A Critique of Three Metrics

M. Shepperd
Department of Computing and Cognition, Bournemouth University, Bournemouth, United Kingdom

D. C. Ince
Department of Computer Science, Open University, Milton Keynes, United Kingdom

This article examines the metrics of the software science model, cyclomatic complexity, and an information flow metric of Henry and Kafura. These were 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 summarized. Each metric is then subjected to an in-depth critique. All are found wanting. We maintain that this is not due to mischance, but indicates deeper problems of methodology used in the field of software metrics. We conclude by summarizing these problems.

1. INTRODUCTION
This article examines three metrics. The first metric—or, rather, family of metrics—are those derived from Halstead's (1977) software science. This is almost certainly the widest studied and "validated" metric. Of the three metrics examined in this article this has, over the last 10 years, been the recipient of most criticism; we will summarize these criticisms.

The second metric, cyclomatic complexity, is attributable to McCabe (1976). The application of this metric has been far ranging, and today, the creation, investigation, and promulgation of graph-theoretic software measures is a major industry. Yet, surprisingly, the usefulness of cyclomatic complexity as a software metric has been allowed to pass relatively unquestioned. Indeed, it is still widely cited in textbooks (Arthur, 1985; Pressman, 1992; Sommerville, 1989; Wiener and Sincovec, 1984), subjected to many minor modifications (Hansen, 1978; Iyengar et al., 1982; Myers, 1977; Sinha et al., 1986; Stetter, 1984), and has even been applied as a design metric (Hall and Preiser, 1984). However, empirical support is small and, as will be seen from this article, its theoretical underpinning extremely suspect.

A more recent measure is the information flow metric of Henry and Kafura (Henry, 1981; Henry et al., 1981). This metric has achieved a great deal of prominence mainly because it was one of the earliest design metrics. The authors claim that it may be used for prediction and to pinpoint weaknesses in a 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—software science, cyclomatic complexity, and information flow—have been selected as representative of a great deal of current work in the software metrics domain and because much additional work treats these metrics and their underlying models as fundamental. They have also been widely applied and have generated more empirical data than other metrics.

This article will reveal a constant pattern of poorly conceived and articulated models that underlie the three metrics. This has led, in turn, to results that are anomalous and out of step with current software engineering practice. We also demonstrate that researchers have ignored the modelling aspects of metrics development. In short, we describe major weaknesses in foundation and methodology.

It is worth stressing that this article is not meant to be a destructive criticism of the metrics and their developers. Each of the metrics represents a land-

---

mark in the history of software metrics: software science was the first attempt to erect a theory of software measurement, cyclomatic complexity alerted the software community to the possibilities of measuring the unstructuredness of program code, and the work of Henry and Kafura pointed out the importance of measuring intermodule factors. What we hope this article achieves is to dissect the work, with the results of the dissection being used to pinpoint weaknesses and indicate fruitful directions for both users and developers of metrics. At minimum, we hope that the work will alert the software engineering community to the problems that they will encounter if they adopt an off-the-shelf approach to metrics selection.

The development of this article has been partly motivated by our experience over the last four years of refereeing metrics papers for major journals and conferences. Many of the articles that we have read have had a somewhat uncritical view of the metrics described here and, more importantly, have made the implicit assumption that the use of metrics is no more than the process of using existing metrics without, for example, asking hard questions about purpose.

2. SOFTWARE SCIENCE

Halstead's (1977) software science was one of the first attempts to provide a theory of software measurement that is oriented toward program code (Funayami and Halstead, 1976; Halstead 1972, 1977; Love and Bowman, 1976; van der Knijf, 1978). The theory is based on the fact that program comprehension involves the mental comprehension of program tokens: variables, keywords etc.

The software science metrics are derived from a number of counts, for example, the count of the number of unique operators in a module or program. From these counts a number of measures are derived: program volume V, minimum program volume V*, program level λ, the inverse of the program level D, effort E, and program length N (Halstead, 1977). Halstead hypothesized that, for a particular programming language, as V* increases, the program level λ will decrease such that V*L remains invariant. By use of this invariant, which he termed l, 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 (Halstead, 1977). A number of studies have attempted to establish l for programming languages, for example, ESS (Bailey and Dingee, 1981), COBOL (Shen and Dunsmore, 1981; Zweben and Fung, 1979), PL/S and BAL (Smith, 1980), RPG (Hartman, 1982), and APL (Zweben and Fung, 1979; Laurmaa and Syrjanen, 1982).

A feature apparent in all the above investigations is that most results accord with intuitive expectations. Early empirical validations of software science produced apparently large correlations between predicted and actual results.

