,* Prentice-Hall, 1976.

[Wood79] Woodward, M.R. Hennell, Ma.A. Hedley, D.A. 'A measure of control flow complexity in program text'. *IEEE Trans. on Softw. Eng.* 5(1) pp45-50. 1979.

[Wood80] Woodfield, S.N. 'Enhanced effort estimation by extending basic programming models to include modularity factors'. PhD dissertation, Dept. Computer Sci., Purdue Univ., IN.. 1980.

[Wood81a] Woodfield, S.N. Shen, V.Y. Dunsmore, H.E. 'A study of several metrics for programming effort'. *J. of Syst. & Softw.* 2, pp139-149. 1981.

[Wood81b] Woodfield, S.N. Dunsmore, H.E. Shen, V.Y. 'The effect of modularisation and comments on program comprehension'. *Proc. 5th IEEE Intl. Conf. on Softw. Eng.,* pp215-223. 1981.

[Yau78] Yau, S.S. Collofello, J.S. MacGregor, T.M. 'Ripple Effect Analysis of Software Maintenance'. *Proc. COMPSAC '78* pp60-65. 1978.

[Yau80] Yau, S.S. Collofello, J.S. 'Some stability measures for software maintenance'. *IEEE Trans. on Softw. Eng.* 6(6) pp545-552. 1980.

[Yin78] Yin, B.H. Winchester, J.W. 'The establishment and use of measures to evaluate the quality of software designs' *Proc. ACM Softw. Qual. Ass. Workshop* pp45-52. 1978.

[Your79] Yourdon, E. Constantine,L.L. *'Structured design: Fundamentals of a discipline of computer program and systems design'.* Prentice-Hall, Englewood Cliffs, N.J.. 1979.

[Zoln81] Zolnowski, J.C. Simmons, D.B. 'Taking the measure of program complexity'. *Proc. National Computer Conf.*,pp329-336. 1981.

[Zweb79] Zweben, S.H. Fung, K. 'Exploring software science relations in COBOL and APL'. *Proc. COMPSAC '79.* pp702-709. 1979.

[Zuse89] Zuse, H. Bollmann, P. 'Software metrics: using measurement theory to describe the properties and scales of static complexity metrics'. *ACM SIGPLAN Notices* 24(8), pp23-33, 1989.

Appendices

Appendix A:

An Algebraic Specification of the Metrics IF4 and $IF4_m$

"Appendix is a euphism for a repository for write-only documentation."

Anon.

The following is the complete algebraic specification for the information flow metrics IF4 and IF4m that comprise part of the uni-dimensional model of system structure and the software quality factors of implementability, reliability and maintainability, given in Chapter Five.

TYPES

graph
node_type = (module , global_data_structure)
name = string

VARS

S_1 , S_2 : graph
t_1 : node_type
n_1 , n_2 , n_3 , n_4 : name

external operations

add_node: graph × node_type × name → graph ∪ {error}
add_edge: graph × name × name → graph ∪ {error}

IF4: graph → nat
IF4m: graph × name → nat ∪ {error}

#links: graph → nat
#modules:graph → nat
#ds: graph → nat

internal operations

concat_n: graph × node_type × name → graph
concat_e: graph × name × name → graph
new: → graph
exists: graph × name → boolean

linked: graph × name × name → boolean
is_a_module: graph × name → boolean
is_a_ds: graph × name → boolean

if4_int: graph × graph → nat

fan_in_l: graph × name → nat
fan_out_l: graph × name → nat
fan_in_g: graph × graph × name → nat
fan_out_g: graph × graph × name → nat
ct_globals_in: graph × graph × name × name → nat

SEMANTICS

1. add_node(S_1,t_1,n_1) = **if** exists(S_1,n_1)
then {error}
else concat_n(S_1,t_1,n_1)

2. add_edge(S_1,n_1,n_2) = **if** exists(S_1,n_1) ∧ exists(S_1,n_2)
∧ ¬ linked(S_1,n_1,n_2)
∧ n_1<>n_2
∧ is_a_module(S_1,n_1) ∨ is_a_module(S_1,n_2))
then concat_e(S_1,t_1,n_1)
else {error}

3. exists(new,n_1) = FALSE
4. exists(concat_n(S_1,t_1,n_2),n_1) = **if** n_1=n_2 **then** TRUE
else exists(S_1,n_1)
5. exists(concat_e(S_1,n_2,n_3),n_1) = **if** n_1=n_2 ∨ n_1=n_3 **then** TRUE
else exists(S_1,n_1)

6. linked(new,n_1,n_2) = FALSE
7. linked(concat_e(S_1,n_3,n_4),n_1,n_2) = **if** n_3=n_1 ∧ n_4=n_2
then TRUE
else linked(S_1,n_1,n_2)
8. linked(concat_n(S_1,t_1,n_3),n_1,n_2) = linked(S_1,n_1,n_2)

9. is_a_module(new,n_1) = FALSE

10.is_a_module(concat_n(S_1,t_1,n_2),n_1) = if n_1=n_2 ∧ t_1=module
then TRUE
else is_a_module(S_1,n_1)

11.is_a_module(concat_e(S_1,n_2,n_3),n_1) = is_a_module(S_1,n_1)

12.is_a_ds(new,n_1) = FALSE

13.is_a_ds(concat_n(S_1,t_1,n_2),n_1) = if n_1=n_2 ∧ t_1=global_data_structure
then TRUE
else is_a_ds(S_1,n_1)

