Full citation: MacDonell, S.G. (1991) Rigor in software complexity measurement experimentation, Journal of Systems and Software 16, pp.141-150.
doi: 10.1016/0164-1212(91)90008-T

Rigour in Software Complexity Measurement Experimentation

Stephen G. MacDonell[a, b]

[a]*Computer and Information Science, University of Otago, New Zealand*
[b]*Department of Engineering, University of Cambridge, England*
stevemac@commerce.otago.ac.nz

Abstract

The lack of widespread industry acceptance of much of the research into the measurement of software complexity must be due at least in part to the lack of experimental rigour associated with many of the studies. This paper examines thirteen areas in which previous empirical problems have arisen, citing examples where appropriate, and provides some recommendations regarding more adequate procedures.

1. INTRODUCTION

Far too often, advances in quantitative software assessment have their validity questioned because of experimental issues. Despite widespread recognition of many of the problems which can and do occur in empirical procedures (e.g. see [51]), it would appear that this awareness is still being disregarded in the rush for immediate results. The final line in many studies is therefore `More experiments are therefore needed...'.

There are several areas in which problems have arisen in the past; thirteen are examined in the following discussion: pre-experiment design, operational definitions, experimental method, subjective assessment, data collection, program sizes, program sample sizes, languages investigated, subjects, confounding factors, statistical validity, result interpretation and publication of data and assumptions. Although the examples cited in the discussion concern complexity measurement experiments, the comments and recommendations could be applied to empirical procedures in most software measurement domains.

2. AREAS OF CONCERN

2.1. Pre-experiment design

It is imperative for any research involving some degree of empirical activity that the tasks and procedures which make up this work are planned and documented before the work begins. At the very least, this helps to ensure that the true objectives of the work are being addressed, in that data is derived in order to perform the experiment or validation, rather than *vice versa*. It is clearly unscientific to design experiments or validation around the data which has been obtained - this can clearly bias the results.

Kearney et al [37] comment that due to the multi-dimensional nature of software measurement, many studies have examined a large number of variables and properties in an attempt to determine feasible hypotheses, rather than testing previously defined predictions. This, they say clearly increases the chances of finding accidental relationships which cannot be generalised to similar studies. Confidence in the results obtained is therefore significantly reduced.

Furthermore, due to poor experimental designs, many studies which are supposed to be tests of given hypotheses are actually incapable of discarding them - according to Hamer and Frewin [28], they simply do not have the power to identify false hypotheses.

Basili et al [5] have therefore provided some guidance on the initial stages of experimental research:

1. Experiment definition - precise specification is required in formulating the problem(s) into documented goals
2. Experiment planning - designers should already be looking towards larger replications of the experiment to reinforce the findings. Factors which are thought to be influential should be determined and recorded at this stage.

Appropriate procedures should also be chosen and clearly documented at this point, keeping in mind the goals of the experiment, the sample size and characteristics of the sample itself.

2.2. Operational definitions

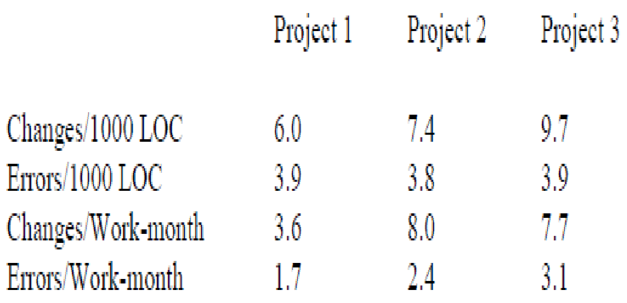
Quantitative software assessment becomes especially difficult with the lack of operational properties which adequately and accurately represent the attributes under investigation. For example, we may be interested in the quality of an emerging system; however, how quality can be objectively measured from aspects of the software product is still unclear.

A similar situation exists for complexity quantification. Ince and Shepperd [33] state that there are many metrics which are said to provide estimates of complexity, but that are in fact derived and validated based on a wide range of actual operational features e.g. construction time, debugging time, the number of errors or control flow in the code.

The result of this lack of clarity in reference to appropriate operational attributes is that important predictive relationships may be missed, as evaluators cannot or do not measure the actual criteria which the metrics are said to estimate [12].

Another practice which causes problems in model validation involves the substitution of one attribute for another, depending on what data is available. For example, Henry and Kafura [32] use program changes as an equivalent substitute for development errors in the empirical validation of their *information flow* metric. Justification for this is taken from previous studies - they cite work by Basili and Reiter [4], who remarked that program changes were "...a reasonable measure of the relative number of programming errors...". This may have been appropriate for this particular study. However, the validity of the assumption could be questionable. A study by Weiss and Basili [59] has shown that changes and errors can be significantly different (Table 1).

