A Critique of Software Defect Prediction Models

Norman E. Fenton, *Member*, *IEEE Computer Society*, and Martin Neil, *Member*, *IEEE Computer Society*

Abstract—Many organizations want to predict the number of defects (faults) in software systems, before they are deployed, to gauge the likely delivered quality and maintenance effort. To help in this numerous software metrics and statistical models have been developed, with a correspondingly large literature. We provide a *critical* review of this literature and the state-of-the-art. Most of the wide range of prediction models use size and complexity metrics to predict defects. Others are based on testing data, the "quality" of the development process, or take a multivariate approach. The authors of the models have often made heroic contributions to a subject otherwise bereft of empirical studies. However, there are a number of serious theoretical and practical problems in many studies. The models are weak because of their inability to cope with the, as yet, unknown relationship between defects and failures. There are fundamental statistical and data quality problems that undermine model validity. More significantly many prediction models tend to model only part of the underlying problem and seriously misspecify it. To illustrate these points the "Goldilock's Conjecture," that there is an optimum module size, is used to show the considerable problems inherent in current defect prediction approaches. Careful and considered analysis of past and new results shows that the conjecture lacks support and that some models are misleading. We recommend holistic models for software defect prediction, using Bayesian Belief Networks, as alternative approaches to the single-issue models used at present. We also argue for research into a theory of "software decomposition" in order to test hypotheses about defect introduction and help construct a better science of software engineering.

Index Terms—Software faults and failures, defects, complexity metrics, fault-density, Bayesian Belief Networks.

1 Introduction

Organizations are still asking how they can predict the quality of their software *before* it is used despite the substantial research effort spent attempting to find an answer to this question over the last 30 years. There are many papers advocating statistical models and metrics which purport to answer the quality question. Defects, like quality, can be defined in many different ways but are more commonly defined as deviations from specifications or expectations which might lead to failures in operation.

Generally, efforts have tended to concentrate on the following three problem perspectives [1], [2], [3]:

1) predicting the number of defects in the system;
2) estimating the reliability of the system in terms of time to failure;
3) understanding the impact of design and testing processes on defect counts and failure densities.

A wide range of prediction models have been proposed. Complexity and size metrics have been used in an attempt to predict the number of defects a system will reveal in operation or testing. Reliability models have been developed to predict failure rates based on the expected operational usage profile of the system. Information from defect detection and the testing process has been used to predict defects. The maturity of design and testing processes have been advanced as ways of reducing defects. Recently large complex multivariate statistical models have been produced in an attempt to find a single complexity metric that will account for defects.

• *N.E. Fenton and M. Neil are with the Centre for Software Reliability, Northampton Square, London EC1V 0HB, England. E-mail: {n.fenton, martin}@csr.city.ac.uk.*

Manuscript received 3 Sept. 1997; revised 25 Aug. 1998. Recommended for acceptance by R. Hamlet. For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105579.

This paper provides a *critical* review of this literature with the purpose of identifying future avenues of research. We cover complexity and size metrics (Section 2), the testing process (Section 3), the design and development process (Section 4), and recent multivariate studies (Section 5). For a comprehensive discussion of reliability models, see [4]. We uncover a number of theoretical and practical problems in these studies in Section 6, in particular the so-called "Goldilock's Conjecture."

Despite the many efforts to predict defects, there appears to be little consensus on what the constituent elements of the problem really are. In Section 7, we suggest a way to improve the defect prediction situation by describing a prototype, Bayesian Belief Network (BBN) based, model which we feel can at least partly solve the problems identified. Finally, in Section 8 we record our conclusions.

2 Prediction Using Size and Complexity Metrics

Most defect prediction studies are based on size and complexity metrics. The earliest such study appears to have been Akiyama's, [5], which was based on a system developed at Fujitsu, Japan. It is typical of many regression based "data fitting" models which became common place in the literature. The study showed that linear models of some simple metrics provide reasonable estimates for the total number of defects D (the dependent variable) which is actually defined as the sum of the defects found during testing and the defects found during two months after release. Akiyama computed four regression equations.

Akiyama's first Equation (1) predicted defects from lines of code (LOC). From (1) it can be calculated that a 1,000 LOC (i.e., 1 KLOC) module is expected to have approximately 23 defects.

$$D = 4.86 + 0.018L \quad (1)$$

Other equations had the following dependent metrics: number of decisions C; number of subroutine calls J; and a composite metric $C + J$.

Another early study by Ferdinand, [6], argued that the expected number of defects increases with the number n of code segments; a code segment is a sequence of executable statements which, once entered, must all be executed. Specifically the theory asserts that for smaller numbers of segments, the number of defects is proportional to a power of n; for larger numbers of segments, the number of defects increases as a constant to the power n.

Halstead, [7], proposed a number of size metrics, which have been interpreted as "complexity" metrics, and used these as predictors of program defects. Most notably, Halstead asserted that the number of defects D in a program P is predicted by (2):

$$D = \frac{V}{3{,}000} \quad (2)$$

where V is the (language dependent) volume metric (which like all the Halstead metrics is defined in terms of number of unique operators and unique operands in P; for details see [8]). The divisor 3,000 represents the mean number of mental discriminations between decisions made by the programmer. Each such decision possibly results in error and thereby a residual defect. Thus, Halstead's model was, unlike Akiyama's, based on some kind of theory. Interestingly, Halstead himself "validated" (1) using Akiyama's data. Ottenstein, [9], obtained similar results to Halstead.

Lipow, [10] went much further, because he got round the problem of computing V directly in (3), by using lines of executable code L instead. Specifically, he used the Halstead theory to compute a series of equations of the form:

$$\frac{D}{L} = A_0 + A_1 \ln L + A_2 \ln^2 L \quad (3)$$

where each of the A_i are dependent on the average number of usages of operators and operands per LOC for a particular language. For example, for Fortran $A_0 = 0.0047$; $A_1 = 0.0023$; $A_2 = 0.000043$. For an assembly language $A_0 = 0.0012$; $A_1 = 0.0001$; $A_2 = 0.000002$.

Gaffney, [11], argued that the relationship between D and L was not language dependent. He used Lipow's own data to deduce the prediction (4):

$$D = 4.2 + 0.0015(L)^{4/3} \quad (4)$$

An interesting ramification of this was that there was an optimal size for individual modules with respect to defect density. For (4) this optimum module size is 877 LOC. Numerous other researchers have since reported on optimal module sizes. For example, Compton and Withrow of UNISYS derived the following polynomial equation, [12]:

$$D = 0.069 + 0.00156L + 0.00000047(L)^2 \quad (5)$$

Based on (5) and further analysis Compton and Withrow concluded that the optimum size for an Ada module, with respect to minimizing error density is 83 source statements. They dubbed this the "Goldilocks Principle" with the idea that there is an optimum module size that is "not too big nor too small."

The phenomenon that larger modules can have lower defect densities was confirmed in [13], [14], [15]. Basili and Perricone argued that this may be explained by the fact that there are a large number of interface defects distributed evenly across modules. Moller and Paulish suggested that larger modules tend to be developed more carefully; they discovered that modules consisting of greater than 70 lines of code have similar defect densities. For modules of size less than 70 lines of code, the defect density increases significantly.

Similar experiences are reported by [16], [17]. Hatton examined a number of data sets, [15], [18] and concluded that there was evidence of "macroscopic behavior" common to all data sets despite the massive internal complexity of each system studied, [19]. This behavior was likened to "molecules" in a gas and used to conjecture an entropy model for defects which also borrowed from ideas in cognitive psychology. Assuming the short-term memory affects the rate of human error he developed a logarithmic model, made up of two parts, and fitted it to the data sets.[1] The first part modeled the effects of small modules on short-term memory, while the second modeled the effects of large modules. He asserted that, for module sizes above 200-400 lines of code, the human "memory cache" overflows and mistakes are made leading to defects. For systems decomposed into smaller pieces than this cache limit the human memory cache is used inefficiently storing "links" between the modules thus also leading to more defects. He concluded that larger components are proportionally more reliable than smaller components. Clearly this would, if true, cast serious doubt over the theory of program decomposition which is so central to software engineering.

The realization that size-based metrics alone are poor general predictors of defect density spurred on much research into more discriminating complexity metrics. McCabe's cyclomatic complexity, [20], has been used in many studies, but it too is essentially a size measure (being equal to the number of decisions plus one in most programs). Kitchenham et al. [21], examined the relationship between the changes experienced by two subsystems and a number of metrics, including McCabe's metric. Two different regression equations resulted (6), (7):

$$C = 0.042MCI - 0.075N + 0.00001HE \quad (6)$$

$$C = 0.25MCI - 0.53DI + 0.09VG \quad (7)$$

For the first subsystem changes, C, was found to be reasonably dependent on machine code instructions, MCI, operator and operand totals, N, and Halstead's effort metric, HE. For the other subsystem McCabe's complexity metric, VG was found to partially explain C along with machine code instructions, MCI and data items, DI.