14.is_a_ds(concat_e(S_1,n_2,n_3),n_1) = is_a_ds(S_1,n_1)

15.if4(S_1) = if4_int(S_1,S_1)

16.if4_int(S_1,new) = 0

17.if4_int(S_1,concat_n(S_2,t_1,n_1)) = if t_1=module
then if4m(S_1,n_1) + if4_int(S_1,S_2)
else if4(S_1,S_2)

18.if4_int(S_1,concat-e(S_2,n_1,n_2)) = if4(S_1,S_2)

19.if4m(S_1,n_1) = if is_a_module(S_1,n_1)
then sqr((fan_in_l(S_1,n_1) + fan_in_g(S_1,S_1,n_1)) *
(fan_out_l(S_1,n_1) + fan_out_g(S_1,S_1,n_1)))
else {error}

20.fan_in_l(new,n_1) = 0

21.fan_in_l(concat_e(S_1,n_2,n_3),n_1) = if n_3=n_1 ∧ is_a_module(n_2)
then 1 + fan_in_l(S_1,n_1)
else fan_in_l(S_1,n_1)

22.fan_in_l(concat_n(S_1,t_1,n_2),n_1) = fan_in_l(S_1,n_1)

23.fan_out_l(new,n_1) = 0

24.fan_out_l(concat_e(S_1,n_2,n_3),n_1) = if n_2=n_1 ∧ is_a_module(n_3)
then 1 + fan_out_l(S_1,n_1)
else fan_out_l(S_1,n_1)

25.fan_out_l(concat_n(S_1,t_1,n_2),n_1) = fan_out_l(S_1,n_1)

26.fan_in_g(S_1,new,n_1) = 0

27.fan_in_g(S_1,concat_e(S_2,n_2,n_3),n_1) = if n_3=n_1 ∧ is_a_ds(n_2)
then ct_globals_in(S_1,S_2,n_1,n_2) +
fan_in_g(S_1,S_2,n_1)
else fan_in_g(S_1,S_2,n_1)

28.fan_in_g(S_1,concat_n(S_2,t_1,n_2),n_1) = fan_in_g(S_1,S_2,n_1)

29.fan_out_g(S_1,new,n_1) = 0

30.fan_out_g(S_1,concat_e(S_2,n_2,n_3),n_1) = if n_3=n_1 ∧ is_a_ds(n_2)
then ct_globals_out(S_1,S_2,n_1,n_2)
+ fan_out_g(S_1,S_2,n_1)
else fan_out_g(S_1,S_2,n_1)

31.fan_out_g(S_1,concat_n(S_2,t_1,n_2),n_1) = fan_out_g(S_1,S_2,n_1)

32.ct_globals_in(S_1,new,n_1,n_2) = 0

33.ct_globals_in(S_1,concat_e(S_2,n_3,n_4),n_1,n_2) = if n_4=n_2
∧ is_a_module(n_3) ∧ n_3<>n_1
then 1 + ct_globals_in(S_1,S_2,n_1,n_2)
else ct_globals_in(S_1,S_2,n_1,n_2)

34.ct_globals_in(S_1,concat_n(S_2,t_1,n_3),n_1,n_2) = ct_globals_in(S_1,S_2,n_1,n_2)

35.ct_globals_out(S_1,new,n_1,n_2) = 0

36.ct_globals_out(S_1,concat_e(S_2,n_3,n_4),n_1,n_2) = if n_3=n_2 ∧
∧ is_a_module(n_4) ∧ n_4<>n_1
then 1 + ct_globals_out(S_1,S_2,n_1,n_2)
else ct_globals_out(S_1,S_2,n_1,n_2)

37.ct_globals_out(S_1,concat_n(S_2,t_1,n_3),n_1,n_2) = ct_globals_out(S_1,S_2,n_1,n_2)

38.#ds(new) = 0

39.#ds(concat_n(S_1,t_1,n_1)) = if t_1=global_data_structure
then 1 + #ds(S_1)
else #ds(S_1)

40.#ds(concat_e(S_1,n_1,n_2)) = #ds(S_1)

41.#module(new) = 0

42.#module(concat_n(S_1,t_1,n_1)) = if t_1=module
then 1 + #module(S_1)
else #module(S_1)

43.#module(concat_e(S_1,n_1,n_2)) = #module(S_1)

44.#links(new)	=	0
45.#links(concat_e(S_1,n_1,n_2))	=	1 + #links(S_1)
46.#links(concat_n(S_1,t_1,n_1))	=	#links(S_1)

Appendix B

Proofs for the axioms of the IF4 metric

The following is an attempted inductive style proof of Axiom 9 for the IF4 model given in Chapter Five.