TABLE 1. Code changes versus coding errors



	Project 1	Project 2	Project 3
Changes/1000 LOC	6.0	7.4	9.7
Errors/1000 LOC	3.9	3.8	3.9
Changes/Work-month	3.6	8.0	7.7
Errors/Work-month	1.7	2.4	3.1

The problems associated with determining adequate quantitative indicators of aspects such as quality and maintainability will be difficult to overcome. It may in fact be impossible to define a general set of properties which accurately reflect such attributes in every case. Aspects of interest (such as the number of development errors) which can be measured however, must be measured, and not replaced by other countable features, no matter how closely related they may be.

2.3. Experimental method

Also of particular importance are the tests which are chosen to provide the required data, particularly in psychological complexity studies. In terms of research into this avenue of complexity (as reflected by programmer ability), problems often occur in determining an appropriate measure of software understanding. As an illustration, it is suggested by Curtis et al [13] that the most sensitive measure of a programmer's understanding is the ability to follow code structure and to reproduce functionally equivalent code. To this end they adopt as their dependent variable the percentage of statements correctly recalled. The obvious flaw in this approach is that the test becomes one of short term memory rather than of long term recall, with individuals often varying in their ability for both tasks. Another study by Dunsmore and Gannon [20] used the exam average of three course tests as the supposedly independent measure of student programmer ability. This fails to consider however, that many students perform differently under an exam situation. The adequacy of the results obtained in providing generalisable conclusions could therefore be questioned.

In most experiments in this domain, programs are created or viewed for the first time during the experiment itself. Davis [14] claims that this differs markedly from the common situation under which many programmers work i.e. concentrating on the same group of programs over a period of time, increasing their familiarity and therefore influencing their understanding and performance. Hence an artificial testing situation is used in experiments employing these types of procedures.

Di Persio et al [18] suggest that there are several `typical' tests:

1. to construct a program from a given specification
2. to make certain changes to a program
3. to locate and/or correct one or more deliberate bugs
4. to study a program and then answer questions about it.

These are said to be of limited use however, as the results obtained are dependent on the particular program domain used in the study at hand. They are also difficult to prepare and mark. The authors therefore suggest memorisation/reconstruction tasks as more appropriate. To ensure that these are not simply tests of syntactic memorisation, it is recommended that the recalled-lines procedure should be tailored to the environment in which the programmers normally work.

As mentioned in the initial part of this discussion however, memorisation/reconstruction-type tests are also not ideal. Further evidence of the inadequacy of these methods is provided by Krall and Harris [in 42]. They examined the effect of various readability improvement strategies using a comprehension quiz and a reconstruction test. Significant results were obtained only under the former scheme, casting doubts upon the use of memorisation assessment to reflect understanding.

Other works however, have also illustrated shortcomings in the quiz approach. Iyengar et al [34] conducted a study investigating the complexity of understanding using various data structures. They chose error location methods for their test, claiming that comprehension quizzes only test short term recollection ability (as suggested above) which can be affected by, among other things, data variable names and the extent of internal documentation. Weissman [60] also concluded that comprehension tests were inadequate for assessing psychological complexity.

What are suggested as the three most widely used approaches for measuring the complexity of understanding have been discussed in some detail by Haas and Hassell [26]. They cite several problems with the use of the first technique, code reconstruction. The initial problem concerns the inaccuracy and subjectivity of deciding what is actually a correct reproduction - is the substitution of functionally equivalent constructs incorrect, or simply a different approach? Is omitting a simple statement as serious as leaving out an important control flow structure? Can partial credit be awarded for certain degrees of reconstruction? Furthermore, reconstruction ability has been found to be related in many instances to the experience of those being tested, yet this is not considered in most cases, because of the significant difficulty in effectively measuring relative levels of expertise.

The second technique investigated by Haas and Hassell [26] is that of program error location/correction. Only one bug is used in most cases of error location, creating an artificial situation in comparison to normal development/maintenance circumstances. Problems also occur in error correction tests, particularly in the assignment of marks for partial correction or when the correction of one error creates another. What is more, this is generally not a test of understanding, but rather one of problem-solving ability [34].

The time required for the completion of program tasks is the third method discussed by Haas and Hassell [26]. Their main objection to this type of criteria is the obvious slant which it has towards quick workers. That one programmer completes a given task faster than another does not ensure a fuller understanding of that task. What is more, how should relative grades be assigned for an incomplete but accurate product as compared to a complete but inaccurate one?

Clearly there are flaws with all three approaches. Haas and Hassell therefore propose a different approach. They suggest that subjects should be provided with the code, documentation and sample output data for a system. Questions may then be put to them concerning the manipulation of input data, the resulting output, the overall functionality of the program and of parts of the code and the possibility of enhancements and the effects that these may have. This approach would appear to overcome at least some of the problems associated with other methods.