Software science-based papers still form the majority of academic publications in the area, with researchers probing the metrics for a wide variety of predictive properties. For example, researchers have related software science measures to development time (Gordon and Halstead, 1976; Halstead, 1977), the incidence of software bugs (Funayami and Halstead, 1976), program recall (Love, 1977), and program quality (Elshoff, 1976; Fitzsimmons, 1978a; Love and Bowman, 1976). Software science metrics have even been used to detect student plagiarism (Ottenstein, 1976).

Software science attracted considerable interest in the 1970s. We feel that there were a number of reasons for this, including the lack of other metrics, the ease with which software science metrics could be extracted from program code, and a huge increase in the perception of software development as being an engineering discipline, with all the baggage associated with engineering, including measurement.

Despite the initial enthusiastic reception for software science, a number of serious problems have recently emerged. A careful analysis of the results obtained since the early 1970s reveal a number of disturbing problems. The surface evidence of these problems is summarized in Table 1, which is taken from Shepperd and Ince (1993). The final column of this table represents a subjective judgement by the authors. 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. First, researchers have applied the metric to many measurable attributes, many of which were not in Halstead's original model, ranging from ease of maintenance, number of errors, program recall, and development time to docu-

---

1 Careful review of the methods and analytical techniques used, for example, by Hamer and Frewin (1982), suggest that these correlations are less significant than was first believed.

A PASCAL program to extract the software science metrics reproduced in deMarco's (1982) well-known metrics text book is scarcely 200 lines of code in length.
A Critique of Three Metrics

J. SYSTEMS SOFTWARE
1994; 26:197–210

199

Table 1. Empirical Validations of Software Science

<table>
<thead>
<tr>
<th>Study</th>
<th>Correlation with LOC</th>
<th>Better than LOC?</th>
<th>Dependent Variable</th>
<th>Useful Predictor?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basili and Phillips, 1981</td>
<td>n.a.</td>
<td>=</td>
<td>Effort</td>
<td>Weak</td>
</tr>
<tr>
<td>Basili et al., 1983</td>
<td>n.a.</td>
<td>No</td>
<td>Bug location</td>
<td>Weak</td>
</tr>
<tr>
<td>Bowen, 1978</td>
<td>n.a.</td>
<td>No</td>
<td>Effort</td>
<td>Weak</td>
</tr>
<tr>
<td>Curtis et al., 1979a</td>
<td>n.a.</td>
<td>No</td>
<td>Bug location</td>
<td>No</td>
</tr>
<tr>
<td>Curtis et al., 1979b</td>
<td>n.a.</td>
<td>Yes</td>
<td>Program recall</td>
<td>No</td>
</tr>
<tr>
<td>Elshoff, 1976</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Length</td>
<td>Yes</td>
</tr>
<tr>
<td>Evangelist, 1984</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Program style</td>
<td>No</td>
</tr>
<tr>
<td>Funayami and Halstead, 1976</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Errors</td>
<td>No</td>
</tr>
<tr>
<td>Gordon and Halstead, 1976</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Debugging effort</td>
<td>No</td>
</tr>
<tr>
<td>Halstead, 1977</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Effort</td>
<td>Yes</td>
</tr>
<tr>
<td>Henry et al., 1981</td>
<td>Yes</td>
<td>n.a.</td>
<td>Changes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lind and Vairavan, 1989</td>
<td>Yes</td>
<td>No</td>
<td>Effort</td>
<td>Weak</td>
</tr>
<tr>
<td>Ottenstein, 1979</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Debugging effort</td>
<td>No</td>
</tr>
<tr>
<td>Shen, 1979</td>
<td>Yes</td>
<td>n.a.</td>
<td>Effort</td>
<td>No</td>
</tr>
<tr>
<td>Woodfield, 1980</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Effort</td>
<td>Weak</td>
</tr>
</tbody>
</table>

n.a., Not applicable.

mentation effort. This leads to difficulties in comparison and reuse of experimental data. Many researchers seem to have treated Halstead’s work as a general model of software complexity that is capable of predicting almost any aspect of software on the basis of program operand and operator counts (Kearney et al., 1986).

Second, even where there is agreement as to which aspect of software development Halstead’s metrics address, many empirical validations encounter problems of what exactly to include within their experiments. For example, in a study that attempts to correlate software science metrics with development time, does this include time spent analyzing requirements or time spent on design work? Should effort expended on documentation be included? What about the scenario in which the developer lies awake at night pondering some particularly intractable problem? The problem is that the software science model is couched in terms too nebulous to admit more precise definition—a problem which, as will be seen later in this article, it shares with other models.