Concatenating an additional local information flow to the design structure must increase the IF4 measure. To show that the axiom holds for the model we argue inductively that it is true for the following two structures where the outside CONCAT_E represents the information flow being added to the design:

concat_e(concat_n(concat_n(new,module,a),module,b),a,b)
concat_e(concat_n(concat_n(N,module,a),module,b),a,b)

The first structure contains zero flows to which we add a local information flow. The second structure is the case where the structure N already contains n+1 flows where n is non-negative integer. Note that the minimal design structure to which a flow may successfully concatenated is:

S = concat_n(concat_n(new,module,a),module,b)

Dealing with the first case we have:

if4(concat_e(concat_n(concat_n(new,module,a),module,b),a,b)

This gives by Eqn.15:

if4_int(S, concat_e(concat_n(concat_n(new,module,a),module,b),a,b))

By Eqns. 18, 17 and 16 we obtain:

```
IF4m(concat_e(concat_n(concat_n(new,module,a),module,b),a,b) , b)   +
IF4m(concat_e(concat_n(concat_n(new,module,a),module,b),a,b) , a)

= sqr((fan_in_l(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b) +
    fan_in_g(S,concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b) *
   (fan_out_l(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b) +
    fan_out_g(S,concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b)))
  +
  sqr((fan_in_l(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),a) +
     fan_in_g(S,concat_e(concat_n(concat_n(new,module,a),module,b),a,b),a) *
    (fan_out_l(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),a) +
     fan_out_g(S,concat_e(concat_n(concat_n(new,module,a),module,b),a,b),a)))
```

We evaluate each FAN_IN or OUT term individually.

```
fan_in_l(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b)
```

Instantiating into Eqn. 21 $n_1 = n_3$ and n_2 is a module:

```
= 1 + fan_in_l(concat_n(concat_n(new,module,a),module,b),b)
                                (Eqn.21)
= 1 + fan_in_l(concat_n(new,module,a),b)
                                (Eqn.22)
= 1 + fan_in_l(new,b)
                                (Eqn.22)
= 1 + 0
                                (Eqn.20)
```

Next the number of global flows into module b:

```
fan_in_g(S,concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b)
```

This can be seen to be zero because IS_A_DS can never be true as the design contains no global data structures. As a result the ELSE part of Eqn. 27 will selected until the structure is empty and Eqn. 26 applied yielding zero. The local FAN_OUT is:

```
fan_out_l(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b)
```

Instantiating into Eqn. 24 $n_2 \neq n_1$ so:

fan_out_l(concat_n(concat_n(new,module,a),module,b),b) (Eqn.24)

Applying Eqn. 25 twice gives:

fan_out_l(new,b) (Eqn.25)

= 0 (Eqn.23)

The global fan_out for module b:

fan_out_g(S,concat_e(concat_n(concat_n(new,module,a),module,b),a,b),b)

will be zero as has already been stated there are no global data structures.

The information flows for module a are:

fan_in_l(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),a)

Instantiating into Eqn. 21 $n_1 \neq n_3$ so:

= fan_in_l(concat_n(concat_n(new,module,a),module,b),a) (Eqn.21)

Applying Eqn. 22 twice yields:

= fan_in_l(new,a) (Eqn.20)
= 0

The local fan_out for module a is:

fan_out_l(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),a)

Instantiating into Eqn. 24 $n_2=n_1$ and n_3 is a module, so:

= 1 + fan_out_l(concat_e(concat_n(concat_n(new,module,a),module,b),a,b),a) (Eqn.24)

= 1 + fan_out_l(concat_n(concat_n(new,module,a),module,b),a) (Eqn.25)

= 1 + fan_out_l(concat_n(new,module,a),a) (Eqn.25)

= 1 + fan_out_l(new,a) (Eqn.25)

= 1 + 0 (Eqn.23)

The global flows are for module a must be zero, again due to the absence of any global data structures. By substituting back into the equation we now have:

IF4m(S,b) = sqr((1+0)*(0+0))
IF4m(S,a) = sqr((0+0)*(1+0))

and

IF4(S) = 0 + 0

This is an unfortunate result because it indicates that Axiom 9 does not hold over our model since adding the edge to the graph to represent the local information flow has not increased the IF4 measure; it remains at zero.

Appendix C

An Algebraic Specification of the "work" metric

The following is a complete specification of the formal model of the "work" metric described in chapter six of this thesis.

SYNTAX

external operations

newspec:	→ spec
addr: req × req × spec	→ spec
newdes:	→ des
addm: mod × mod × des	→ des
sat: mod × req × des	→ des
comprises: req × spec	→ reqset
exists?: req × spec	→ boolean
prim?: req × spec	→ boolean ∪ {ERROR}
calls: mod × des	→ modset
descend?: mod × mod × des	→ boolean
Pwork: mod × des × spec	reqset
inherits: mod × des	→ reqset
abs: reqset × spec	→ reqset
$work_i$: mod × spec × des	→ real
work: spec × des	→ real
satspec: des × spec	→ boolean
satreq: des × req	→ boolean

internal operations

desc?: mod × mod × des × des	→ boolean
inherit: mod × des × des	→ reqset
abstract: reqset × spec × spec	→ reqset
wk: spec × des × des	→ real
satspec': des × spec × spec	→ boolean

SEMANTICS

vars

r1,r2,r3,r4:req
m1,m2,m3: mod
S,S1: spec
D,D1:des
R:reqset

1. comprises(r1,newspec) = {}
2. comprises(r1,addr(r2,r3,S)) = IF r1==r3 THEN
   comprises(r1,S) ∪ {r2}
   ELSE
   comprises(r1,S)

3. exists?(r1,newspec) = FALSE
4. exists?(r1,addr(r2,r3,S)) = IF r1==r2 THEN TRUE
   ELSE exists?(r1,S)

5. prim?(r1,newspec) = {ERROR}
6. prim?(r1,addr(r2,r3,S)) = IF ¬ exists?(r1,S) THEN {ERROR}
   ELSE
   IF comprises(r1,S)=={} THEN TRUE
   ELSE FALSE