2.4. Subjective assessment

The subjective nature of many studies is unfortunate and often misleading, given the fact that objective results are said to have been achieved in many cases. It is understandable that where human endeavours are considered (such as programming), differences among individuals will occur. However, the actual *assessment* of aspects of the software product must be approached quantitatively to lessen the influence of personal attributes and perceptions.

Tanik [55] investigated the relationship between subjective programmer-assigned complexity levels and such aspects as the number of runs to complete a program and the time spent in debugging a program. The correlations obtained (from limited samples) were weakly positive. In spite of the low explanatory powers observed, Tanik suggested that programmers can in fact guess complexity and that this `ability' might be used in development time estimations. This conclusion seems unjustified given the actual results obtained, and highlights the lack of accuracy of subjective evaluations.

Subjective rating has also been used in several other studies. In their investigation of metrics and software maintenance, Kafura and Reddy [36] used subjective ratings of complexity, chosen because objective data was not available and to test the accuracy of `expert' judgment. The model of software complexity proposed by Van Verth [56] was also validated using programs which were graded by experts.

This approach can only be valid, however, if it is accepted that human opinion is an adequate discerner with respect to relative levels of software complexity. If this were the case, then what would be the need for objective measures?

Studies which utilise subjective evaluations have always been questioned, despite the fact that `experts' may have often been used. It is therefore essential that if at all possible, quantitative assessment methods be employed.

2.5. Data collection

The general lack of useful and accurate development project data has been widely acknowledged as a major impediment to the effective validation of software complexity research ([46], [49], [44]).

What appears to be a standard approach to data collection involves the use of collection forms which must be completed by those working on the project. An example of this procedure can be seen in Blaine and Kemmerer [7]. A Software Change Proposal (SCP) was required (over the six year project duration) for any modification to be made to the system. Each proposal outlined a reason for the change, a list of all the routines affected and an estimate of the man-hours required to complete the total modification. Rodriguez and Tsai [48] employed a similar approach in the collection of error data for their study, as did Vessey and Weber [58].

Although these are valiant attempts at obtaining accurate data, several flaws can occur: those involved may (willfully or otherwise) forget to submit the relevant forms, or incomplete and/or inaccurate data may be similarly provided (it is common for programmers not to report clerical-type errors [3]); problems with the allocation of error/change quantification data may also occur when, for example, the change made affects more than one section of code [6].

The previous remarks relate only to the acquisition of data pertaining to the coding and maintenance stages. It would appear that the acquisition of early development phase data is even more difficult [44]. This may be due in part to the often inconsistent points chosen to begin data collection in various projects [31], or to the lack of currency of the data [46]. When, for example, the error history of a project is quite old, Potier et al [46] state that in the event of missing data, recovery or recollection of this data is extremely difficult and that in any case incorrect data may be obtained due to changes in the collection procedures used over the period.

Collecting useful data is clearly difficult. For one thing, our models of the development process may be inadequate as to even permit sufficient levels of data quantification [57] and for another, the data which is obtained may be of such a questionable nature that only narrow and qualified acceptance can be made for much of the research [65].

Rault [47] claims that the problems which are encountered are a result of two inter-related issues; on the one hand, the lack of accurate and timely data prevents full validation of any models suggested; on the other hand, the absence of sufficiently valid models restricts the determination of what data is actually required. Rault suggests that to remedy this situation, the systematic collection of large volumes of data, albeit empirically, should be the first step. Basili and Phillips [3] state however, that data **must** be obtained from commercial development environments, rather than from experimental work. The following methodology for achieving this is suggested by Weiss and Basili [59]:

1. determine aims of research
2. develop questions to satisfy aims
3. define data collection form(s)
4. derive collection procedures
5. collect, validate and analyse the data acquired.

This approach may too be afflicted by the problems outlined previously. However it would seem that a defined procedural approach involving commercial development must be adopted. The initial objective relating to this issue is therefore to obtain full cooperation from the company (or companies) and their development personnel. This alone will significantly increase the likelihood of obtaining complete and accurate data.

2.6. Program sizes

The issue of program size appears in two guises within complexity research; firstly when programs are used as tools for the provision of data e.g. investigating bug location times; and secondly as a medium for metric validation. The use of insignificant programs (in terms of their size) for both purposes has seen criticism.

Sayward [50] suggests that material selection has been poor in several data collection experiments, with the programs used being too small to reflect `real-world' circumstances. It is stated that this is due in many cases to economic constraints. Weissman [60] remarks further that it is difficult to find programs of a size which falls between trivial and manageable.

Examples of small-scale experimentation can be found in Woodfield [61] and to a lesser extent in Henry and Kafura [32]. Although in the latter study the sample of procedures was significant *as a system*, the length of 53% of the modules examined was less than twenty lines, and the largest was just 180 lines long. If larger procedures had been analysed, a different metric may have been developed, or more significant evidence may have been derived for the validity of the authors' findings.