Third, a problem that also arises from the nebulous description of software science is what to count and what not to count in program code. This alone should cause us to treat empirical results with some degree of caution. This has, for example, led some researchers to treat the system as a single module in calculating the $E$ metric and other researchers to sum the $E$ metric values for each module in a system to give an overall $E$ value for a system.

Fourth, there is considerable disquiet concerning the quality of many empirical validations and their associated statistical analyses. Some examples are as follows:

- Two studies (Comer and Halstead, 1979; Halstead, 1977) claim a high correlation between software science $E$ and programming time. Unfortunately, the correlation is with the square root of $E$ (Hamer and Frewin, 1982).

- A study used misreported data and incorrectly estimated the software science metrics $q_1$ and $q_2$ but, nevertheless, obtained a high correlation with the actual debugging data (Funayami and Halstead, 1976).

- Another study (Ottenstein, 1979) applied the software science model and obtained results that did not agree with the actual data; as a consequence, the author multiplied her estimates by a project-dependent constant in a manner entirely inconsistent with the software science model.

Card and Agresti (1987) argued that many of the impressive empirical results supporting software science are illusory and can be explained by the fact that the software science metrics $N$ and $\dot{N}$ are dependent by definition.

---

4 An excellent discussion of the quality of statistical validations of the Halstead metrics can be found in Hamer and Frewin (1982).

5 Hamer and Frewin (1982) presented a table comparing the differing results. This seems to have excited no comment other than that both studies lend support to Halstead’s work. Clearly, such support is entirely illusory.
Fifth, there are a number of theoretical objections to the metric. A number of these are summarized below:

- Many of the software science counting rules seem arbitrary. For example, the treatment of `goto (label)` as a unique operator for each unique label is inconsistent with the view that `if (condition)` is a single operator irrespective of the number of unique conditions (Lassez et al., 1981).\(^6\) This is a major problem given that software science metrics are sensitive to rule changes (Becei, 1982; Shen et al., 1983).

- A major assumption made by Halstead was that humans use a binary search mechanism. There is no empirical evidence for this assertion within the domain of programming (Coulter, 1983).

- Halstead used $S$, the Stroud number (Stroud, 1966) in one of his equations and assigned it the arbitrary value of 18, even though Stroud had postulated a range of 5 to 20 (Coulter, 1983) and had also used the concept to sensory memory. There has been little validation of the use of the Stroud number and its assignment of the value 18 apart from two studies (Halstead, 1977; Ottenstein 1979), which have major concerns about statistical validity attached to them.

- The view of the software science model as software as a sequence of tokens is far too simplistic and ignores control, data, file, and module structures. For example, in many programming languages, Halstead operators and operands cannot be regarded as mutually exclusive (Lassez et al., 1981). Two relatively recent works provide evidence that this causes problems (Johnston and Lister, 1981; Lister, 1982).

- Software science was developed at a time when software development consisted of the development in batch mode of commercial and industrial systems that were relatively small—often no more than a few hundreds of lines of executable code written in third-generation languages such as FORTRAN and COBOL. Many of the validations involved student-developed software of $< 100$ statements, for example, Gordon and Halstead (1976). To attempt to scale up the results of even the small number of statistically valid experiments carried out in the 1960s and early 1970s to the development of the large systems produced today would seem to be highly problematical.

To summarize: many of the assumptions made by Halstead have been found wanting, particularly the psychological ones; there are few valid empirical validations of the software science metrics; and in many studies, close perusal of the results show that software science metrics have performed no better than lines of code. Despite these difficulties, there remains widespread and uncritical reference to it, even in the literature and text books (Arthur, 1985; Prather, 1988; Schneider, 1988).

Possibly the most important legacy of software science is 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—something lacking in the later attempts at metrification that we describe here.

3. CYCLOMATIC COMPLEXITY

In the mid 1960s, a software metric based on graph theory was devised by McCabe. It was known as cyclomatic complexity (McCabe, 1976). McCabe was particularly interested in the number of control flow paths through a piece of software, because 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 in which each line in the graph represents a flow of control. From such a graph a metric, known as cyclomatic complexity, can be extracted that represents the number of basic paths within the graph. McCabe saw two practical applications of the metric: as an upper limit beyond which a module should be further split into smaller modules and as a measure of the testability of a module.

Initially, just like software science, cyclomatic complexity was well received by the software engineering community and made the subject of a considerable number of validations. Also, like software science, it has been widely interpreted as a general measure of software complexity that is able to predict a wide variety of software factors.