7. calls(m1,newdes) = {}
8. calls(m1,addm(m2,m3,D)) = IF m1==m3 THEN calls(m1,D)∪{m2}
   ELSE calls(m1,D)
9. calls(m1,sat(m2,r1,D)) = calls(m1,D)

10. descend?(m1,m2,D) = desc?(m1,m2,D,D)

11. desc?(m1,m2,newdes,D) = FALSE
12. desc?(m1,m2,addm(m3,m4,D1),D) = IF ((m1==m3)∧((m2==m4)) ∨
    (desc?(m2,m4,D1,D))∧(m4<>''))
    THEN TRUE
    ELSE desc?(m1,m2,D1,D)
13. desc?(m1,m2,sat(m3,r1,D1),D) = desc?(m1,m2,D1,D)

14. Pwork(m1,newdes,S) = {}
15. Pwork(m1,D,newspec) = {}
16. Pwork(m1,sat(m2,r1,D),S) = IF (m1==m2) ∧ comprises(r1,S)={}
THEN Pwork(m1,D,S) ∪ {r1}
ELSE Pwork(m1,D,S)

17. Pwork(m1,addm(m2,m3,D),S) = Pwork(m1,D,S)

18. inherits(m1,D) = inherit(m1,D,D)

19. inherit(m1,newdes,D) = {}
20. inherit(m1,addm(m2,m3,D1),D) = inherit(m1,D1,D)
21. inherit(m1,sat(m2,r1,D1),D) = IF desc?(m2,m1,D)
THEN inherit(m1,D1,D) ∪ {r1}
ELSE inherit(m1,D1,D)

22. abs(R,S) = abstract(R,S,S)

23. abstract({},S1,S) = {}
24. abstract(R,newspec,S) = R
25. abstract(R,addr(r1,r2,S1),S) = IF comprises(r1,S)<>{}
∧ (comprises(r1,S) ⊆ R)
THEN abstract(R,S1,S)∪R∪{r1}
- comprises(r1,S)
ELSE abstract(R,S1,S)

26. $work_i$(m1,newspec,D) = 0
27. $work_i$(m1,S,newdes) = 0
28. $work_i$(m1,S,D) = #(Pwork(m1,D,S)) + α(#(abs(inherits(m1,D)
∪ Pwork(m1,D,S),S)) + #(inherits(m1,D) ∪ Pwork(m1,D,S)))

29. work(S,D) = wk(S,D,D)

30. wk(S,newdes,D) = 0
31. wk(S,addm(m1,m2,D1),D) = wk(S,D1,D) + $work_i$(m1,S,D)
32. wk(S,sat(m1,r1,D1),D) = wk(S,D1,D)

33. satreq(newdes,r1) = FALSE
34. satreq(addm(m1,m2,D),r1) = satreq(D,r1)
35. satreq(sat(m1,r2,D),r1) = IF r1==r2 THEN TRUE
    ELSE satreq(D,r1)

36. satspec(D,S) = satspec'(D,S,S)
37. satspec'(D,newspec,S) = TRUE
38. satspec'(newdes,addr(r1,r2,S1),S) = FALSE
39. satspec'(D,addr(r1,r2,S1),S) = IF prim?(r1,S) THEN
      IF satreq(r1,D) THEN
        satspec'(D,S1,S)
      ELSE
        FALSE
    ELSE satspec'(D,S1,S)

Appendix D

Proofs for the "work" metric

This appendix presents the full proofs for the axioms describing the desired behaviours of the model underlying the "work" metric described in chapter six of this thesis.

Axiom 2: The measure must generate at least two equivalence classes. This obligation is satisified by the a null or empty design that has a measurement value of work equal to zero, whilst a design that satisifies a single functional requirement will have a value of work approximately equal to 1.67.

More formally, the two designs are given by:

newdes

and

sat(m,r,addm(m,",newdes))

The related specifications are respectively:

newspec

and

addr(r,",newspec)

The first design has a work metric value of zero.

work(S,newdes)

= wk(S,newdes,newdes) --Eqn. 29

= 0 --Eqn. 30

The second design has work metric value of about 1.67.

work(S,sat(m,r,addm(m,",newdes)))

= wk(S,sat(m,r,addm(m,",newdes)),D) --Eqn. 29

where D is sat(m,r,addm(m,",newdes))

= wk(S,addm(m,",newdes)),D) --Eqn. 32

= wk(S,newdes,D) + $work_i$(m,S,D) --Eqn. 31

= 0 + $work_i$(m,S,D) --Eqn. 30

= 0 + #(Pwork(m,D,S) + α(#(abs(inherits(m1,D) ∪ Pwork(m1,D,S),S)) +
#(inherits(m1,D) ∪ Pwork(m1,D,S))) --Eqn. 28

Solving for the cardinality of the set of primitive requirements satisfied by module m, we obtain:

#(Pwork(m,sat(m,r,addm(m,",newdes),S))

= #(Pwork(m,addm(m,",newdes),S) ∪ {r}) --Eqn. 16

= #(Pwork(m,newdes),S ∪ {r}) --Eqn. 17

= #({} ∪ {r}) --Eqn. 14

= 1

Instantiating back into our expression for the work metric for the second design we have:

0+1+α(#(abs(inherits(m,D) ∪ {r}),S) + #(abs(inherits(m,D) ∪ {r})))