All of the metrics discussed so far are defined on code. There are now a large number of metrics available earlier in the life-cycle, most of which have been claimed by their proponents to have some predictive powers with respect

1. There is nothing new here since Halstead [3] was one of the first to apply Miller's finding that people can only effectively recall seven plus or minus two items from their short-term memory. Likewise the construction of a partitioned model contrasting "small" module effects on faults and "large" module effects on faults was done by Compton and Withrow in 1990 [7].

to residual defect density. For example, there have been numerous attempts to define metrics which can be extracted from design documents using counts of "between module complexity" such as call statements and data flows; the most well known are the metrics in [22]. Ohlsson and Alberg, [23], reported on a study at Ericsson where metrics derived automatically from design documents were used to predict especially fault-prone modules prior to testing. Recently, there have been several attempts, such as [24], [25], to define metrics on object-oriented designs.

The advent and widespread use of Albrecht Function Points (FPs) raises the possibility of defect density predictions based on a metric which can be extracted at the specification stage. There is widespread belief that FPs are a better (one-dimensional) size metric than LOC; in theory at least they get round the problems of lack of uniformity and they are also language independent. We already see defect density defined in terms of defects per FP, and empirical studies are emerging that seem likely to be the basis for predictive models. For example, in Table 1, [26] reports the following bench-marking study, reportedly based on large amounts of data from different commercial sources.

TABLE 1
DEFECTS PER LIFE-CYCLE PHASE PREDICTION USING TESTING METRICS

Defect Origins	Defects per Function Point
Requirements	1.00
Design	1.25
Coding	1.75
Documentation	0.60
Bad fixes	0.40
Total	5.00

TABLE 2
DEFECTS FOUND PER TESTING APPROACH

Testing Type	Defects Found/hr
Regular use	0.210
Black box	0.282
White box	0.322
Reading/inspections	1.057

3 PREDICTION USING TESTING METRICS

Some of the most promising local models for predicting residual defects involve very careful collection of data about defects discovered during early inspection and testing phases. The idea is very simple: you have n predefined phases at which you collect data d_n (the defect rate. Suppose phase n represents the period of the first six months of the product in the field, so that d_n is the rate of defects found within that period. To predict d_n at phase $n - 1$ (which might be integration testing) you look at the actual sequence $d_1, ..., d_{n-1}$ and compare this with profiles of similar, previous products, and use statistical extrapolation techniques. With enough data it is possible to get accurate predictions of d_n based on observed $d_1, ..., d_m$ where m is less than $n - 1$. This method is an important feature of the Japanese software factory approach [27], [28], [29]. Extremely accurate predictions are claimed (usually within 95 percent confidence limits) due to stability of the development and testing environment and the extent of data collection. It appears that the IBM NASA Space shuttle team is achieving similarly accurate predictions based on the same kind of approach [18].

In the absence of an extensive local database it may be possible to use published bench-marking data to help with this kind of prediction. Dyer, [30], and Humphrey, [31], contain a lot of this kind of data. Buck and Robbins, [32], report on some remarkably consistent defect density values during different review and testing stages across different types of software projects at IBM. For example, for new code developed the number of defects per KLOC discovered with Fagan inspections settles to a number between 8 and 12. There is no such consistency for old code. Also the number of man-hours spent on the inspection process per major defect is always between three and five. The authors speculate that, despite being unsubstantiated with data, these values form "natural numbers of programming," believing that they are "inherent to the programming process itself." Also useful (providing you are aware of the kind of limitations discussed in [33]) is the kind of data published by [34] in Table 2.

One class of testing metrics that appear to be quite promising for predicting defects are the so called test coverage measures. A structural testing strategy specifies that we have to select enough test cases so that each of a set of "objects" in a program lie on some path (i.e., are "covered") in at least on test case. For example, statement coverage is a structural testing strategy in which the "objects" are the statements. For a given strategy and a given set of test cases we can ask what proportion of coverage has been achieved. The resulting metric is defined as the Test Effectiveness Ratio (TER) with respect to that strategy. For example, TER1 is the TER for statement coverage; TER2 is the TER for branch coverage; and TER3 is the TER for linear code sequence and jump coverage. Clearly we might expect the number of discovered defects to approach the number of defects actually in the program as the values of these TER metrics increases. Veevers and Marshall, [35], report on some defect and reliability prediction models using these metrics which give quite promising results. Interestingly Neil, [36], reported that the modules with high structural complexity metric values had a significantly lower TER than smaller modules. This supports our intuition that testing larger modules is more difficult and that such modules would appear more likely to contain undetected defects.

Voas and Miller use static analysis of programs to conjecture the presence or absence of defects before testing has taken place, [37]. Their method relies on a notion of program testability, which seeks to determine how likely a program will fail *assuming* it contains defects. Some programs will contain defects that may be difficult to discover by testing by virtue of their structure and organization. Such programs have a low defect revealing potential and may, therefore, hide defects until they show themselves as failures during operation. Voas and Miller use program mutation analysis to simulate the conditions that would cause a defect to reveal itself as a failure if a defect was indeed present. Essentially if program testability could be estimated before testing takes place the estimates could help predict those programs that would reveal less defects during testing even if they contained

defects. Bertolino and Strigini, [38], provide an alternative exposition of testability measurement and its relation to testing, debugging, and reliability assessment.

4 Prediction Using Process Quality Data

There are many experts who argue that the "quality" of the development process is the best predictor of product quality (and hence, by default, of residual defect density). This issue, and the problems surrounding it, is discussed extensively in [33]. There is a dearth of empirical evidence linking process quality to product quality. The simplest metric of process quality is the five-level ordinal scale SEI Capability Maturity Model (CMM) ranking. Despite its widespread popularity, there was until recently no evidence to show that level $(n + 1)$ companies generally deliver products with lower residual defect density than level (n) companies. The Diaz and Sligo study, [39], provides the first promising empirical support for this widely held assumption.

Clearly the strict 1–5 ranking, as prescribed by the SEI-CMM, is too coarse to be used directly for defect prediction since not all of the processes covered by the CMM will relate to software quality. The best available evidence relating particular process methods to defect density concerns the Cleanroom method [30]. There is independent validation that, for relatively small projects (less than 30 KLOC), the use of Cleanroom results in approximately three errors per KLOC during statistical testing, compared with traditional development postdelivery defect densities of between five to 10 defects per KLOC. Also, Capers Jones hypothesizes quality targets expressed in "defect potentials" and "delivered defects" for different CMM levels, as shown in Table 3 [40].

TABLE 3
Relationship Between CMM Levels and Delivered Defects Multivariate Approaches

SEI CMM Levels	Defect Potentials	Removal Efficiency (%)	Delivered Defects
1	5	85	0.75
2	4	89	0.44
3	3	91	0.27
4	2	93	0.14
5	1	95	0.05

5 Multivariate Approaches

There have been many attempts to develop multilinear regression models based on multiple metrics. If there is a consensus of sorts about such approaches it is that the accuracy of the predictions is never significantly worse when the metrics set is reduced to a handful (say 3-6 rather than 30), [41]; a major reason for this is that many of the metrics are colinear; that is they capture the same underlying attribute (so the reduced set of metrics has the same information content, [42]). Thus, much work has concentrated on how to select those small number of metrics which are somehow the most powerful and/or representative. Principal Component Analysis (see [43]) is used in some of the studies to reduce the dimensionality of many related metrics to a smaller set of "principal components," while retaining most of the variation observed in the original metrics.

For example, [42] discovered that 38 metrics, collected on around 1,000 modules, could be reduced to six orthogonal dimensions that account for 90 percent of the variability. The most important dimensions; size, nesting, and prime were then used to develop an equation to discriminate between low and high maintainability modules.

Munson and Khoshgoftaar in various papers, [41], [43], [44] use a similar technique, factor analysis, to reduce the dimensionality to a number of "independent" factors. These factors are then labeled so as to represent the "true" underlying dimension being measured, such as control, volume and modularity. In [43] they used factor analytic variables to help fit regression models to a number of error data sets, including Akiyama's [5]. This helped to get over the inherent regression analysis problems presented by multicolinearity in metrics data.

Munson and Khoshgoftaar have advanced the multivariate approach to calculate a "relative complexity metric." This metric is calculated using the magnitude of variability from each of the factor analysis dimensions as the input weights in a weighted sum. In this way a single metric integrates all of the information contained in a large number of metrics. This is seen to offer many advantages of using a univariate decision criterion such as McCabe's metric [44].

6 A Critique of Current Approaches to Defect Prediction

Despite the heroic contributions made by the authors of previous empirical studies, serious flaws remain and have detrimentally influenced our models for defect prediction. Of course, such weaknesses exist in all scientific endeavours but if we are to improve scientific enquiry in software engineering we must first recognize past mistakes before suggesting ways forward.

The key issues affecting the software engineering community's historical research direction, with respect to defect prediction, are:

- the unknown relationship between defects and failures (Section 6.1);
- problems with the "multivariate" statistical approach (Section 6.2);
- problems of using size and complexity metrics as sole "predictors" of defects (Section 6.3);
- problems in statistical methodology and data quality (Section 6.4);
- false claims about software decomposition and the "Goldilock's Conjecture" (Section 6.5).