There are a number of theoretical concerns with the metric:

- McCabe was originally concerned with the metrification of FORTRAN programs, in which the mapping from source code to directed graph was clear. Such a mapping is not clear for other languages.

\(^6\)The dangers of this are illustrated by the decision of Balut et al. (1974) to treat each `goto` as a unique operator with only the justification that it improved the correspondence between $N$ and $N$. 
A Critique of Three Metrics

such as fourth-generation languages and concurrent languages such as Ada, in which, for example, it is not clear how its exception handling construct can be handled within McCabe's framework.

- Cyclomatic complexity values of 1 will be generated by any length of linear code. Thus, the metric is insensitive to complexity contributed by, for example, a large number of interdependent assignment statements. Thus, function-bound software represents a major class of system for which the metric is a poor predictor.

- Cyclomatic complexity is insensitive to the structuring of software. A number of studies have pointed out that application of structure-improving heuristics can lead to an increase of cyclomatic complexity, and many rules of good programming do not lead to a decrease of the metric (Baker and Zweben, 1980; Oulsnam, 1979; Prather, 1984; Sinha et al., 1986; Evangelist, 1983, 1984). The probable reason for this startling anomaly is that McCabe, like Halstead, takes a lexical view of program code, rather than a structural view.

- It has been shown that cyclomatic complexity increases when a designer uses the technique of factoring out duplicate code in order to increase modularization (Shepperd, 1988a). This is at variance with current software engineering ideas on design.

- The metric ignores factors that are becoming increasingly important, such as data and functional complexity.

These are theoretical objections. The empirical evidence is no more encouraging. As Table 2 (taken from Shepperd and Ince [1993]) indicates, the results of various empirical validation studies do not lend much credence to the metric. A more detailed account is given in Shepperd (1988a). The clearest result is the strong relationship between cyclomatic complexity and lines of code (LOC); ironically, it was the "inadequacy" of LOC as a module complexity metric that led to McCabe to propose cyclomatic complexity as an alternative to the more traditional LOC.

There are a number of reasons for the erratic results presented in Table 2. There are the theoretical objections that we have already voiced. Also, there is the tendency of validators to try to correlate cyclomatic complexity with factors with which it was not intended to correlate. For example, 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 overall software development effort. Thus, counting cyclomatic complexity across entire programs, rather than individual modules as some researchers have done (Woodfield et al., 1981), is not entirely appropriate.

In summary, the limited work that has already been carried out is not very encouraging. The only convincing role for cyclomatic complexity seems to be as an intramodular complexity metric. However, even this appears suspect as judged by the empirical work carried out by Basili and Perricone (1984). In any case, many researchers, for example, Stevens et al. (1974), would argue that the problem of how to

Table 2. Empirical Validations of Cyclomatic Complexity

<table>
<thead>
<tr>
<th>Study</th>
<th>Correlation with LOC</th>
<th>Better than LOC?</th>
<th>Dependent Variable</th>
<th>Useful Predictor?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basili et al., 1983</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Bug location</td>
<td>No</td>
</tr>
<tr>
<td>Basili and Perricone, 1984</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Effort</td>
<td>No</td>
</tr>
<tr>
<td>Bowen, 1978</td>
<td>Yes</td>
<td>n.a.</td>
<td>Error density</td>
<td>No</td>
</tr>
<tr>
<td>Curtis et al., 1979</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Bug location</td>
<td>No</td>
</tr>
<tr>
<td>Curtis et al., 1979b</td>
<td>Yes</td>
<td>No</td>
<td>Program recall</td>
<td>No</td>
</tr>
<tr>
<td>Gaffney, 1979</td>
<td>Yes</td>
<td>n.a.</td>
<td>Effort</td>
<td>Weak</td>
</tr>
<tr>
<td>Henry et al., 1981</td>
<td>Yes</td>
<td>n.a.</td>
<td>Changes</td>
<td>Yes</td>
</tr>
<tr>
<td>Kitchenham, 1981</td>
<td>Yes</td>
<td>No</td>
<td>Errors</td>
<td>Weak</td>
</tr>
<tr>
<td>Lind and Vairavan, 1989</td>
<td>Yes</td>
<td>No</td>
<td>Effort</td>
<td>Weak</td>
</tr>
<tr>
<td>Paige, 1980</td>
<td>Yes</td>
<td>No</td>
<td>Testing effort</td>
<td>Weak</td>
</tr>
<tr>
<td>Schneider, 1988</td>
<td>Yes</td>
<td>n.a.</td>
<td>Errors</td>
<td>No</td>
</tr>
<tr>
<td>Shen et al., 1985</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Errors</td>
<td>Yes</td>
</tr>
<tr>
<td>Sunohara et al., 1981</td>
<td>n.a.</td>
<td>n.a.</td>
<td>Effort</td>
<td>Weak</td>
</tr>
<tr>
<td>Wang and Dunsmore, 1984</td>
<td>Yes</td>
<td>No</td>
<td>EFFICIENCY</td>
<td>Yes</td>
</tr>
<tr>
<td>Woodfield et al., 1981</td>
<td>n.a.</td>
<td>n.a.</td>
<td>EFFICIENCY</td>
<td>No</td>
</tr>
<tr>
<td>Woodward et al., 1979</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