Next we solve:

inherits(m,D)

= inherit(m,D,D)) ∪ {r},S) —Eqn. 18

= inherit(m,sat(m,r,addm(m,'',newdes)),D)

Next we must evaluate whether module m is a descendant of m, from Equation 21.

descend?(m,m,D)

= desc?(m,m,sat(m,r,addm(m,'',newdes)),D) —Eqn. 10

But since m4='':

= desc?(m,m,newdes,D) —Eqn. 12

= FALSE —Eqn. 11

Since, m is not descendant of itself we have:

= inherit(m,addm(m,'',newdes) —Eqn. 21

= inherit(m,newdes,D) —Eqn. 20

= {} —Eqn. 19

Returning to $work_i$:

= 1+α(#(abs({} ∪ {r},S)) + #({} ∪ {r}))

= 1+α(#(abstract({},S,S)) + #({r})) —Eqn. 22

= 1+α(#({}) + #({r})) —Eqn. 23

= 1+α(1+1)

Assuming a value of 0.33 for the coefficient α, for reasons discussed earlier we obtain a value of close to 1.67 for the work metric. Even if the precise value of the coefficient is

questioned, the model constrains it to a positive real number, consequently the metric value for the second design must be greater than one and the axiom shown to hold.

Axiom 4: There must exist two or more structures that will be assigned to the same equivalence class. To satisfy this all that is required is to find two different design structures and specifications that they fulfil, which yield the same measurement.

More formally, the specification is:

addr(r,",newspec)

And the two designs are defined as:

sat(m_1,r,addm(m_1,",newdes))

and:

addm(m_2,m_1,sat(m_1,r,addm(m_1,",newdes)))

Thus it is only by the operation addm(m_2,m_1,...) that the two designs differ. By reference to Equation 31, it is apparent that this operation has no effect upon the result of the work metric since:

wk(S,addm(m_2,m_1,sat(m_1,r,addm(m_1,",newdes))),D)
= wk(S,sat(m_1,r,addm(m_1,",newdes)),D) + $work_i$(m_2,S,D)

It now only remains to show that $work_i$(m_2,S,D) yields zero. Substituting for S and D gives:

$work_i$(m_2,addr(r,",newspec),sat(m_1,r,addm(m_1,",newdes)))

= #(Pwork(m_2,sat(m_1,r,addm(m_1,",newdes)),
addr(r,",newspec))) + α(#(abs(inherits(m_2,D) $\cup$
Pwork(m_2,D,S),S)) + #(inherits(m_2,D) $\cup$ Pwork(m_2,D,S))) —Eqn. 28

Solving Pwork:

Pwork(m_2,sat(m_1,r,addm(m_1,",newdes)),addr(r,",newspec))

We note that since $m_1 <> m_2$:

= Pwork(m_2,addm(m_1,",newdes),addr(r,",newspec)) --Eqn. 16
= Pwork(m_2,newdes,addr(r,",newspec)) --Eqn. 17

= {} --Eqn. 14

Next we turn to the inherits expression:

inherits(m_2,D)

= inherits(m_2,sat(m_1,r,addm(m_1,",newdes)))

= inherit(m_2,sat(m_1,r,addm(m_1,",newdes)),D) --Eqn. 18

Because m_1 is not a descendant of itself, Equation 21 leads us to:

inherit(m_2,addm(m_1,",newdes),D) --Eqn. 21

= inherit(m_2,newdes,D) --Eqn. 20

= {} --Eqn. 19

This leaves us with the following for $work_i$:

0 + α(#(abs({} ∪ {},S))+#({} ∪ {})))

= 0 + α(#(abs({},S)+#({})))

= 0 + α(#(abstract({},S,S)+0)) --Eqn. 22

= 0 + α(#({})+0)) --Eqn. 23

= 0 + α(0+0)

= 0

Since $work_i(m_2,S,D)$ has been shown to yield zero the addm operation has been demonstrated to have no effect upon the work metric. Consequently the axiom has been shown to be true, as examples of different designs have been given that are members of

the same equivalence class - in this case the class with a value of 1.67 - for the work metric.

Axiom 7: Adding an additional requirement to the system specification must increase the work metric.

$$\forall\ S{:}spec;\ D{:}des\ r{:}req;\ m{:}mod \cdot$$
$$SATSPEC(D,S) \Rightarrow work(S,D) < work(addr(S),sat(m,r,D))$$

Essentially there are two parts to this proof. First, it must shown that if specification *S* is implemented by design *D*, then adding a new requirement to *S* implies adding a requirement satisfaction operation *sat* to *D*. The design may also be augmented by an additional module, but this is incidental to our argument as will become apparent[1]. Second, it must be shown that if the number of *sat* operations for design *D* is increased then the work metric must also increase for *D*.

Since the required proof is universal in nature, induction will be employed. The base case is a null specification *S* and design *D*, whilst the n+1 case is a requirement concatenated to *S* and a module to *D*.

We commence by showing that for the base case the design fails to implement the specification *S* plus a new requirement unless it is augmented by the *sat* operation. The base case specification and design are given by:

newspec

newdes

and the updated specification by:

addr(r,",newspec)

It is clear that the design does not implement the modified specification, that is *satspec* returns false.

satspec(newdes,addr(r,",newspec))

= satspec'(newdes,addr(r,",newspec),addr(r,",newspec)) --Eqn. 36
= FALSE --Eqn. 38

[1]It is the *sat* and not the *addm* operation that increases the value of $work_i$.