6.1 The Unknown Relationship between Defects and Failures

There is considerable disagreement about the definitions of defects, errors, faults, and failures. In different studies defect counts refer to:

- postrelease defects;
- the total of "known" defects;
- the set of defects discovered after some arbitrary fixed point in the software life cycle (e.g., after unit testing).

The terminology differs widely between studies; defect rate, defect density, and failure rate are used almost interchangeably. It can also be difficult to tell whether a model is predicting discovered defects or residual defects. Because of these problems (which are discussed extensively in [45]) we have to be extremely careful about the way we interpret published predictive models.

Apart from these problems of terminology and definition the most serious weakness of any prediction of residual defects or defect density concerns the weakness of defect count itself as a measure of software reliability.[2] Even if we knew exactly the number of residual defects in our system we have to be extremely wary about making definitive statements about how the system will operate in practice. The reasons for this appear to be:

- difficulty of determining in advance the seriousness of a defect; few of the empirical studies attempt to distinguish different classes of defects;
- great variability in the way systems are used by different users, resulting in wide variations of operational profiles. It is thus difficult to predict which defects are likely to lead to failures (or to commonly occurring failures).

The latter point is particularly serious and has been highlighted dramatically by [46]. Adams examined data from nine large software products, each with many thousands of years of logged use world wide. He charted the relationship between detected defects and their manifestation as failures. For example, 33 percent of all defects led to failures with a mean time to failure greater than 5,000 years. In practical terms, this means that such defects will almost never manifest themselves as failures. Conversely, the proportion of defects which led to a mean time to failure of less than 50 years was very small (around 2 percent). However, it is these defects which are the important ones to find, since these are the ones which eventually exhibit themselves as failures to a significant number of users. Thus, Adams' data demonstrates the Pareto principle: a very small proportion of the defects in a system will lead to almost all the observed failures in a given period of time; conversely, most defects in a system are benign in the sense that in the same given period of time they will not lead to failures.

It follows that finding (and removing) large numbers of defects may not necessarily lead to improved reliability. It also follows that a very accurate residual defect density prediction may be a very poor predictor of operational reliability, as has been observed in practice [47]. This means we should be very wary of attempts to equate fault densities with failure rates, as proposed for example by Capers Jones (Table 4 [48]). Although highly attractive in principle, such a model does not stand up to empirical validation.

Defect counts cannot be used to predict reliability because, despite its usefulness from a system developer's point of view, it does not measure the quality of the system as the user is likely to experience it. The promotion of defect counts as a measure of "general quality" is, therefore, misleading.

2. Here we use the "technical" concept of reliability, defined as mean time to failure or probability of failure on demand, in contrast to the "looser" concept of reliability with its emphasis on defects.

TABLE 4
DEFECTS DENSITY (F/KLOC) VS. MTTF

F/KLOC	MTTF
> 30	1 min
20–30	4-5 min
5–10	1 hr
2–5	several hours
1–2	24 hr
0.5–1	1 month

Reliability prediction should, therefore, be viewed as complementary to defect density prediction.

6.2 Problems with the Multivariate Approach

Applying multivariate techniques, like factor analysis, produces metrics which cannot be easily or directly interpretable in terms of program features. For example, in [43] a factor dimension metric, *control*, was calculated by the weighted sum (8):

$$control = a_1 HNK + a_2 PRC + a_3 E + a_4 VG + a_5 MMC + a_6 Error + a_7 HNP + a_8 LOC \quad (8)$$

where the a_i s are derived from factor analysis. *HNK* was Henry and Kafura's information flow complexity metric, *PRC* is a count of the number of procedures, E is Halstead's effort metric, *VG* is McCabe's complexity metric, *MMC* is Harrison's complexity metric, and *LOC* is lines of code. Although this equation might help to avoid multicolinearity it is hard to see how you might advise a programmer or designer on how to redesign the programs to achieve a "better" *control* metric value for a given module. Likewise the effects of such a change in module *control* on defects is less than clear.

These problems are compounded in the search for an ultimate or relative complexity metric [43]. The simplicity of such a single number seems deceptively appealing but the principles of measurement are based on identifying differing well-defined attributes with single standard measures [45]. Although there is a clear role for data reduction and analysis techniques, such as factor analysis, this should not be confused or used instead of measurement theory. For example, statement count and lines of code are highly correlated because programs with more lines of code typically have a higher number of statements. This does not mean that the *true* size of programs is some combination of the two metrics. A more suitable explanation would be that both are alternative measures of the same attribute. After all centigrade and fahrenheit are highly correlated measures of temperature. Meteorologists have agreed a convention to use one of these as a standard in weather forecasts. In the United States temperature is most often quoted as fahrenheit, while in the United Kingdom it is quoted as centigrade. They do not take a weighted sum of both temperature measures. This point lends support to the need to define meaningful and standard measures for specific attributes rather than searching for a single metric using the multivariate approach.

6.3 Problems in Using Size and Complexity Metrics to Predict Defects

A discussion of the theoretical and empirical problems with many of the individual metrics discussed above may be

found in [45]. There are as many empirical studies (see, for example, [49], [50], [51]) refuting the models based on Halstead, and McCabe as there are studies "validating" them. Moreover, some of the latter are seriously flawed. Here we concentrate entirely on their use within models used to predict defects.

The majority of size and complexity models assume a straightforward relationship with defects—defects are a function of size or defects are caused by program complexity. Despite the reported high correlations between design complexity and defects the relationship is clearly not a straightforward one. It is clear that it is not entirely causal because if it were we couldn't explain the presence of defects introduced when the requirements are defined. It is wrong to mistake correlation for causation. An analogy would be the significant positive correlation between IQ and height in children. It would be dangerous to predict IQ from height because height doesn't *cause* high IQ; the underlying causal factor is physical and mental maturation. There are a number of interesting observations about the way complexity metrics are used to predict defect counts:

- the models ignore the causal effects of programmers and designers. After all it is they who introduce the defects so any attribution for faulty code must finally rest with individual(s);
- overly complex programs are themselves a consequence of poor design ability or problem difficulty. Difficult problems might demand complex solutions and novice programmers might produce "spaghetti code";
- defects may be introduced at the design stage because of the overcomplexity of the designs already produced. Clerical errors and mistakes will be committed because the existing design is difficult to comprehend. Defects of this type are "inconsistencies" between design modules and can be thought of as quite distinct from requirements defects.

6.4 Problems in Data Quality and Statistical Methodology

The weight given to knowledge obtained by empirical means rests on the quality of the data collected and the degree of rigor employed in analyzing this data. Problems in either data quality or analysis may be enough to make the resulting conclusions invalid. Unfortunately some defect prediction studies have suffered from such problems. These problems are caused, in the main, by a lack of attention to the assumptions necessary for successful use of a particular statistical technique. Other serious problems include the lack of distinction made between model fitting and model prediction and the unjustified removal of data points or misuse of averaged data.

The ability to replicate results is a key component of any empirical discipline. In software development different findings from diverse experiments could be explained by the fact that different, perhaps uncontrolled, processes were used on different projects. Comparability over case studies might be better achieved if the processes used during development were documented, along with estimates of the extent to which they were actually followed.

6.4.1 Multicolinearity

Multicolinearity is the most common methodological problem encountered in the literature. Multicolinearity is present when a number of predictor variables are highly positively or negatively correlated. Linear regression depends on the assumption of zero correlation between predictor variables, [52]. The consequences of multicolinearity are many fold; it causes unstable coefficients, misleading statistical tests, and unexpected coefficient signs. For example, one of the equations in [21] (9):

$$C = 0.042\,MCI - 0.075\,N + 0.00001\,HE \qquad (9)$$

shows clear signs of multicolinearity. If we examine the equation coefficients we can see that an increase in the operator and operand total, N, should result in an increase in changes, c, all things being equal. This is clearly counterintuitive. In fact analysis of the data reveals that machine code instructions, *MCI*, operand, and operator count, N, and Halstead's Effort metric, *HE*, are all highly correlated [42]. This type of problem appears to be common in the software metrics literature and some recent studies appear to have fallen victim to the multicolinearity problem [12], [53].

Colinearity between variables has also been detected in a number of studies that reported a negative correlation between defect density and module size. Rosenberg reports that, since there must be a negative correlation between X, size, and 1/X it follows that the correlation between X and Y/X (defects/size) must be negative whenever defects are growing at most linearly with size [54]. Studies which have postulated such a linear relationship are more than likely to have detected negative correlation, and therefore concluded that large modules have smaller defect densities, because of this property of arithmetic.

6.4.2 Factor Analysis vs. Principal Components Analysis

The use of factor analysis and principal components analysis solves the multicolinearity problem by creating new orthogonal factors or principal component dimensions, [43]. Unfortunately the application of factor analysis assumes the errors are Gaussian, whereas [55] notes that most software metrics data is non-Gaussian. Principal components analysis can be used instead of factor analysis because it does not rely on any distributional assumptions, but will on many occasions produce results broadly in agreement with factor analysis. This makes the distinction a minor one, but one that needs to be considered.