The effectiveness of many proposed metrics is also often illustrated using very small programs. This, according to Dunsmore [19], places in question the applicability of the measures to large modularised software systems, particularly since it has been acknowledged ([8], [54]) that large systems exhibit different properties to those of smaller counterparts. The study by Henry and Kafura [32] mentioned above also provides an example of this practice. Another is Gordon's work on software clarity [24]. The fifteen programs used to illustrate the effectiveness of his measure were all between just three and twelve lines long.

It is acknowledged by Dunsmore [19] that it is simply not practical to validate a measure for all program sizes, languages and applications. He suggests that compromises are inevitable, but that they should be made sensibly e.g. generalisation by size within a specific language may be acceptable. This again assumes however, that a change in size is only that i.e. that no other aspects are affected. This is despite widely held principles which encourage the use of modularised structure for larger systems, when they would not be employed for smaller segments of code.

It is clearly important that future empirical studies use programs which accurately reflect the domain to which the measures are said to apply. The most satisfactory approach would appear to be the use of programs and systems of varying sizes, to ensure general applicability for the measure(s) being validated.

2.7. Program sample sizes

Studies in this area also frequently suffer from severely limited program samples on which determination or validation of measures is performed. Sample sizes should be at least large enough to enable significance tests to be carried out. Lister [39] comments that if this is not achieved, results may become overly dependent on a few extreme observations.

As an example, Zweben [66] attributes some degree of the `overselling' of the *software science* quantification theory [27] to results based on small samples. The sample used by Gordon [24] in the development of a clarity measure is similarly criticised by Evangelist [22] for its lack of comprehensiveness.

DeMarco [17] acknowledges that the validation of his cost prediction model was performed with small data sets. He suggests however, that the use of partially validated objective models (such as his) should be judged against the alternative, that is, subjective estimation. The implication is that partly objective models are better than none at all, or guesswork. This comment has since been extensively cited as justification for incomplete validation, yet its general applicability may be questionable. A model based on a few extreme or widely dispersed observations may be significantly worse than one based on information provided by skilled and experienced personnel.

Ideally however, large replicable samples are necessary for reliable model development and validation. Henry and Kafura's suggestion [32] that a test-bed should be high-level, well-documented, large, real and reusable is clearly appropriate for work in this area.

2.8. Languages investigated

Under ideal circumstances, research should be validated over a wide range of programming languages. To a large extent, this has not occurred in previous work - Han et al [29] remark that conventional complexity metrics are largely dependent on data derived from lower to medium level procedural language programs.

It would appear that, in particular, the study of metrics as applied to COBOL systems is needed. Côté et al [11] and Gibson and Senn [23] suggest that the small number of investigations on metrics using this language fails to correspond with its usage in industry. Many business applications are written in it and so a large degree of maintenance is performed on COBOL applications [45]. Another area which should be investigated is the use of metrics as applied to software developed in a fourth-generation environment, particularly as the use of these products becomes more widespread [11].

It is likely in fact, that some metrics may perform poorly when applied to certain languages, or they may simply be inapplicable for particular language types. For example, both *software science* [27] because of its syntactic focus and McCabe's *cyclomatic complexity* [41] with its control flow derivation are likely to be inappropriate for software developed in a 4GL environment. It is therefore probable that new measures will be developed for use with this and other new environments. In any case, it is most important that appropriate measures are used according to the language or environment of the software being assessed.

2.9. Subjects

Sayward [50] remarks that subject selection is one of the two most problematic aspects of psychological complexity research. Due to resource constraints, many studies in this domain have used students as their means of acquiring data. (Clearly this is also due to the fact that much of the research is performed at universities, with a `captive' subject sample.) This practice however, has been widely criticised as students are often considered to be in fact atypical of the overall population [33]. Halstead [27] and Gordon [24] both remark that programs written by novice programmers tend to contain a high proportion of impurities. This would suggest then that researchers who use such an approach must be especially cautious in developing generalisable conclusions from their results [52].

There are several other flaws associated with this approach. One which has been openly acknowledged is the extent of variation which may occur among individuals. Empirical work in the programming domain is said to be very difficult [24] given the inevitable and often enormous variations in individuals' abilities ([10], [16], [5]). Shepperd [53] remarks that statistically significant results can be severely undermined due to these variations in ability, with a few extreme observations lessening the importance of the majority.

What is more, student development projects are often restricted to insubstantial programming exercises [7] and are consequently not indicative of a normal development/maintenance environment. Therefore the results obtained should not be (but often are) generalised to this situation.

Weissman [60] has suggested then, that a wide range of subjects with varying backgrounds be used. However, combining subjects with differing experience levels has been criticised elsewhere (e.g. [40]). Furthermore, there is little reason to believe that experienced and novice programmers work in the same way, albeit at different speeds. Combining results from both groups to develop general conclusions may therefore also be of questionable validity.