n.a., Not applicable.
modularize a program is better resolved by considerations of intermodular complexity, a theme taken up in the next section.

It may well be that some find McCabe's metric "intellectually very appealing" (Woodward et al., 1979), but there are few grounds for its widespread adoption. The empirical evidence shows erratic support for the measure and a rather more consistent relationship with the more easily measured LOC. The likely explanation is that of cross-correlation: decisions in software have a relatively uniform density and hence are correlated with the number of LOC.

The theoretical basis is equally insubstantial. The underlying model does not address any particular facet of program design and construction; also, being an almost purely lexical entity, it does not include any notion of software structure.

4. HENRY AND KAFURA'S INFORMATION FLOW MEASURE

More recently, attention has focused on metrics derived from system design. Probably the most well known metric is that of Henry and Kafura (Henry, 1979, 1981; Henry and Kafura, 1984; Henry and Selif, 1990) known as the information flow metric, although it is fair to say that other metrics such as those described in Agresti and Evanco (1992) are also being developed and validated.

The basic idea behind this metric is that the complexity of a software module—that is, a functional design unit—is related to the number of 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 or cyclomatic complexity. Such ideas are loosely derived from the design evaluation criteria of module coupling and cohesion described by Stevens et al. (1974). The calculation of the metric is as follows.

Connections are defined as channels or information flows whereby one module can influence another. The following types of information flows are defined; in each case, there is a flow from module $A$ to $B$:

- local flows, which may either be direct, when module $A$ passes parameters to $B$ (Figure 1), or indirect, when $A$ returns a value to $B$ (Figure 1); or module $C$ calls $A$ and $B$ and passes the result value from $A$ to $B$
- global flows, in which module $A$ writes to a data structure $DS$, and $B$ reads from $DS$.

A module's connections to its environment are a function of its fan-in and fan-out. The fan-in $fi$ 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 $fo$ is the number of local flows that originate from a procedure plus the number of data structures updated. The value of the metric is

$$\text{length} = (fi + fo)^2$$

where length is measured by LOC and represents the internal contribution to complexity.

A number of applications have been suggested for the information flow metric. It may be used to identify potentially troublesome modules with a high metric value (Kafura and Reddy, 1987). Another application is to measure the trend in metric values between levels in a design: a sharp increase in complexity between levels indicates design problems.

The types of problem that the metric is intended to detect are as follows: lack of cohesion; stress points such as modules, where there is a high level of "through traffic" and where modification of the modules can lead to major modifications to the overall system in which they reside; inadequate refinement, for example, a missing layer of abstraction; and overloaded data structures that need to be decomposed. Initially, Henry and Kafura applied their metric to the UNIX operating system and had some success in identifying problem areas. They found a high correlation ($r = 0.95$) between information flow and the number of program changes, which they took as a proxy for errors (Henry et al., 1981). The information flow metric is unusual in that it is one of the few design metrics that has been subjected to a number of empirical investigations, although fewer than those associated with software science and cyclomatic complexity. These are sum-

---

*An early indication of problems with this metric was that the length component of the metric actually detracted from its performance, and a Spearman correlation of $r = 0.98$ was obtained without the inclusion of length in the metric.*
marized in Table 3, which is taken from Shepperd and Ince (1993).

It is apparent that the results of the studies are varied in nature and have produced mixed results. This is in part due to modelling deficiencies: just as for the other metrics previously discussed, the basic model does not indicate what aspects of software or the software process are being addressed, thus leaving, as a default, a degree of universalism that may not be fully warranted. In the original work (Henry, 1979, 1981), 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. However, the motivation for the work is also reported as the high cost of software maintenance (Henry, 1981), the high cost of software development (Henry, 1979), improving software reliability (Henry and Kafura, 1984), providing quantitative guidelines for software designers (Henry et al., 1981), and controlling software complexity (Henry, 1979).