6.4.3 Fitting Models vs. Predicting Data

Regression modeling approaches are typically concerned with fitting models to data rather than predicting data. Regression analysis typically finds the least-squares fit to the data and the goodness of this fit demonstrates how well the model explains *historical* data. However a truly successful model is one which can predict the number of defects discovered in an *unknown* module. Furthermore, this must be a module not used in the derivation of the model. Unfortunately, perhaps because of the shortage of data, some researchers have tended to use their data to fit the model without being able to test the resultant model out on a new data set. See, for example, [5], [12], [16].

6.4.4 Removing Data Points

In standard statistical practice there should normally be strong theoretical or practical justification for removing data points during analysis. Recording and transcription errors are often an acceptable reason. Unfortunately, it is often difficult to tell from published papers whether any data points have been removed before analysis, and if they have, the reasons why. One notable case is Compton and Withrow, [12], who reported removing a large number of data points from the analysis because they represented modules that had experienced zero defects. Such action is surprising in view of the conjecture they wished to test; that defects were minimised around an optimum size for Ada. If the majority of smaller modules had zero defects, as it appears, then we cannot accept Compton and Withrow's conclusions about the "Goldilock's Conjecture."

6.4.5 Using "Averaged" Data

We believe that the use of "averaged" data in analysis rather than the original data prejudices many studies. The study in [19] uses graphs, apparently derived from the original NASA-Goddard data, plotting "average size in statements" against "number of defects" or "defect density." Analysis of averages are one step removed from the original data and it raises a number of issues. Using averages reduces the amount of information available to test the conjecture under study and any conclusions will be correspondingly weaker. The classic study in [13] used average fault density of grouped data in a way that suggested a trend that was not supported by the raw data. The use of averages may be a practical way around the common problem where defect data is collected at a higher level, perhaps at the system or subsystem level, than is ideal; defects recorded against individual modules or procedures. As a consequence data analysis must match defect data on systems against statement counts automatically collected at the module level. There may be some modules within a subsystem that are over penalized when others keep the average high because the other modules in that subsystem have more defects or vice versa. Thus, we cannot completely trust any defect data collected in this way.

Misuse of averages has occurred in one other form. In Gaffney's paper, [11], the rule for optimal module size was derived on the assumption that to calculate the total number of defects in a system we could use the same model as had been derived using module defect counts. The model derived at the module level is shown by (4) and can be extended to count the total defects in a system, D_T, based on L_i, (9). The total number of modules in the system is denoted by N.

$$D_T = \sum_{i=1}^{N} D_i = 4.2N + 0.0015 \sum_{i=1}^{N} (L_i)^{4/3} \quad (9)$$

Gaffney assumes that the average module size can be used to calculate the total defect count and also the optimum module size for any system, using (10):

$$D_T = 4.2N + 0.0015N \left[\frac{\sum_{i=1}^{N} L_i}{N} \right]^{4/3} \quad (10)$$

However we can see that (9) and (10) are not equivalent. The use of (10) mistakenly assumes the power of a sum is equal to a sum of powers.

6.5 The "Goldilock's Conjecture"

The results of inaccurate modeling and inference is perhaps most evident in the debate that surrounds the "Goldilock's Conjecture" discussed in Section 2—the idea that there is an optimum module size that is "not too big nor too small." Hatton, [19], claims that there is

> "*compelling empirical evidence from disparate sources to suggest that in any software system, larger components are proportionally more reliable than smaller components*."

If these results were generally true the implications for software engineering would be very serious indeed. It would mean that program decomposition as a way of solving problems simply did not work. Virtually all of the work done in software engineering extending from fundamental concepts, like modularity and information-hiding, to methods, like object-oriented and structured design would be suspect because all of them rely on some notion of decomposition. If decomposition doesn't work then there would be no good reason for doing it.

Claims with such serious consequences as these deserve special attention. We must ask whether the data and knowledge exists to support them. These are clear criteria —if the data exist to refute the conjecture that large modules are "better" and if we have a sensible explanation for this result then a claim will stand. Our analysis shows that, using these criteria, these claims cannot currently stand. In the studies that support the conjecture we found the following problems:

- none define "module" in such a way as to make comparison across data sets possible;
- none explicitly compare different approaches to structuring and decomposing designs;
- the data analysis or quality of the data used could not support the results claimed;
- a number of factors exist that could partly explain the results which these studies have neglected to examine.

Additionally, there are other data sets which do not show any clear relationships between module size and defect density.

If we examine the various results we can divide them into three main classes. The first class contains models, exemplified by graph Fig. 1a, that shows how defect density falls as module size increases. Models such as these have been produced by Akiyama, Gaffney, and Basili and Pericone. The second class of models, exemplified by Fig. 1b, differ from the first because they show the Goldilock's principle at work. Here defect density rises as modules get bigger in size. The third class, exemplified by Fig. 1c, shows no discernible pattern whatsoever. Here the relationship between defect density and module size appears random (no meaningful curvilinear models could be fitted to the data at all.

The third class of results show the typical data pattern from a number of very large industrial systems. One data set was collected at the Tandem Corporation and was reported in, [56]. The Tandem data was subsequently analyzed by Neil [42], using the principal components technique to produce a

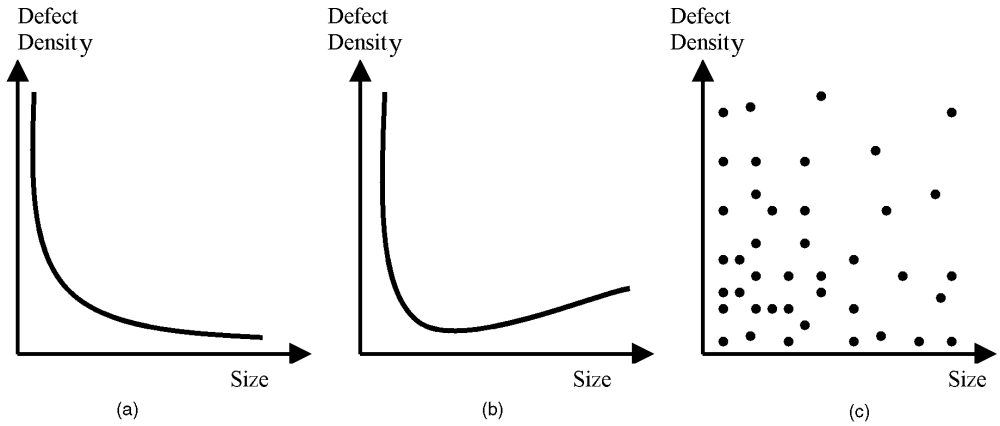

Fig. 1. Three classes of defect density results. (a) Akiyama (1971), Basili and Perricome (1984), and Gaffney (1984); (b) Moeller and Paulish (1993), Compton and Withrow (1990), and Hatton (1997); (c) Neil (1992) and Fenton and Ohlsson (1997).

"combined measure" of different size measures, such as decision counts. This principal component statistic was then plotted against the number of changes made to the system modules (these were predominantly changes made to fix defects). This defect data was standardized according to normal statistical practice. A polynomial regression curve was fitted to the data in order to determine whether there was significant nonlinear effects of size on defect density. The results were published and are reproduced here in Fig. 2.

Despite some parameters of the polynomial curve being statistically significant it is obvious that there is no discernible relationship between defect counts and module size in the Tandem data set. Many small modules experienced no defects at all and the fitted polynomial curve would be useless for prediction. This data clearly refutes the simplistic assumptions typified by class Fig. 1a and 1b models (these models couldn't explain the Tandem data) nor accurately predict the defect density values of these Tandem modules. A similar analysis and result is presented in [47].

We conclude that the relationship between defects and module size is too complex, in general, to admit to straightforward curve fitting models. These results, therefore, contradict the idea that there is a general law linking defect density and software component size as suggested by the "Goldilock's Conjecture."

7 Predicting Defects Using BBNs

It follows from our analysis in Section 6 that the suggestion that defects can be predicted by complexity or size measures alone presents only a skewed picture. The number of defects discovered is clearly related to the amount of testing performed, as discussed above. A program which has never been tested, or used for that matter, will have a zero defect count, even though its complexity may be very high. Moreover, we can assume the test effectiveness of complex programs is relatively low, [37], and such programs could be expected to exhibit a lower number of defects per line of code during testing because they "hide" defects more effectively. This could explain many of the empirical results that larger modules have lower defect densities. Therefore, from what we know of testability, we could conclude that large modules contained many residual defects, rather than concluding that large modules were more reliable (and by implication that software decomposition is wrong).