Although the ideal situation would employ a large sample of professional developers of roughly uniform ability, restrictions on time, money and other resources coupled with the low chances of the availability of such a sample, and the relative availability of a student populace, all contribute to the likelihood of further use of student samples. It is important then, that the students chosen are at a senior level and are of approximately similar ability and that the sample is sufficiently large as to lessen the effect of individual observations.

2.10. Confounding factors

Due to the large number of factors which may be considered in software engineering experimentation (and the subjective nature of many of these), straightforward analysis of results is often difficult. Factors such as development methodology, programmer ability and experience, problem difficulty, development environment and the availability of development tools all vary across projects and are difficult to express and consider in a quantitative manner ([43], [37]).

An initial step in overcoming this problem is to acknowledge the existence of uncontrolled factors in the study, as in Davis and LeBlanc [15] and in Woodfield et al [63]. Although praiseworthy, this unfortunately does little to enhance the credibility and usefulness of the studies' findings.

Clearly, experiments where all variables can be controlled are the most likely to provide `pure' observations. This situation is unlikely to be achieved in a domain such as software development however, due to the human factors which are often of significant influence. Therefore the effect of all other factors which can be controlled and are of no direct interest to the study must be minimised. For example, Woodfield et al [62] made all variable names meaningless and removed all program indentation in their investigation of programmer understanding under various comment and modularisation schemes.

2.11. Statistical validity of predictive relationships

Criticism of this aspect of software complexity experimentation has been centred mainly around the use of the correlation coefficient (usually Pearson's product-moment correlation coefficient *r*) as the main determinant of predictive relationships between two variables.

Rodriguez and Tsai [49] remark that only rarely have metrics been fully validated in a statistical sense. The extent of validation for many metrics is limited to finding linear or rank correlations to determine a relationship between the measure and a system attribute.

The inadequacy of this approach has been extensively illustrated. Kearney et al [37] adopt a practical argument; even when a strong correlation is discovered, the applicability of the result may be minimal. For example, they rightly suggest that it would be a gross misjudgment to conclude that software size predicts the occurrence of bugs, or that the size of all software should be reduced, simply because a linear relationship is found between size and errors. Large size is an obviously inherent property of many applications.

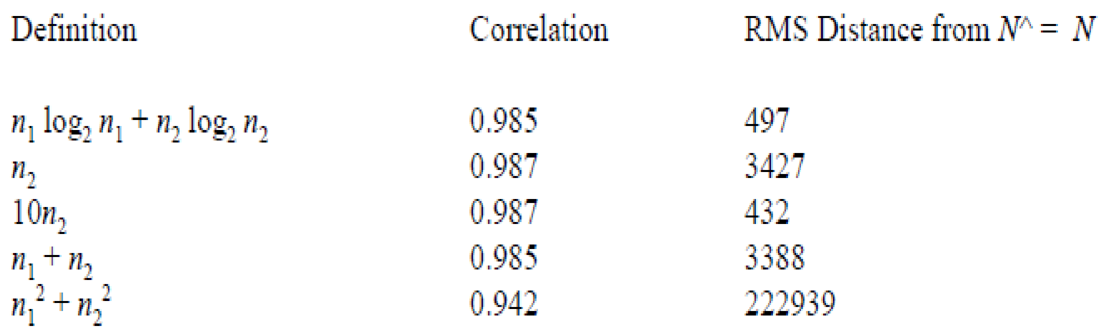
Knafl and Sacks [38] also illustrate the often unjustified importance which is attached to high correlations. They examined the data used in Albrecht and Gaffney's *function point* study [1] and showed that for one data set three sparsely spaced large projects were overly influential in the correlation obtained; removal of those data points resulted in a 44% decrease in the correlation, substantially reducing the significance of the results.

Lister [39] also points out that although correlation is a measure of random variable relationship, few experiments have even attempted to prove the random nature of the variables involved. Furthermore, he suggests that even if randomness were proved, a high correlation only indicates the existence of a relationship, not what the actual relationship is. The use of linear regression techniques would provide an accurate assessment of the relationship itself, yet Lister states that this procedure is seldom even mentioned in research reports.

To reinforce the danger of unquestioned reliance of correlation coefficients, Lister provides the following table (Table 2) showing relationships between predicted and actual software lengths (*N*^ and *N* under Halstead's *software science* [27]) for a sample of thirty-one PL/1 programs [21], for various definitions of predicted length. Although the correlations are comparable, the errors associated with the predictions vary significantly.

Another problem with the use of correlation measures in this domain is the need for the variables investigated to come from (approximations to) normal distributions. The impossibility of obtaining a negative number of errors is cited by Shepperd [53] as a situation where a skewed distribution result, particularly for small sets of data. Transformations which could lessen the effects of such a distribution can be easily applied, but these are seldom performed.