Nowhere is this confusion more evident than in the statement of the problem in Henry's doctoral dissertation (Henry, 1979): "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."

Given the large variety of purposes that this relatively simple metric is intended to serve, it is perhaps not unsurprising that the results outlined in Table 3 are so mixed. Before looking at some theoretical problems with the metrics, it is worth examining the statistical methods used. The empirical work, cited by Henry and Kafura to support the metric, is based on a change analysis of parts of the UNIX operating system. Very high correlations are reported, for example, Spearman $r = 0.94$, $p = 0.0214$ (Henry, 1981), and Pearson's product moment $r = 0.95$ (Henry et al., 1981) between the metric and module changes.

Unfortunately, there are a number of problems with this:

- Spearman and Pearson tests are inappropriate given that the assumption of normally distributed data points is scarcely credible for change or error data.

- Henry and Kafura rather misleadingly refer to change data as "error data" on a number of occasions; a model of maintenance work is unlikely to be isomorphic to a reliability model.

- Henry and Kafura's discovery that information flow is outperformed by McCabe's cyclomatic complexity metric (Henry et al., 1981). This seems to suggest that a more easily extractable metric is more effective than the information flow metric.

- There are problems with consistency of reported results. Kafura gives different correlation coefficients for the same study (Kafura and Henry, 1981). Reference back to Henry's original thesis (Henry, 1979) suggests that the later paper (Kafura and Henry, 1981) contains typographical errors.

- The technique of logarithmic class interval analysis used by Henry is a surprising one to use. Logarithmic class analysis loses a lot of data, resulting in what is at best a weak ordering. Also, it decreases the number of data points from 165 to 8, making statistical significance harder to obtain and trends less reliable. Unfortunately, insufficient data are 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, not 0.94. The other major result that Henry and Kafura present is that the length, or the internal module complexity

### Table 3. Empirical Validations of Information Flow

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Henry, 1981</td>
<td>Changes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Kafura and Canning, 1985</td>
<td>Coding time + errors</td>
<td>No</td>
<td>n.a.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Kafura and Reddy, 1987</td>
<td>Maintainability</td>
<td>Equal</td>
<td>Yes</td>
<td>n.a.</td>
<td>Yes</td>
</tr>
<tr>
<td>Rombach, 1987</td>
<td>Modifiability</td>
<td>No</td>
<td>n.a.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Maintainability</td>
<td>Yes</td>
<td>n.a.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Comprehension</td>
<td>Yes</td>
<td>n.a.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Locality</td>
<td>Yes</td>
<td>n.a.</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>Kitchenham, 1988</td>
<td>Errors</td>
<td>No</td>
<td>Weak</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>Shepperd, 1988b</td>
<td>Development effort</td>
<td>Yes</td>
<td>Weak</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Shepperd, 1989</td>
<td>Maintainability</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

n.a., Not applicable.
Three out of the six remaining investigations find useful relationships between the metric and a variety of software quality factors. A fourth study indicates that the module length component of their metric, actually detracts from the performance of the metric (Henry, 1981); without length and by use of class intervals based on $10^n/2$, a Spearman correlation of $r = 0.98$ was obtained.

There have been other validations of the metric. Three out of the six remaining investigations find useful relationships between the metric and a variety of software quality factors. A fourth study indicates that for three out of four maintenance factors, the metric proved to be an effective indicator (Rombach, 1987). Kitchenham (1988) found only weak relationships and superior performance by use of more traditional code-based metrics. A sixth study found that for three out of four maintenance factors, the metric has identified some relationship, but that it is less compelling than it might appear at first sight.

These results suggest that the information flow metric has identified some relationship, but that it is less compelling than it might appear at first sight. We contend that the results given in Table 4 support the view that a powerful idea underlies the metric, but considerably more work both in refining the metric and strengthening the statistical case is needed. These, then, are the statistical problems with the metric. There are also some major theoretical problems, which are detailed below.

One of the first theoretical problems is the definition of flow, particularly indirect flow. As with the software science metric and cyclomatic complexity, informality predominates in the Kafura and Henry metric. For example, there are no definitions of global data and procedures. Many of the definitions are ambiguous and capricious.

A second problem is that many local indirect flows can only be detected by a complicated internal analysis of a module. Unfortunately, such information is unlikely to be available during the system design stage of software development.