Next, we consider the case where the design is augmented by a requirement satisfaction. Where S' is addr(r,",newspec).

satspec(sat(m,r,addm(m,",newdes)),addr(r,",newspec))

= satspec'(sat(m,r,addm(m,",newdes)),addr(r,",newspec),S')

To resolve Equation 39 one must establish that requirement *r* is primitive, hence:

prim?(r,addr(r,",newspec))

= IF comprises(r,addr(r,",newspec)) = {} --Eqn. 6

And since *r* <> '':

= IF comprises(r,newspec) = {} --Eqn. 2

= IF {} = {} --Eqn. 1

= TRUE

To complete the evaluation of Equation 39 it must be determined whether the design implements requirement *r*:

satreq(r,sat(m,r,addm(m,",newdes)))

= IF r=r --Eqn. 35

= TRUE

Returning to Equation 39 we now obtain:

satspec'(sat(m,r,addm(m,",newdes)),newspec,S') --Eqn. 39

= TRUE --Eqn. 37

Thus, we have shown that the an additional requirement satisfaction is a necessary and sufficient condition, in order for the design *D* to satisfy the specification *S* augmented by another requirement. As it is our aim to establish that the above is true for all designs and specifications we turn now to the $n^{th}+1$ case. This gives the following specification and design:

addr(r1,r2,S)

where *r1*<> *r2* but *r2* may be equal to *r*, *S* is newspec and:

sat(m,r1,D)

or

sat(m1,r1,addm(m1,m2,D))

where *m1* <> *m*, but *m2* may be equal to *m* and *D* is addm(m,",newdes). Note that there are two possible cases for the design, since the satisifaction of a new requirement may be implemented by an existing module - the first case - or by the introduction of a new module - the second case.

Now we show that an additional requirement added to the specification creating *S'*:

addr(r3,r2,addr(r1,r2,S))

causes:

satspec(sat(m,r1,D),addr(r3,r2,addr(r1,r2,S))) = FALSE

and

satspec(sat(m,r1,addm(m1,m2,D)),addr(r3,r2,addr(r1,r2,S))) = FALSE

In practise the first proposition is based upon a meaningless structure, because a design without modules cannot implement any specification, so we only need evaluate:

satspec(sat(m,r1,addm(m1,m2,newdes)),addr(r3,r2,addr(r1,r2,newspec))) = FALSE

= satspec'(sat(m,r1,addm(m1,m2,newdes)),addr(r3,r2,addr(r1,r2,newspec)),S')
–Eqn. 36

Since r1 is primitive the following expression must be evaluated for Equation 39:

satreq(r3,sat(m,r1,addm(m,",newdes))) –Eqn. 39

Because r3<>r1:

= satreq(r3,addm(m,'',newdes)) --Eqn. 35

= satreq(r3,newdes) --Eqn. 34

= FALSE --Eqn. 33

Returning to Equation 39 we have:

satspec'=FALSE

Therefore the n^{th}+1 design does not satsify the n^{th}+1 specification when an additional requirement is introduced. We can now state that for all designs and specifications, the intoduction of a further requirement ot the specification will cause the design to fail to satisfy it. We now proceed to show that for all cases described above, if the design presently meets its specification then it must be augmented by a *sat* operation in order to continue to meet the specification.

Such a design, *D'* is defined:

sat(m,r3,D)

= sat(m,r3,sat(m,r1,addm(m,'',newdes)))

The proposition to be found true is:

satspec'(sat(m,r1,sat(m,r,addm(m,'',newdes))),
addr(r3,r2,addr(r1,r2,newspec)),S') --Eqn. 36

As has already been shown:

prim?(r3,S') = TRUE

So we must next determine:

satreq(r3,sat(m,r3,sat(m,r1,addm(m,'',newdes)))) --Eqn. 39

As r3=r3 this expression yields true from Equation 35. Returning to satspec' we obtain:

satspec'(sat(m,r1,sat(m,r,addm(m,'',newdes))),addr(r1,r2,newspec),S')

--Eqn. 39

We know that requirement r1 is primitive so we must solve:

satreq(r1,sat(m,r3,sat(m,r1,addm(m,",newdes))))

= satreq(r1,sat(m,r1,addm(m,",newdes))) --Eqn. 35

As r1=r1, satreq yields TRUE, consequently we now have:

satspec'(sat(m,r1,sat(m,r,addm(m,",newdes))),newspec,S') --Eqn. 39

= TRUE --Eqn. 37

This means that for all designs, if their specification is extended by an extra requirement, then the addition of a *sat* operation will mean that the design continues to meet its specification.

Armed with this information, it must now demonstrated that the implementation of an additional requirement in a design always increases the value of the work metric. Again, the approach will be to employ an inductive proof, since the universal case is being considered. The base design is:

newdes

and the $n^{th}+1$ design is:

sat(m,r,addm(m,",newdes))

The associated specifications are:

newspec

and:

addr(r,",newspec)

In both cases it will be shown that by implementing an additional requirement, represented by the *sat* operation will increase the value of the work metric for the design. Starting with the base case, it has already been shown that an empty design has

a work metric value of zero in the proof of Axiom 2. By implementing an additional requirement we get:

sat(m,r,addm(m,",newdes))

which is, incidentally, the identical to the n^{th}+1 design. Again, it has been previously demonstrated that this design has a value of 1.67. Thus, the addition of a *sat* operation has increased the value of the work metric.