TABLE 2. Correlation and error data for actual and observed lengths



Definition	Correlation	RMS Distance from $N\hat{} = N$
$n_1 \log_2 n_1 + n_2 \log_2 n_2$	0.985	497
n_2	0.987	3427
$10n_2$	0.987	432
$n_1 + n_2$	0.985	3388
$n_1^2 + n_2^2$	0.942	222939

The extent to which the correlation coefficient pervades metric research is further illustrated in a study by Woodward [64]. The author acknowledges that there are well publicised pitfalls in using the Pearson correlation coefficient to confirm metric relationships, citing previous studies. He then states however, that the method has become the *de facto* standard practice for work in this area and so presents his results using the Pearson procedure without further comment.

As much as the correlation coefficient may adequately indicate the degree of a linear relationship between two random variables, it fails to provide any support for predictive relationships which may be derived from it. Therefore other tests should be used in conjunction with this procedure so that the problems alluded to above can be avoided. This could involve the use of regression techniques and the examination of the relative and absolute errors associated with the predictions to enable determination of the actual relationship(s) and the explanatory powers of the predictive models used.

2.12. Result interpretation

Testing methods are clearly important in software engineering experiments, but correct result interpretation is an equally vital and difficult task [10]. In several cases researchers have adopted different interpretations of the same model, making the comparison of results meaningless [53].

Kearney et al [37] cite Basili and Hutchens' study [2] as an example of result misinterpretation. The authors suggest that unjustified general conclusions were drawn, as they were based only on the results obtained from a single programmer concerning the relationship of program changes and the *SynC* complexity measures.

Incorrect interpretations are also often perpetuated by subsequent studies e.g. Jensen and Vairavan [35] report that Henry and Kafura's work [32] related their *information flow* metric to the occurrence of errors, whereas the original relationship was actually between the metric and program changes.

Basili et al [5] suggest that the correct presentation of results assists in their correct interpretation. They also recommend that results and conclusions be qualified according to the particular samples used. An example of this occurs in a study by Blaine and Kemmerer [7]: "The recommended measures of maintenance effort for Rolm assembly language software are..." p.239. This specifies exactly what is being predicted and to what software the results apply. Another good example of result qualification is used by Gremillion [25]. The author states that in drawing conclusions from his study it should be remembered that the research represented only one situation using one language and style, and that generalisation beyond this scope would be inappropriate.

This practice is an important part of the validation process and should be performed in all studies. This would assist in ensuring firstly that those examining the results are aware of the uses and limitations associated with them and secondly that subsequent studies which attempt to replicate the original work have clearly defined boundaries on which to base their research.

2.13. Publication of data and assumptions

The credibility of several studies has also been lessened by their failure to provide the raw or derived data and/or the assumptions on which the results and conclusions were based. Comer et al [9], for example, fail even to provide information on the type and size of the programs which were used in the experiments. This is of vital importance, particularly so repeated comparative experiments can be performed.

3. CONCLUSIONS

This review has highlighted several areas within software engineering experimentation in which more rigid procedures must be developed and put to use. All must be addressed if the validity of results obtained through empirical studies is to be assured.

It is acknowledged, however, that this is by no means a simple task. Several of the criteria are particularly difficult to achieve, for example, the exact, large-scale replication of experiments, and the availability of representative subjects and large program samples. Others may be conflicting - a model based on small data sets is often insignificant; yet the only other option, subjective assessment, is also questionable.

In any case, it is most important that experimenters are constantly aware of the problems which may occur and the limitations that these may place upon the findings obtained. Many studies conclude with a recommendation for further experiments (e.g. [6], [23]). This suggestion is pointless however, unless the formulation, operation and validation of this research is strengthened through more rigorous definition and control of experimental procedures.

REFERENCES

1 A.J. Albrecht and J.E. Gaffney Jr, Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation *IEEE Transactions on Software Engineering* 9 (6), 1983: 639-648.

2 V.R. Basili and D.H. Hutchens, An Empirical Study of a Syntactic Complexity Family IEEE Transactions on Software Engineering 8 (3), 1983: 664-672.

3 V.R. Basili and T.Y. Phillips, Evaluating and Comparing Software Metrics in the Software Engineering Laboratory *ACM SIGMetrics* 10 (1), 1981: 95-106.

4 V.R. Basili and R.W. Reiter Jr, Evaluating Automatable Measures of Software Development *Proceedings Workshop on Quantitative Software Models*, 1979: 107-116.

5 V.R. Basili, R.W. Selby and D.H. Hutchens, Experimentation in Software Engineering IEEE Transactions on Software Engineering 12 (7), 1986: 733-743.

6 V.R. Basili, R.W. Selby and T.Y. Phillips, Metric Analysis and Data Validation Across Fortran Projects *IEEE Transactions on Software Engineering* 9 (6), 1983: 652-663.

7 J.D. Blaine and R.A. Kemmerer, Complexity Measures for Assembly Language Programs *Journal of Systems and Software* 5, 1985: 229-245.