A further problem is that definition of local indirect flows (Kafura and Henry, 1981) appears to recognize only flows over two levels of a system structure. There is no good reason why the number of levels should be restricted to two. However, simply extending the number of levels included is unsatisfactory, because the outcome is potential over-counting, with the consequence that all modules are linked to all other modules. Therefore, one must have grave reservations concerning the validity of indirect flows. This is in accord with Kitchenham (1988), who decides to omit such flows from her empirical investigation of design metrics.

There is another problem in that the equation for the metric is poorly formulated because 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 ELOC. Although 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” (Henry, 1979), this still leads to the fact that the model as it stands penalizes the reuse of any module that exports or imports any information due to its quadratic nature—something counter to conventional software engineering practice.

Another problem arises from anomalies in the treatment of parameterized communication compared with communication via global data structures. Despite defining global information flows, Henry and Kafura fail to incorporate them into their definition of fan-in and fan-out. Instead, they merely use them as a count of data structure accesses. This potentially has a considerable impact on the metric.

One problem arises from the simplistic assumptions made about the metric. For example, all flows are considered to have uniform complexity, but the information might be a simple Boolean or a complex

<table>
<thead>
<tr>
<th>Order of Complexity</th>
<th>Number of Procedures</th>
<th>Number of “error” Procedures</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>38</td>
<td>12</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>41</td>
<td>19</td>
<td>46</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>19</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>15</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>11</td>
<td>92</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>67</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Totals</td>
<td>165</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>
structure containing many record variants; the metric is not sensitive to the difference. Thus, complex connections between modules that, for example, could result in higher change effort being expanded are ignored.

The use of the metric as a design measure is also problematic because of the late availability of the length term. Henry and Kafura raise the possibility of refining this measure by replacing it with either Halstead's E measure or McCabe's cyclomatic complexity (Henry and Kafura, 1984). However, given that these measures are still only extractable from program code, this would not represent much of an improvement.

There is also the problem of definition. For example, entities such as global data structures and parameters are undefined, and only a hazy definition of procedure is offered.

These theoretical problems lead to the metric being difficult to apply and analyze (Ince and Shepperd, 1988) and must, together with some of the questionable statistics used, explain the extent to which empirical results differ.

In conclusion, there are two main problems with Henry and Kafura's information flow metric. The authors have no clear idea of what they are modelling, and then seem to commute between a variety of problem domains in an uncritical way. 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 arbitrariness. Many of the inconsistencies and anomalies contained in their model could have been avoided had a more formal approach been adopted.

5. MODELING AND METRICATION

5.1 The Nature of Models

Before bringing together the problems outlined in our dissection of the three metrics, it is worth providing a short diversion on the process of model building. A model is an abstraction or simplification of reality. Thus, the first concern when a model is developed is that it serves some stated purpose. Without a purpose, the process of building a model is severely limited. For example, the main rationale in modelling is to exclude areas not relevant to an area being addressed; without a purpose, we do not have a starting point when considering factors that need to be included and excluded. Good examples of modelling being attempted without a dimension of purpose are attempts by academics to develop mathematically rigorous models, for example, Bache and Tinker (1988), Fenton and Whitty (1986), Fenton and Kaposi (1987a, 1987b), and Fenton (1991, 1992) in which, while addressing one of the problems outlined in previous sections of this article, the fuzziness of metric specifications, the researchers seem to have ignored the problem domain.

Second, a model will establish relationships between various entities. This relationship holds between inputs and outputs, sometimes known as endogenous and exogenous variables. These are linked by relationships, or a set of equalities, which may use parameters. This is illustrated in Figure 2 which is taken from Shepperd and Ince (1993).

Third, a model should also describe the relationship between the real world and the inputs and outputs identified. This establishes the 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 nothing but metaphysical.

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

It is a general principle that, as models are generalized across larger domains, they require increasing numbers of inputs and, hence, grow in sophistication. Given our limited understanding of the software development process, it would seem ambitious to continue to propose the type of all-encompassing, all-purpose, yet remarkably simplistic models that
have abounded in the literature, for example, Card and Agresti (1988), Halstead (1977), Henry (1981), and McCabe (1976), without consideration of the limitations and applicability of the model.

The fourth component of a model is its limitations. These are best expressed in terms of assumptions that 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 validations the software science model (Halstead, 1977). It is clear that this model assumes a small-scale programming environment and a third-generation 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 the software. Subsequent work, for instance by Basili and Phillips (1981), involving large-scale, team-based software development highlights the ambiguities lurking behind the term "effort": for example, should requirements analysis be included, or for that matter, changes in requirements during development?

The fifth component of a model is its veracity. 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. One alternative is to specify the degree of accuracy, as in the COCOMO model, in which it is suggested that there is a 70% probability of the model being accurate to within ±20% (Boehm, 1981).