Clearly all of the problems described in Section 6 are not going to be solved easily. However, we believe that modeling the complexities of software development using new probabilistic techniques presents a positive way forward. These methods, called Bayesian Belief Networks (BBNs), allow us to express complex interrelations within the model

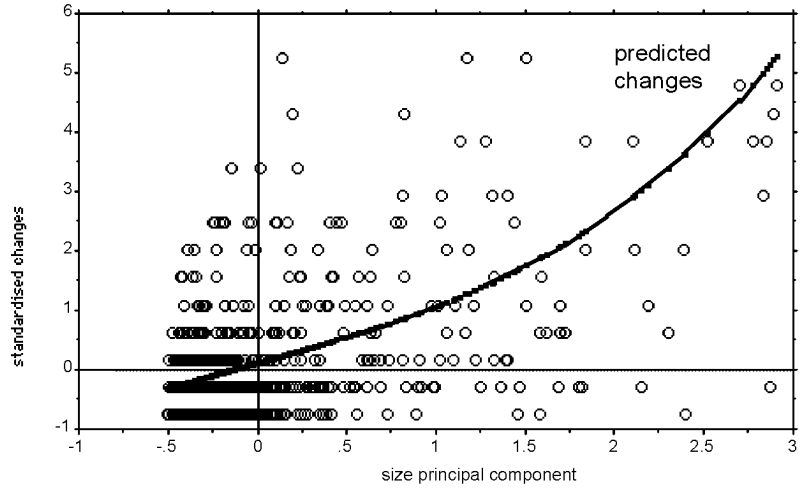

Fig. 2. Tandem data defects counts vs. size "principal component."

at a level of uncertainty commensurate with the problem. In this section, we first provide an overview of BBNs (Section 7.1) and describe the motivation for the particular BBN example used in defects prediction (Section 7.2). In Section 7.3, we describe the actual BBN.

7.1 An Overview of BBNs

Bayesian Belief Networks (also known as Belief Networks, Causal Probabilistic Networks, Causal Nets, Graphical Probability Networks, Probabilistic Cause-Effect Models, and Probabilistic Influence Diagrams) have attracted much recent attention as a possible solution for the problems of decision support under uncertainty. Although the underlying theory (Bayesian probability) has been around for a long time, the possibility of building and executing realistic models has only been made possible because of recent algorithms and software tools that implement them [57]. To date BBNs have proven useful in practical applications such as medical diagnosis and diagnosis of mechanical failures. Their most celebrated recent use has been by Microsoft where BBNs underlie the help wizards in Microsoft Office; also the "intelligent" printer fault diagnostic system which you can run when you log onto Microsoft's web site is in fact a BBN which, as a result of the problem symptoms you enter, identifies the most likely fault.

A BBN is a graphical network that represents probabilistic relationships among variables. BBNs enable reasoning under uncertainty and combine the advantages of an intuitive visual representation with a sound mathematical basis in Bayesian probability. With BBNs, it is possible to articulate expert beliefs about the dependencies between different variables and to propagate consistently the impact of evidence on the probabilities of uncertain outcomes, such as "future system reliability." BBNs allow an injection of scientific rigor when the probability distributions associated with individual nodes are simply "expert opinions."

A BBN is a special type of diagram (called a graph) together with an associated set of probability tables. The graph is made up of nodes and arcs where the nodes represent uncertain variables and the arcs the causal/relevance relationships between the variables. Fig. 3 shows a BBN for an example "reliability prediction" problem. The nodes represent discrete or continuous variables, for example, the node "use of IEC 1508" (the standard) is discrete having two values "yes" and "no," whereas the node "reliability" might be continuous (such as the probability of failure). The arcs represent causal/influential relationships between variables. For example, software reliability is defined by the number of (latent) faults and the operational usage (frequency with which faults may be triggered). Hence, we model this relationship by drawing arcs from the nodes "number of latent faults and "operational usage" to "reliability."

For the node "reliability" the node probability table (NPT) might, therefore, look like that shown in Table 5 (for ultra-simplicity we have made all nodes discrete so that here reliability takes on just three discrete values low, medium, and high). The NPTs capture the conditional probabilities of a node given the state of its parent nodes. For nodes without parents (such as "use of IEC 1508" in Fig. 3.) the NPTs are simply the marginal probabilities.

There may be several ways of determining the probabilities for the NPTs. One of the benefits of BBNs stems from the fact that we are able to accommodate both subjective probabilities (elicited from domain experts) and probabilities based on objective data. Recent tool developments, notably on the SERENE project [58], mean that it is now possible to build very large BBNs with very large probability tables (including continuous node variables). In three separate industrial applications we have built BBNs with several hundred nodes and several millions of probability values [59].

There are many advantages of using BBNs, the most important being the ability to represent and manipulate complex models that might never be implemented using conventional methods. Another advantage is that the model can predict events based on partial or uncertain data. Because BBNs have a rigorous, mathematical meaning there are software tools that can interpret them and perform the complex calculations needed in their use [58].

The benefits of using BBNs include:

- specification of complex relationships using conditional probability statements;
- use of "what-if? analysis and forecasting of effects of process changes;
- easier understanding of chains of complex and seemingly contradictory reasoning via the graphical format;
- explicit modeling of "ignorance" and uncertainty in estimates;
- use of subjectively or objectively derived probability distributions;
- forecasting with missing data.

7.2 Motivation for BBN Approach

Clearly defects are not directly caused by program complexity alone. In reality the propensity to introduce defects will be influenced by many factors unrelated to code or design complexity. There are a number of causal factors at play when we want to explain the presence of defects in a program:

- Difficulty of the problem
- Complexity of designed solution
- Programmer/analyst skill
- Design methods and procedures used

Eliciting requirements is a notoriously difficult process and is widely recognized as being error prone. Defects introduced at the requirements stage are claimed to be the most expensive to remedy if they are not discovered early enough. *Difficulty* depends on the individual trying to understand and describe the nature of the problem as well as the problem itself. A "sorting" problem may appear difficult to a novice programmer but not to an expert. It also seems that the difficulty of the problem is partly influenced by the number of failed attempts at solutions there have been and whether a "ready made" solution can be reused. Thus, novel problems have the highest potential to be difficult and "known" problems tend to be simple because known solutions can be identified and reused. Any software development project will have a mix of "simple" and "difficult" problems depending on what intellectual resources are available to tackle them. Good managers know this and attempt to prevent defects by pairing up people and problems; easier problems to novices and difficult problems to experts.

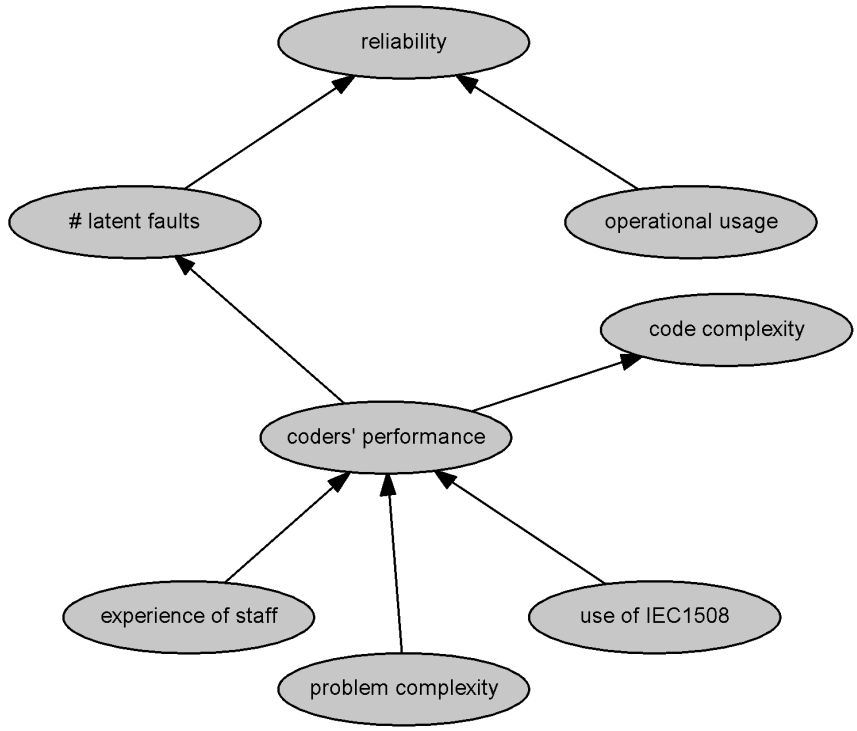

Fig. 3. "Reliability prediction" BBN example.

TABLE 5
NODE PROBABILITY TABLE (NPT) FOR THE NODE "RELIABILITY"

operational usage		*low*			*med*				*high*	
faults		*low*	*med*	*high*	*low*	*med*	*high*	*low*	*med*	*high*
reliability	*low*	0.10	0.20	0.33	0.20	0.33	0.50	0.20	0.33	0.70
	med	0.20	0.30	0.33	0.30	0.33	0.30	0.30	0.33	0.20
	high	0.70	0.50	0.33	0.50	0.33	0.20	0.50	0.33	0.10

When assessing a defect it is useful to determine when it was introduced. Broadly speaking there are two types of defect; those that are introduced in the requirements and those introduced during design (including coding/ implementation which can be treated as design). Useful defect models need to explain why a module has a high or low defect count if we are to learn from its use, otherwise we could never intervene and improve matters. Models using size and complexity metrics are structurally limited to assuming that defects are solely caused by the internal organization of the software design. They cannot explain defects introduced because:

- the "problem" is "hard";
- problem descriptions are inconsistent;
- the wrong "solution" is chosen and does not fulfill the requirements.