8 F.P. Brooks Jr, No Silver Bullet - Essence and Accidents of Software Engineering *IEEE Computer* 20 (4), 1987: 10-19.

9 J.R. Comer, J.R. Rinewalt and M.M. Tanik, A Comparison of Two Different Program Complexity Measures *ACM SIGMetrics* 10 (2), 1981: 26-28.

10 M.L. Cook, Software Metrics: An Introduction and Annotated Bibliography *ACM SIGSoft* 7 (2), 1982: 41-60.

11 V. Côté, P. Bourque, S. Oligny and N. Rivard, Software Metrics: An Overview of Recent Results *Journal of Systems and Software* 8, 1988: 121-131.

12 B. Curtis, Software Metrics: Guest Editor's Introduction *IEEE Transactions on Software Engineering* 9 (6), 1983: 637-638.

13 B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst and T. Love, Measuring the Psychological Complexity of Software Maintenance Tasks with the

Halstead and McCabe Metrics *IEEE Transactions on Software Engineering* 5 (2), 1979: 96-104.

14 J.S. Davis, Chunks: A Basis for Complexity Measurement *Information Processing & Management* 20 (1-2), 1984: 119-127.

15 J.S. Davis and R.J. LeBlanc, A Study of the Applicability of Complexity Measures *IEEE Transactions on Software Engineering* 14 (9), 1988: 1366-1372.

16 T. DeMarco, Controlling Software Projects New York, Yourdon Inc., 1982.

17 T. DeMarco, An Algorithm for Sizing Software Products *ACM SIGMetrics* 12 (2), 1984: 13-22.

18 T. Di Persio, D. Isbister and B. Shneiderman, An Experiment Using Memorization/Reconstruction as a Measure of Programmer Ability *International Journal of Man-Machine Studies* 13, 1980: 339-354.

19 H.E. Dunsmore, Software Metrics: An Overview of an Evolving Methodology *Information Processing & Management* 20 (1-2), 1984: 183-192.

20 H.E. Dunsmore and J.D. Gannon, Analysis of the Effects of Programming Factors on Programming Effort *Journal of Systems and Software* 1, 1980: 141-153.

21 J.L. Elshoff, An Investigation into the Effects of the Counting Method Used on Software Science Measurements *ACM SigPlan* 13 (2), 1978: 30-45.

22 W.M. Evangelist, Software Complexity Metric Sensitivity to Program Structuring Rules *Journal of Systems and Software* 3, 1983: 231-243.

23 V.R. Gibson and J.A. Senn, System Structure and Software Maintenance Performance Communications of the ACM 32 (3), 1989: 347-358.

24 R.D. Gordon, Measuring Improvements in Program Clarity *IEEE Transactions on Software Engineering* 5 (2), 1979: 79-90.

25 L.L. Gremillion, Determinants of Program Repair Maintenance Requirements *Communications of the ACM* 27 (8), 1984: 826-832.

26 M. Haas and J. Hassell, A Proposal for a Measure of Program Understanding *ACM SIGCSE* 15 (1), 1983: 7-13.

27 M.H. Halstead, Elements of Software Science New York, Elsevier North-Holland, 1977.

28 P.G. Hamer and G.D. Frewin, M.H. Halstead's Software Science - A Critical Examination *Proceedings 6th International Conference on Software Engineering*, 1982: 197-206.

29 W.T. Han, Y.C. Choe and Y.J. Park, Software Metrics Using Operand Type *Proceedings TENCON '87*, 1987: 1212-1215.

30 W.A. Harrison and C. Cook, A Method of Sharing Industrial Software Complexity Data *ACM SIGPlan* 20 (2), 1985: 42-53.

31 S.D. Hartman, A Counting Tool for RPG ACM SigMetrics 11 (3), 1982: 86-100.

32 S. Henry and D. Kafura, Software Structure Metrics Based on Information Flow *IEEE Transactions on Software Engineering* 7 (5), 1981: 510-518.

33 D.C. Ince and M.J. Shepperd, System Design Metrics: A Review and Perspective *Proceedings 2nd IEE/BCS Conference on Software Engineering*, 1988: 23-27.

34 S.S. Iyengar, F.B. Bastani and J.W. Fuller, An Experimental Study of the Logical Complexity of Data Structures *Proceedings 2nd Symposium on Empirical Foundations of Information and Software Science*, 1985: 225-239.

35 H.A. Jensen and K. Vairavan, An Experimental Study of Software Metrics for Real-Time Software *IEEE Transactions on Software Engineering* 11 (2), 1985: 231-234.

36 D. Kafura and G.R. Reddy, The Use of Software Complexity Metrics in Software Maintenance *IEEE Transactions on Software Engineering* 13 (3), 1987: 335-343.