The final step is to show that the inclusion of an extra *sat* operation has the same effect for the n^{th}+1 design. This is already known to have a value of 1.67 so it merely remains to evaluate this design, augmented by a further requirement implementation.

work(addr(r1,",addr(r,",newspec)),
sat(m1,r1,addm(m1,",sat(m,r,addm(m,",newdes)))))

= wk(addr(r1,",addr(r,",newspec)),
sat(m1,r1,addm(m1,",sat(m,r,addm(m,",newdes)))),D1)
--Eqn. 29

where D1:

= sat(m1,r1,addm(m1,",sat(m,r,addm(m,",newdes))))

= wk(addr(r1,",addr(r,",newspec)),
addm(m1,",sat(m,r,addm(m,",newdes))),D1) --Eqn. 32

= wk(addr(r1,",addr(r,",)),sat(m,r,addm(m,",newdes))),D1)
+ $work_i$(m1,addr(r1,",addr(r,",newspec)),D) --Eqn. 31

= wk(addr(r1,",addr(r,",)),addm(m,",newdes),D1)
+ $work_i$(m1,addr(r1,",addr(r,",newspec)),D) --Eqn. 32

= wk(addr(r1,",addr(r,",newspec)),newdes,D1)
+ $work_i$(m1,addr(r1,",addr(r,",newspec)),D1)
+ $work_i$(m,addr(r1,",addr(r,",newspec)),D1) --Eqn. 31

= 0 + $work_i$(m1,addr(r1,",addr(r,",newspec)),D1)
+ $work_i$(m,addr(r1,",addr(r,",newspec)),D1) --Eqn. 30

= #(Pwork(m1,D1,addr(r1,",addr(r,",newspec)))) + α(#(abs(inherits(m1,D1)
∪ Pwork(m1,D1,addr(r1,",addr(r,",newspec)))))
+ #(inherits(m1,D1) ∪
Pwork(m1,D1,addr(r1,",addr(r,",newspec))))))) +
#(Pwork(m2,D1,addr(r1,",addr(r,",newspec)))) +
α(#(abs(inherits(m2,D1)
∪ Pwork(m2,D1,addr(r1,",addr(r,",newspec)))))
+ #(inherits(m2,D1) ∪
Pwork(m2,D1,addr(r1,",addr(r,",newspec)))))))

--Eqn. 28

In order to shorten the solution of the above expression, the reader should note the following. First, that neither module *m* nor *m1* have any descendants and therefore both inherit an empty set of requirements with a cardinality of zero. Second, both modules satisfy exactly one requirement, thus the cardinality of the result of Pwork is one in each case. Third, the abstraction operation cannot reduce a set with only member, consequently it may be disregarded. This allows us to subsitute into the above expression to obtain:

$1 + \alpha(1+1) + 1 + \alpha(1+1)$

$= 3.33$

Clearly 3.33 is greater than 1.67 and therefore our axiom holds for the $n^{th}+1$ case, and therefore by inductive reasoning, for all cases.

Axiom 9: There must exist different designs that satisfy the same specification but which have different work metric values. In other words design size is not only a function of the specification but also the intrinsic organisation of the design. The consequence of this axiom is that for a given specification choice of architecture can influence size.

$\exists D_1,D_2$:des; S:spec · satspec(D_1,S) ∧ satspec(D_2,S) ⇒
work(S,D_1) <> work(S,D_2)

The proof of the validity of this axiom for the model is an existential one; we merely have to find two designs for which the axiom holds in order to establish its validity. In order to further reduce the length of the proof, the last example employed to establish axiom 7 will be re-utilised along with a different design to implement the same specification (refer to Figure 6.2). The specification common to both designs, S is:

addr(r1,",addr(r,",newspec))

The first design, *D1*, with a known work metric value of 3.33 is:

sat(m1,r1,addm(m1,",sat(m,r,addm(m,",newdes))))

whilst the second design, *D2*, is formally described as:

sat(m1,r1,addm(m1,m,sat(m,r,addm(m,",newdes))))

The difference between the designs lies in the fact that in the second design *D2*, module *m1* is a subordinate of module *m* and as a consequence *m* inherits the requirement satisfaction of module *m* and therefore has to do scheduling work, the result of which is a higher work metric value. This is shown more formally below where we solve:

work(addr(r1,",addr(r,",newspec)),
sat(m1,r1,addm(m1,m,sat(m,r,addm(m,",newdes)))))

= wk(addr(r1,",addr(r,",newspec)),
sat(m1,r1,addm(m1,m,sat(m,r,addm(m,",newdes)))),D2)
–Eqn. 29

$= 0 + work_i(m1,S,D2) + work_i(m,S,D2)$ –Eqns.30,31,32

Solving for module *m1* yields:

$work_i(m1,S,D2)$

$= \#(Pwork(m1,D2,S) + \alpha(\#(abs(inherits(m1,D2) \cup Pwork(m1,D2),S))$
$+ \#(inherits(m1,D2) \cup Pwork(m1,D2,S)))$ –Eqn. 28

Commencing with the Pwork expression:

Pwork(m1,D2,S)

= Pwork(m1,sat(m1,r1,addm(m1,m,sat(m,r,addm(m,",newdes)))),S)