We have long recognized in software engineering that program quality can be *potentially* improved through the use of proper project procedures and good design methods. Basic project procedures like configuration management, incident logging, documentation and standards should help reduce the likelihood of defects. Such practices may not help the unique genius you need to work on the really difficult problems but they should raise the standards of the mediocre.

Central to software design method is the notion that problems and designs can be decomposed into meaningful chunks where each can be readily understood alone and finally recomposed to form the final system. Loose coupling between design components is supposed to help ensure that defects are localized and that consistency is maintained. What we have lacked as a community is a theory of program composition and decomposition, instead we have fairly ill-defined ideas on coupling, modularity and cohesiveness. However, despite not having such a theory every day experience tells us that these ideas help reduce defects and improve comprehension. It is indeed hard to think of any other scientific or engineering discipline that has not benefited from this approach.

Surprisingly, much of the defect prediction work has been pursued without reference to testing or testability. According to [37], [38] the testability of a program will dictate its propensity to reveal failures under test conditions and use. Also, at a superficial level the amount of testing performed will determine how many defects will be discovered, assuming there are defects there to discover. Clearly, if no testing is done then no defects will be found. By extension we might argue that difficult problems, with complex solutions, might be difficult to test and so might demand more test effort. If such testing effort is not forthcoming (as is typical in many commercial projects when deadlines

loom) then less defects will be discovered, thus giving an over estimate of the quality achieved and a false sense of security. Thus, any model to predict defects must include testing and testability as crucial factors.

7.3 A Prototype BBN

While there is insufficient space here to fully describe the development and execution of a BBN model here we have developed a prototype BBN to show the potential of BBNs and illustrate their useful properties. This prototype does not exhaustively model all of the issues described in Section 7.2 nor does it solve all of the problems described in Section 6. Rather, it shows the possibility of combining the different software engineering schools of thought on defect prediction into a single model. With this model we should be able to show how predictions might be made and explain historical results more clearly.

The majority of the nodes have the following states: "very-high," "high," "medium," "low," "very low," except for the design size node and defect count nodes which have integer values or ranges and the defect density nodes which have real values. The probabilities attached to each of these states are fictitious but are determined from an analysis of the literature or common-sense assumptions about the direction and strength of relations between variables.

The defect prediction BBN can be explained in two stages. The first stage covers the life-cycle processes of specification, design or coding and the second stage covers testing. In Fig. 4 *problem complexity* represents the degree of complexity inherent in the set of problems to be solved by development. We can think of these problems as being discrete functional requirements in the specification. Solving these problems accrues benefits to the user. Any mismatch between the problem complexity and design effort is likely to cause the introduction of defects, *defects introduced*, and a greater *design size*. Hence the arrows between *design effort*, *problem complexity*, *introduced defects*, and *design size*. The testing stage follows the design stage and in practice the testing effort actually allocated may be much less than that required. The mismatch between testing effort and *design size* will influence the number of *defects detected*, which is bounded by the number of *defects introduced*. The difference between the defects detected and defects introduced is the *residual defects count*. The *defect density at testing* is a function of the design size and defects detected (defects / size). Similarly, the *residual defect density* is *residual defects* divided by *design size*.

Fig. 5 shows the execution of the defect density BBN model under the "Goldilock's Conjecture" using the Hugin Explorer tool [58]. Each of the nodes is shown as a window with a histogram of the predictions made based on the facts entered (facts are represented by histogram bars with 100 percent probability). The scenario runs as follows. A very complex problem is represented as a fact set at "very high" and a "high" amount of *design effort* is allocated, rather than "very high" commensurate with the *problem complexity*. The *design size* is between 1.0–2.0 KLOC. The model then propagates these "facts" and predicts the *introduced defects*, *detected defects* and the *defect density* statistics. The distribution for *defects introduced* peaks at two with 33 percent probability but, because less testing effort was allocated than required, the distribution of *defects detected* peaks around zero with probability 62 percent. The distribution for *defect density at testing* contrasts sharply with the *residual defect density* distribution in that the defect density at testing appears very favourable. This is of course misleading because the residual defect density distribution shows a much higher probability of higher defect density levels.

From the model we can see a credible explanation for observing large "modules" with lower defect densities. Underallocation of design effort for complex problems results in more introduced defects and higher design size. Higher design size requires more testing effort, which if unavailable, leads to less defects being discovered than are actually there. Dividing the small detected defect counts with large design size values will result in small defect densities at the testing stage. The model explains the "Goldilock's Conjecture" without ad hoc explanation.

Clearly the ability to use BBNs to predict defects will depend largely on the stability and maturity of the development processes. Organizations that do not collect metrics data, do not follow defined life-cycles or do not perform any forms of systematic testing will find it hard to build or apply such models. This does not mean to say that less mature organizations cannot build reliable software, rather it implies that they cannot do so predictably and controllably. Achieving predictability of output, for any process, demands a degree of stability rare in software development organizations. Similarly, replication of experimental results can only be predicated on software processes that are defined and repeatable. This clearly implies some notion of Statistical Process Control (SPC) for software development.

8 Conclusions

Much of the published empirical work in the defect prediction area is well in advance of the unfounded rhetoric sadly typical of much of what passes for software engineering research. However every discipline must learn as much, if not more, from its failures as its successes. In this spirit we have reviewed the literature critically with a view to better understand past failures and outline possible avenues for future success.

Our critical review of state-of-the-art of models for predicting software defects has shown that many methodological and theoretical mistakes have been made. Many past studies have suffered from a variety of flaws ranging from model misspecification to use of inappropriate data. The issues and problems surrounding the "Goldilock's Conjecture" illustrate how difficult defect prediction is and how easy it is to commit serious modeling mistakes. Specifically, we conclude that the existing models are incapable of predicting defects accurately using size and complexity metrics alone. Furthermore, these models offer no coherent explanation of how defect introduction and detection variables affect defect counts. Likewise any conclusions that large modules are more reliable and that software decomposition doesn't work are premature.

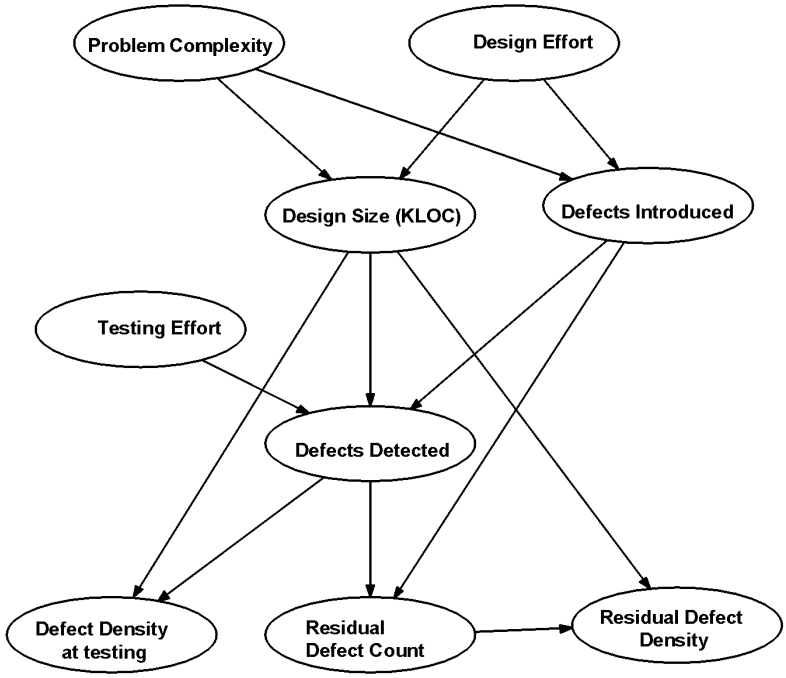

Fig. 4. BBN topology for defect prediction.