37 J.K. Kearney, R.L. Sedlmeyer, W.B. Thompson, M.A. Gray and M.A. Adler, Software Complexity Measurement *Communications of the ACM* 29 (11), 1986: 1044-1050.

38 G.J. Knafl and J. Sacks, Software Development Effort Prediction Based on Function Points *Proceedings COMPSAC '86*, 1986: 319-324.

39 A.M. Lister, Software Science - The Emperor's New Clothes? *Australian Computer Journal* 14 (2), 1982: 66-70.

40 T. Love, An Experimental Investigation of the Effect of Program Structure on Program Understanding in D.B. Wortman (ed.) *Proceedings 1977 ACM Conference on Language Design for Reliable Software*, 1977: 105-113.

41 T.J. McCabe, A Complexity Measure IEEE Transactions on Software Engineering 2 (4), 1976: 308-320.

42 R.J. Miara, J.A. Musselman, J.A. Navarro and B. Shneiderman, Program Indentation and Comprehensibility *Communications of the ACM* 26 (11), 1983: 861-867.

43 G.C. Moss, Developing a Usable Metric Toolkit for a Commercial Environment *Proceedings 2nd IEE/BCS Conference on Software Engineering*, 1988: 123-127.

44 L.M. Ottenstein, Predicting Numbers of Errors Using Software Science *ACM SIGMetrics* 10 (1), 1981: 157-167.

45 G. Parikh, Programmer Productivity Reston, Reston Publishing Company, 1984.

46 D. Potier, J.L. Albin, R. Ferreol and A. Bilodeau, Experiments with Computer Software Complexity

and Reliability *Proceedings 6th International Conference on Software Engineering*, 1982: 94-103.

47 J.C. Rault, An Approach Towards Reliable Software *Proceedings 4th International Conference on Software Engineering*, 1979: 220-230.

48 V. Rodriguez and W.T. Tsai, Software Metrics Interpretation Through Experimentation *Proceedings COMPSAC '86*, 1986: 368-374.

49 V. Rodriguez and W.T. Tsai, A Tool for Discriminant Analysis and Classification of Software Metrics *Information and Software Technology* 29 (3), 1987: 137-151.

50 F.G. Sayward, Experimental Design Methodologies in Software Science *Information Processing & Management* 20 (1-2), 1984: 223-227.

51 V.Y. Shen, S.D. Conte and H.E. Dunsmore, Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support *IEEE Transactions on Software Engineering* 9 (2), 1983: 155-165.

52 J.A. Shepherd and J.L. Lassez, Opposing Views on the Use of Software Science Measures for Automatic Assessment of Student Programs *Australian Computer Science Communications* 2 (1), 1980: 205-215.

53 M. Shepperd, A Critique of Cyclomatic Complexity as a Software Metric *Software Engineering Journal* 3 (2), 1988a: 30-36.

54 M. Shepperd, An Evaluation of Software Product Metrics *Information and Software Technology* 30 (3), 1988b: 177-188.

55 M.M. Tanik, Two Experiments on a Program Complexity Perception by Programmers ACM SIGPlan 15 (9), 1980: 64-66.

56 P.B. Van Verth, A Program Complexity Model that Includes Procedures *Proceedings 11th International Conference on Software Engineering*, 1987: 252-258.

57 J.M. Verner and G. Tate, Software Sizing and Costing *Proceedings 9th New Zealand Computer Conference*, 1985: 287-304.

58 I. Vessey and R. Weber, Some Factors Affecting Program Repair Maintenance: An Empirical Study Communications of the ACM 26 (2), 1983: 128-134.

59 D.M. Weiss and V.R. Basili, Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory *IEEE Transactions on Software Engineering*, 1985: 157-168.

60 L. Weissman, Psychological Complexity of Computer Programs: An Experimental Methodology *ACM SIGPlan* 9 (6), 1974: 25-36.

61 S.N. Woodfield, An Experiment on Unit Increase in Problem Complexity *IEEE Transactions on Software Engineering* 5 (2), 1979: 76-79.

62 S.N. Woodfield, H.E. Dunsmore and V.Y. Shen, The Effect of Modularization and Comments on Program Comprehension *Proceedings 5th International Conference on Software Engineering*, 1981a: 215-223.

63 S.N. Woodfield, V.Y. Shen and H.E. Dunsmore, A Study of Several Metrics for Programming Effort *Journal of Systems and Software* 2, 1981b: 97-103.

64 M.R. Woodward, The Application of Halstead's Software Science Theory to Algol 68 Programs *Software - Practice and Experience* 14 (3), 1984: 263-276.

65 S.S. Yau and J.S. Collofello, Some Stability Measures for Software Maintenance *IEEE Transactions on Software Engineering* 6 (6), 1980: 545-552.

66 S.H. Zweben, Heads I Win, Tails You Lose (Zweben's Comment) *Computing Surveys* 11 (3), 1979: 277-278.