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. It will contain

- an adequate definition of input and output variables;
- mappings between the real world and the model, thereby allowing real-world use;
- relationships linking inputs to outputs;
- assumptions that are made and, thus, delineation of the domains to be modelled.

We claim that many of the problems encountered with the three metrics described in this article are attributable to the fact that one or more of these modelling dimensions is missing, and that the poor empirical validations that have arisen are a direct consequence of these omissions, although it is worth pointing out that the statistical methods that have been used have been somewhat wanting (Hamer and Frewin, 1982).

6. SUMMARY
This article has revealed a recurring pattern of ill-conceived and poorly articulated models that underlie the metrics presented above. This, in turn, has led to metrics 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 because of the lowly role ascribed the model. Directly stemming from modelling weaknesses are the problems of empirical validation. Empirical validation 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, in particular, that of modelling methodology. It is evident from the survey of metrics research described here that the general tenor of the work has been a preoccupation with detail, arguably at the expense of higher level modelling 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 topics will be briefly reviewed.

The first issue is, what does a metric mean? In this article, there are many references to models and underlying models, because it is only within the context of a model or a theory that a measurement has a meaning (Kyburg, 1984). All the metrics reviewed here have relied on implicit ideas, unspoken assumptions, and partial definitions in terms of their underlying models. Consequently, there is a need for the development of a more formal framework for software metric models.

The second issue is model evaluation. Because, 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. Unfortunately, there has been scant regard paid to model evaluation. We contend that is 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 overstressed: metrics based on flawed models are worse than valueless: they are potentially misleading.

The third issue is the clear need for some method to guide software engineers in the selection and
A Critique of Three Metrics

tailoring of software metrics to a particular application that involves measurement goals. In the past, there has been the unfounded belief in “general complexity metrics” of one form or another that can be used for any application or environment and has led to the view that metrics are just an off-the-shelf commodity.

These, then, are the issues. This article would be incomplete without mentioning the fact that metrics development work that is relevant to model building and evaluation is being carried out. For example, the development of the GQM method by Victor Basili and colleagues at the University of Maryland is an attempt to ask hard questions about purpose before developing and using a metric (Basili and Rombach, 1988; Rombach and Ulery, 1989). Also of note is the COQUAMO methodology, which uses a similar approach to Basili’s based on a hierarchy of software quality factors (Kitchenham, 1987). It attempts to convince the software engineering community of the limited utility of general testing metrics (Myers, 1992), and includes some design metrics modelling work aimed at managers (Cádenas-García and Zelkowitz, 1991). We have also defined a more formal approach to metrics definition and evaluation based on an axiomatic approach (Shepperd and Ince, 1993).

However, there is still considerable evidence from work reported in the last three years that metrics researchers are still ignoring the core modelling issues that lie at the heart of measurement by adopting scatter gun approaches (Munson and Khoshgoftaar 1990, 1992a, 1992b) using metrics whose utility has been widely questioned (Henry and Wake, 1991; Leach, 1990), not providing any idea of the model on which they are based (Reynolds and Maletic, 1990; Ejiogu, 1991), and developing tools based on questionable metrics (Tsaldis et al., 1992). Also, there is still quite a large amount of acceptance of the metrics in general texts on software engineering, for example, in Pressman (1992), where it is stated that for software science, “good agreement has been found between analytically predicted and experimental results,” and in Yourdon (1992), where the source work for software science is nominated as one of the 87 essential books for American software developers.

Nowhere is the evidence of modelling being neglected than in the newly emerging subfield of metrics, which addresses itself to object-oriented technology. For example, Maus (1992) develops a mathematical model for metrification of the maintenance process based on information theory, which is almost devoid of purpose; Chidamber and Kemerer (1992) describe a metric suite that seems to have no effective mapping to the real work and in which the only link to reality is the statement that the metrics suite was developed for “measuring elements contributing to the size and complexity of object-oriented design”; and Rajaraman and Lyu (1992) describe a mathematical model that again does not specify a purpose and in which the definition of the counts used are, at best, fuzzy.

It was these concerns, together with a desire to point out to authors whose articles and manuscripts we regularly referee a single reference source that details many of the problems that occur during the development and validation of software product metrics, that motivated this paper. Although occasionally some criticisms emerge of current metrics work within a limited domain, for example, Rombach (1990), we hope that this article can be regarded as a work that draws these together into a coherent whole.

REFERENCES


Kafura, D., and Canning, J. T., A validation of software metrics using many metrics and two resources, in *Pro-