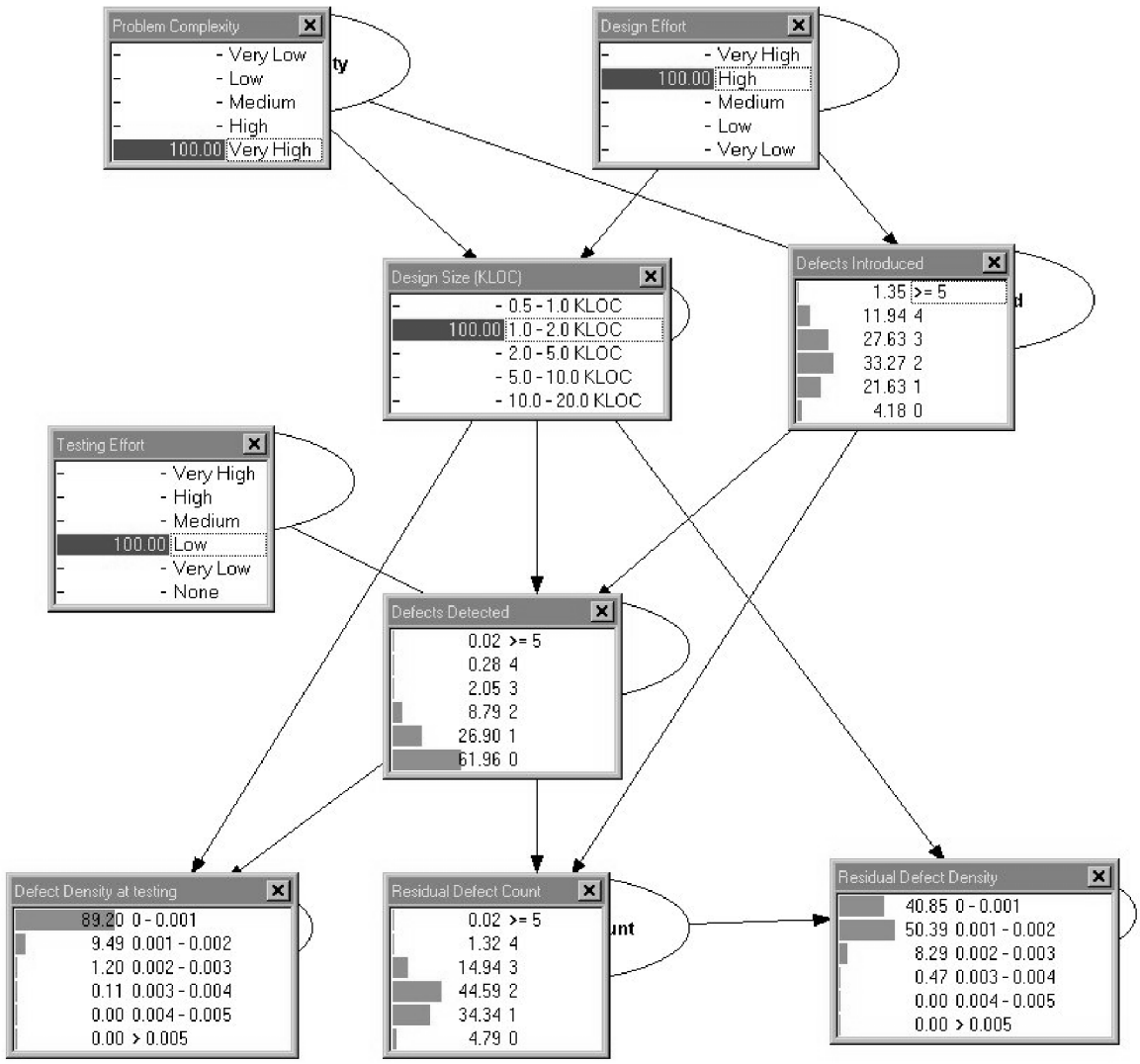

Fig. 5. A demonstration of the “Goldilock’s Conjecture.”

Each of the different "schools of thought" have their own view of the prediction problem despite the interactions and subtle overlaps between process and product identified here. Furthermore each of these views model a *part* of the problem rather than the whole. Perhaps the most critical issue in any scientific endeavor is agreement on the constituent elements or variables of the problem under study. Models are developed to represent the salient features of the problem in a systemic fashion. This is as much the case in physical sciences as social sciences. Economists could not predict the behavior of an economy without an integrated, complex, macroeconomic model of all of the known, pertinent variables. Excluding key variables such as *savings rate* or *productivity* would make the whole exercise invalid. By taking the wider view we can construct a more accurate picture and explain supposedly puzzling and contradictory results. Our analysis of the studies surrounding the "Goldilock's Conjecture" shows how empirical results about defect density can make sense if we look for alternative explanations.

Collecting data from case studies and subjecting it to isolated analysis is not enough because statistics on its own does not provide scientific explanations. We need compelling and sophisticated theories that have the power to explain the empirical observations. The isolated pursuit of these single issue perspectives on the quality prediction problem are, in the longer-term, fruitless. Part of the solution to many of the difficulties presented above is to develop prediction models that *unify* the key elements from the diverse software quality prediction models. We need models that predict software quality by taking into account information from the development process, problem complexity, defect detection processe, and design complexity. We must understand the cause and effect relations between important variables in order to explain why certain design processes are more successful than others in terms of the products they produce.

It seems that successful engineers already operate in a way that tacitly acknowledges these cause-effect relations. After all if they didn't how else could they control and deliver quality products? Project managers make decisions about software quality using best guesses; it seems to us that will always be the case and the best that researchers can do is

1) recognize this fact and
2) improve the "guessing" process.

We, therefore, need to model the subjectivity and uncertainty that is pervasive in software development. Likewise, the challenge for researchers is in transforming this uncertain knowledge, which is already evident in elements of the various quality models already discussed, into a prediction model that other engineers can learn from and apply. We are already working on a number of projects using Bayesian Belief Networks as a method for creating more sophisticated models for prediction, [59], [60], [61], and have described one of the prototype BBNs to outline the approach. Ultimately, this research is aiming to produce a method for the statistical process control (SPC) of software production implied by the SEI's Capability Maturity Model.

All of the defect prediction models reviewed in this paper operate without the use of any formal theory of program / problem decomposition. The literature is however replete with acknowledgments to cognitive explanations of shortcomings in human information processing. While providing useful explanations of why designers employ decomposition as a design tactic they do not, and perhaps cannot, allow us to determine objectively the optimum level of decomposition within a system (be it a requiremen's specification or a program). The literature recognizes the two structural[3] aspects of software, "within" component structural complexity and "between" component structural complexity, but we lack the way to crucially *integrate* these two views in a way that would allow us to say whether one design was more or less structurally complex than another. Such a theory might also allow us to compare different decompositions of the same solution to the same problem requirement, thus *explaining* why different approaches to problem or design decomposition might have caused a designer to commit more or less defects. As things currently stand without such a theory we cannot compare different decompositions and, therefore, cannot carry out experiments comparing different decomposition tactics. This leaves a gap in any evolving science of software engineering that cannot be bridged using current case study based approaches, despite their empirical flavor.

ACKNOWLEDGMENTS

The work carried out here was partially funded by the ESPRIT projects SERENE and DeVa, the EPSRC project IMPRESS, and the DISPO project funded by Scottish Nuclear. The authors are indebted to Niclas Ohlsson and Peter Popov for comments that influenced this work and also to the anonymous reviewers for their helpful and incisive contributions.

REFERENCES

[1] N.E. Schneidewind and H. Hoffmann, "An Experiment in Software Error Data Collection and Analysis," *IEEE Trans. Software Eng.*, vol. 5, no. 3, May 1979.
[2] D. Potier, J.L. Albin, R. Ferreol, A, and Bilodeau, "Experiments with Computer Software Complexity and Reliability," *Proc. Sixth Int'l Conf. Software Eng.*, pp. 94-103, 1982.
[3] T. Nakajo, and H. Kume, "A Case History Analysis of Software Error Cause-Effect Relationships," *IEEE Trans. Software Eng.*, vol. 17, no. 8, Aug. 1991.
[4] S. Brocklehurst and B. Littlewood, "New Ways to Get Accurate Software Reliability Modelling," *IEEE Software*, vol. 34, no. 42, July 1992.
[5] F. Akiyama, "An Example of Software System Debugging," *Information Processing*, vol. 71, pp. 353-379, 1971.
[6] A.E. Ferdinand, "A Theory of System Complexity," *Int'l J. General Syst*ems, vol. 1, pp. 19-33, 1974.
[7] M.H. Halstead, *Elements of Software Science*. Elsevier, North-Holland, 1975.
[8] N.E. Fenton and B.A. Kitchenham, "Validating Software Measures," *J. Software Testing, Verification & Reliability*, vol. 1, no. 2, pp. 27-42, 1991.

3. We are careful here to use the term structural complexity when discussing attributes of design artifacts and cognitive complexity when referring to an individuals understanding of such an artifact. Suffice it to say, that structural complexity would influence cognitive complexity.