We note that m1=m1 so Equation 16 requires us to evaluate:

comprises(r1,S)

= comprises(r1,addr(r1,",addr(r,",newspec)))

From Equation 2 the predicate r1=" is false thus we obtain:

comprises(r1,addr(r,",newspec)) --Eqn. 2

= comprises(r1,newspec) --Eqn. 2

= {} --Eqn. 1

Instantiating back into Equation 16 yields the following predicate which is clearly true:

m1=m1 AND {}

Consequently our Pwork expression may be simplified to:

Pwork(m1,addm(m1,m,sat(m,r,addm(m,",newdes))),S) ∪ {r1} --Eqn.16

= Pwork(m1,sat(m,r,addm(m,",newdes)),S) ∪ {r1} --Eqn.17

m1<>m, therefore:

= Pwork(m1,addm(m,",newdes),S) ∪ {r1} --Eqn.16

= Pwork(m1,newdes),S) ∪ {r1} --Eqn.17

= {} ∪ {r1} --Eqn.14

= {r1}

Returning to the original $work_i$ expression for module m1 and substituting for Pwork(m1,D2,S) gives:

#({r1}) + α(#(abs(inherits(m1,D2) ∪ {r1},S)
+ #(inherits(m1,D2) ∪ {r1})))

Next we turn to inherits(m1,D2). From Figure 6.2 we observe that module m1 is a leaf module and therefore cannot inherit requirement satisfaction, and thus the inherits expression evaluates to the empty set.

= #({r1}) + α(#(abs({} ∪ {r1},S) + #({} ∪ {r1})))

= #({r1}) + α(#(abs({r1},S) + #({r1})))

The abs({r1},S) expression yields {r1} because a single requirement cannot be made more abstract, therefore we have:

$$\#(\{r1\}) + \alpha(\#(\{r1\}) + \#(\{r1\}))$$

$$= 1 + \alpha(1 + 1)$$

$$= 1.67$$

Now, we turn to the second module in the architecture *D2* to solve $work_i$ for m.

$$work_i(m,S,D2)$$

$$= \#(Pwork(m,D2,S) + \alpha(\#(abs(inherits(m,D2) \cup Pwork(m,D2),S)) + \#(inherits(m,D2) \cup Pwork(m,D2,S))) \quad \text{--Eqn. 28}$$

Starting with Pwork(m,D2,S):

$$= Pwork(m,sat(m1,r1,addm(m1,m,sat(m,r,addm(m,'',newdes)))),S)$$

$$= Pwork(m,addm(m1,m,sat(m,r,addm(m,'',newdes))),S) \quad \text{--Eqn.16}$$

$$= Pwork(m,sat(m,r,addm(m,'',newdes)),S) \quad \text{--Eqn.17}$$

Since m=m and comprises(r,S)={}:

$$= Pwork(m,addm(m,'',newdes),S) \cup \{r\} \quad \text{--Eqn.16}$$

$$= Pwork(m,newdes,S) \cup \{r\} \quad \text{--Eqn.17}$$

$$= \{\} \cup \{r\} \quad \text{--Eqn.14}$$

$$= \{r\}$$

Returning to the original $work_i$ expression for module m and substituting for Pwork(m1,D2,S) gives:

$$\#(\{r\}) + \alpha(\#(abs(inherits(m,D2) \cup \{r\},S) + \#(inherits(m,D2) \cup \{r\})))$$

Next we turn to inherits(m,D2). Reference to Figure 6.1 reveals that unlike module m1 it is not a leaf module.

= inherit(m,D2,D2) --Eqn.18

= inherit(m,sat(m1,r1,addm(m1,m,sat(m,r,addm(m,",newdes")))),D2)

Since desc(m1,m) is true:

= inherit(m,addm(m1,m,sat(m,r,addm(m,",newdes"))),D2) ∪ {r1} --Eqn.21

= inherit(m,sat(m,r,addm(m,",newdes")),D2) ∪ {r1} --Eqn.20

Note that *desc(m,m)* is false because a module cannot be a descendent of itself, therefore:

= inherit(m,addm(m,",newdes),D2) ∪ {r1} --Eqn.21

= inherit(m,newdes,D2) ∪ {r1} --Eqn.20

= {} ∪ {r1} --Eqn.19

= {}

Returning to $work_i$ for module m:

#({r}) + α(#(abs({r1} ∪ {r},S) + #({r1} ∪ {r})))

= #({r}) + α(#(abs({r1,r},S) + #({r1,r})))

Now we address abs({r1,r},S):

= abstract({r1,r},S,S) --Eqn.22

= abstract({r1,r},addr(r1,",addr(r,",newspec)),S)

Requirement *r1* is primitive thus:

comprises(r1,S) = {} so:

= abstract({r1,r},addr(r,",newspec),S) --Eqn.25

Again requirement *r* is primitive, therefore:

= abstract({r1,r},newspec),S) –Eqn.25

= {r1,r} –Eqn.24

Finally, and hopefully not in the Pauline sense, for $work_i$ we have:

#({r}) + α(#({r1,r} + #({r1,r})))

= 1 + α(2+2)

= 2.33

When this added to the $work_i$ value for module m1 of 1.67 an overall value for design *D2* of 4.0 is obtained. Clearly this differs from the value of 3.33 for design *D1*, despite the fact that both designs implement the same specification *S*. Consequently, Axiom 9 stands.