[9] L.M. Ottenstein, "Quantitative Estimates of Debugging Requirements," *IEEE Trans. Software Eng.*, vol. 5, no. 5, pp. 504-514, 1979.
[10] M. Lipow, "Number of Faults per Line of Code," *IEEE Trans. Software Eng.*, vol. 8, no. 4, pp. 437-439, 1982.
[11] J.R. Gaffney, "Estimating the Number of Faults in Code," *IEEE Trans. Software Eng.*, vol. 10, no. 4, 1984.
[12] T. Compton, and C. Withrow, "Prediction and Control of Ada Software Defects," *J. Systems and Software*, vol. 12, pp. 199-207, 1990.
[13] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, vol. 27, no. 1, pp. 42-52, 1984.
[14] V. Y. Shen, T. Yu, S.M., Thebaut, and L.R. Paulsen, "Identifying Error-Prone Software—An Empirical Study," *IEEE Trans. Software Eng.*, vol. 11, no. 4, pp. 317-323, 1985.
[15] K.H. Moeller and D. Paulish, "An Empirical Investigation of Software Fault Distribution," *Proc. First Int'l Software Metrics Symp*, pp. 82-90, IEEE CS Press, 1993.
[16] L. Hatton, "The Automation of Software Process and Product Quality," M. Ross, C.A. Brebbia, G. Staples, and J. Stapleton, eds., *Software Quality Management*, pp. 727-744, Southampton: Computation Mechanics Publications, Elsevier, 1993.
[17] L. Hatton, *C and Safety Related Software Development: Standards, Subsets, testing, Metrics, Legal Issues*. McGraw-Hill, 1994.
[18] T. Keller, "Measurements Role in Providing Error-Free Onboard Shuttle Software," *Proc. Third Int'l Applications of Software Metrics Conf.* La Jolla, Calif., pp. 2.154-2.166, 1992. Proc. available from Software Quality Engineering.
[19] L. Hatton, "Re-examining the Fault Density-Component Size Connection," *IEEE Software*, vol. 14, no. 2, pp. 89-98, Mar./Apr. 1997.
[20] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308 - 320, 1976.
[21] B.A. Kitchenham, L.M. Pickard, and S.J. Linkman, "An Evaluation of Some Design Metrics," *Software Eng J.*, vol. 5, no. 1, pp. 50-58, 1990.
[22] S. Henry and D. Kafura, "The Evaluation of Software System's Structure Using Quantitative Software Metrics," *Software—Practice and Experience*, vol. 14, no. 6, pp. 561-573, June 1984.
[23] N. Ohlsson and H. Alberg "Predicting Error-Prone Software Modules in Telephone Switches, *IEEE Trans. Software Eng.*, vol. 22, no. 12, pp. 886-894, 1996.
[24] V. Basili, L. Briand, and W.L. Melo, "A Validation of Object Oriented Design Metrics as Quality Indicators," *IEEE Trans. Software Eng.*, 1996.
[25] S.R. Chidamber and C.F. and Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-498, 1994.
[26] C. Jones, *Applied Software Measurement*. McGraw-Hill, 1991.
[27] M.A. Cusumano, *Japan's Software Factories*. Oxford Univ. Press, 1991.
[28] K. Koga, "Software Reliability Design Method in Hitachi," *Proc. Third European Conf. Software Quality*, Madrid, 1992.
[29] K. Yasuda, "Software Quality Assurance Activities in Japan," *Japanese Perspectives in Software Eng.*, pp. 187-205, Addison-Wesley, 1989.
[30] M. Dyer, *The Cleanroom Approach to Quality Software Development*. Wiley, 1992.
[31] W.S. Humphrey, *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989.
[32] R.D. Buck and J.H. Robbins, "Application of Software Inspection Methodology in Design and Code," *Software Validation*, H.-L. Hausen, ed., pp. 41-56, Elsevier Science, 1984.
[33] N.E. Fenton, S. Lawrence Pfleeger, and R. Glass, "Science and Substance: A Challenge to Software Engineers," *IEEE Software*, pp. 86-95, July 1994.
[34] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992.
[35] A. Veevers and A.C. Marshall, "A Relationship between Software Coverage Metrics and Reliability," *J Software Testing, Verification and Reliability*, vol. 4, pp. 3-8, 1994.
[36] M.D. Neil, "Statistical Modelling of Software Metrics," PhD thesis, South Bank Univ. and Strathclyde Univ., 1992.
[37] J.M. Voas and K.W. Miller, "Software Testability: The New Verification," *IEEE Software*, pp. 17-28, May 1995.
[38] A. Bertolino and L. Strigini, "On the Use of Testability Measures for Dependability Assessment," *IEEE Trans. Software Eng.*, vol. 22, no. 2, pp. 97-108, 1996.
[39] M. Diaz and J. Sligo, "How Software Process Improvement Helped Motorola," *IEEE Software*, vol. 14, no. 5, pp. 75-81, 1997.
[40] C. Jones, "The Pragmatics of Software Process Improvements," *Software Engineering Technical Council Newsletter, Technical Council on Software Eng.*, IEEE Computer Society, vol. 14 no. 2, Winter 1996.
[41] J.C. Munson and T.M. Khoshgoftaar, "Regression Modelling of Software Quality: An Empirical Investigation," *Information and Software Technology*, vol. 32, no. 2, pp. 106-114, 1990.
[42] M.D. Neil, "Multivariate Assessment of Software Products," *J. Software Testing, Verification and Reliability*, vol. 1, no. 4, pp. 17-37, 1992.
[43] T.M. Khoshgoftaar and J.C. Munson, "Predicting Software Development Errors Using Complexity Metrics," *IEEE J. Selected Areas in Comm.*, vol. 8, no. 2, pp. 253-261, 1990.
[44] J.C. Munson and T.M. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 423-433, 1992.
[45] N.E. Fenton and S. Lawrence Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, second edition, Int'l Thomson Computer Press, 1996.
[46] E. Adams, "Optimizing Preventive Service of Software Products," *IBM Research J.*, vol. 28, no. 1, pp. 2-14, 1984.
[47] N. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.*, 1999. to appear
[48] T. Stalhane, "Practical Experiences with Safety Assessment of a System for Automatic Train Control," *Proc. SAFECOMP'92*, Zurich, Switzerland, Oxford, U.K.: Pergamon Press, 1992.
[49] P. Hamer and G. Frewin, "Halstead's Software Science: A Critical Examination," *Proc. Sixth Int'l Conf. Software Eng.*, pp. 197-206, 1982.
[50] V.Y. Shen, S.D. Conte, and H. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support," *IEEE Trans. Software Eng.*, vol. 9, no. 2, pp. 155-165, 1983.
[51] M.J. Shepperd, "A Critique of Cyclomatic Complexity as a Software Metric," *Software Eng. J.*, vol. 3, no. 2, pp. 30-36, 1988.
[52] B.F. Manly, *Multivariate Statistical Methods: A Primer*. Chapman & Hall, 1986.
[53] F. Zhou, B. Lowther, P. Oman, and J. Hagemeister, "Constructing and Testing Software Maintainability Assessment Models," *First Int'l Software Metrics Symp.*, Baltimore, Md., IEEE CS Press, 1993.
[54] J. Rosenberg, "Some Misconceptions About Lines of Code," *Software Metrics Symp.*, pp. 37-142, IEEE Computer Society, 1997.
[55] B.A. Kitchenham, "An Evaluation of Software Structure Metrics," *Proc. COMPSAC'88*, Chicago Ill., 1988.
[56] S. Cherf, "An Investigation of the Maintenance and Support Characteristics of Commercial Software," *Proc. Second Oregon Workshop on Software Metrics (AOWSM)*, Portland, 1991.
[57] S.L. Lauritzen and D.J. Spiegelhalter, "Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems (with discussion)," *J.R. Statistical Soc. Series B*, 50, no. 2, pp. 157-224, 1988.
[58] HUGIN Expert Brochure. Hugin Expert A/S, Aalborg, Denmark, 1998.
[59] Agena Ltd, "Bayesian Belief Nets," http://www.agena.co.uk/bbnarticle/bbns.html
[60] M. Neil and N.E. Fenton, "Predicting Software Quality Using Bayesian Belief Networks," *Proc 21st Ann. Software Eng. Workshop*, pp. 217-230, NASA Goddard Space Flight Centre, Dec. 1996.
[61] M. Neil, B. Littlewood, and N. Fenton, "Applying Bayesian Belief Networks to Systems Dependability Assessment," *Proc. Safety Critical Systems Club Symp.*, Springer-Verlag, Leeds, Feb. 1996.

Norman E. Fenton is professor of computing science at the Centre for Software Reliability, City University, London and is also a director at Agena Ltd. His research interests include software metrics, empirical software engineering, safety critical systems, and formal development methods. However, the focus of his current work is on applications of Bayesian nets; these applications include critical systems' assessment, vehicle reliability prediction, and software quality assessment. He is a chartered engineer (member of the IEE), a fellow of the IMA, and a member of the IEEE Computer Society.

Martin Neil holds a first degree in mathematics for business analysis from Glasgow Caledonian University and has achieved a PhD in statistical analysis of software metrics jointly from South Bank University and Strathclyde University. Currently he is a lecturer in computing at the Centre for Software Reliability, City University, London. Before joining the CSR, He spent three years with Lloyd's Register as a consultant and researcher and a year at South Bank University. He has also worked with J.P. Morgan as a software quality consultant. His research interests cover software metrics, Bayesian probability, and the software process. Dr. Neil is a director at Agena Ltd., a consulting company specializing in decision support and risk assessment of safety and business critical systems. He is a member of the CSR Council, the IEEE Computer Society, and the ACM.