ELSEVIER

The Journal of Systems and Software 47 (1999) 149–157

The Journal of Systems and Software

www.elsevier.com/locate/jss

Software metrics: successes, failures and new directions

Norman E. Fenton *, Martin Neil

Centre for Software Reliability, City University, Northampton Square, London EC1V OHB, UK

Abstract

The history of software metrics is almost as old as the history of software engineering. Yet, the extensive research and literature on the subject has had little impact on industrial practice. This is worrying given that the major rationale for using metrics is to improve the software engineering decision making process from a managerial and technical perspective. Industrial metrics activity is invariably based around metrics that have been around for nearly 30 years (notably *Lines of Code* or similar size counts, and defects counts). While such metrics can be considered as massively successful given their popularity, their limitations are well known, and mis-applications are still common. The major problem is in using such metrics in isolation. We argue that it *is* possible to provide genuinely improved management decision support systems based on such simplistic metrics, but only by adopting a less isolationist approach. Specifically, we feel it is important to explicitly model: (a) cause and effect relationships and (b) uncertainty and combination of evidence. Our approach uses Bayesian Belief nets, which are increasingly seen as the best means of handling decision-making under uncertainty. The approach is already having an impact in Europe. © 1999 Elsevier Science Inc. All rights reserved.

1. Introduction

'Software metrics' is the rather misleading collective term used to describe the wide range of activities concerned with measurement in software engineering. These activities range from producing numbers that characterise properties of software code (these are the classic software 'metrics') through to models that help predict software resource requirements and software quality. The subject also includes the quantitative aspects of quality control and assurance – and this covers activities like recording and monitoring defects during development and testing. Although there are assumed to be many generic benefits of software metrics (see for example the books by DeMarco, 1982; Gilb, 1988; Fenton and Pfleeger, 1996; Jones, 1991), it is fair to say that the most significant is to provide information to support managerial decision-making during software development and testing. With this focus we provide here a dispassionate assessment of the impact of software metrics (both industrially and academically) and the direction we feel the subject might now usefully take.

We begin our assessment of the status of software metrics with a necessary look at its history in Section 2. It is only in this context that we can really judge its successes and failures. These are described in Section 3. If we judge the entire software metrics subject area by the extent to which the level of metrics activity has increased in industry then it has been a great success. However, the increased industrial activity is not related to the increased academic and research activity. The metrics being practised are essentially the same basic metrics that were around in the late 1960s. In other words these are metrics based around LOC (or very similar) size counts, defect counts and crude effort figures (in person months). Industrialists have accepted the simple metrics and rejected the esoteric because they are easy to understand and relatively simple to collect. Our objective is to use primarily these simple metrics to build management decision support tools that combine different aspects of software development and testing and enable managers to make many kinds of predictions, assessments and trade-offs during the software life-cycle. Most of all our objective is to handle the key factors largely missing from the usual metrics models: *uncertainty and combining different* (*often subjective*) *evidence*. In Section 4 we explain why traditional (largely regression-based) models are inadequate for this purpose. In Section 5 we explain how the exciting technology of Bayesian nets (which is hidden to users) helps us meet our objective.

* Corresponding author. Tel.: +44 171 477 8425; fax: +44 171 477 8585; e-mail: n.fenton@csr.city.ac.uk

2. History of software metrics as a subject area

To assess the current status of software metrics, and its successes and failures, we need to consider first its history. Although the first dedicated book on software metrics was not published until (Gilb, 1976), the history of active software metrics dates back to the late 1960s. Then the Lines of Code measure (LOC or KLOC for thousands of lines of code) was used routinely as the basis for measuring both programmer productivity (LOC per programmer month) and program quality (defects per KLOC). In other words LOC was being used as a surrogate measure of different notions of program *size*. The early resource prediction models (such as those of Putnam, 1978; Boehm, 1981) also used LOC or related metrics like *delivered source instructions* as the key size variable. Akiyama (1971) published what we believe was the first attempt to use metrics for software quality *prediction* when he proposed a crude regression-based model for module defect density (number of defects per KLOC) in terms of the module size measured in KLOC. In other words he was using KLOC as a surrogate measure for program complexity.

The obvious drawbacks of using such a crude measure as LOC as a surrogate measure for such different notions of program size such as effort, functionality and complexity, were recognised in the mid-1970s. The need for more discriminating measures became especially urgent with the increasing diversity of programming languages. After all, an LOC in an assembly language is not comparable in effort, functionality, or complexity to an LOC in a high-level language. Thus, the decade starting from the mid-1970s saw an explosion of interest in measures of software complexity (pioneered by the likes of Halstead, 1977; McCabe, 1976) and measures of size (such as function points pioneered by Albrecht, 1979 and later by Symons, 1991) which were intended to be independent of programming language.

Work on extending, validating and refining complexity metrics (including applying them to new paradigms such as object oriented languages, Chidamber and Kemerer, 1994) has been a dominant feature of academic metrics research up to the present day (Fenton, 1991; Zuse, 1991).

In addition to work on specific metrics and models, much recent work has focused on meta-level activities, the most notable of which are as follows.

Work on the mechanics of implementing metrics programs. Two pieces of work stand out in this respect:

1. The work of Grady and Caswell (1987) (later extended in Grady, 1992) which was the first and most extensive experience report of a company-wide software metrics program. This work contains key guidelines (and lessons learned) which influenced (and inspired) many subsequent metrics programs.

2. The work of Basili, Rombach and colleagues on GQM (Goal-Question Metric) (Basili and Rombach, 1988). By borrowing some simple ideas from the Total Quality Management field, Basili and his colleagues proposed a simple scheme for ensuring that metrics activities were always goal-driven. A metrics program established *without* clear and specific goals and objectives is almost certainly doomed to fail (Hall and Fenton, 1997). Basili's high profile in the community and outstanding communications and technology transfer skills ensured that this important message was subsequently widely accepted and applied. That does not mean it is without its criticisms (Bache and Neil, 1995; Hetzel, 1993 argue that the inherent top-down approach ignores what is feasible to measure at the bottom). However, most metrics programs at least pay lip service to GQM with the result that such programs should in principle be collecting only those metrics which are relevant to the specific goals.

The use of metrics in empirical software engineering. Specifically we refer to empirical work concerned with evaluating the effectiveness of specific software engineering methods, tools and technologies. This is a great challenge for the academic/research software metrics community. There is now widespread awareness that we can no longer rely purely on the anecdotal claims of self-appointed experts about which new methods really work and how. Increasingly we are seeing measurement studies that quantify the effectiveness of methods and tools. Basili and his colleagues have again pioneered this work (see, for example, Basili and Reiter, 1981; Basili et al., 1986). Success here is judged by acceptance of empirical results, and the ability to repeat experiments to independently validate results.

Work on theoretical underpinnings of software metrics. This work (exemplified by Briand et al., 1996; Fenton, 1991; Zuse, 1991) is concerned with improving the level of rigour in the discipline as a whole. For example, there has been considerable work in establishing a measurement theory basis for software metrics activities. The most important success from this work has been the acceptance of the need to consider scale types when defining measures with restrictions on the metrics analyses techniques that are relevant for the given scale type.

3. Successes and failures

First the good news about software metrics and its related activities.

It has been a phenomenal success if we judge it by the number of books, journal articles, academic research projects, and dedicated conferences. The number of published metrics papers grows at an exponential rate (CSR maintains a database of such papers which currently numbers over 1600). There are now at least 40 books on software metrics, which is as many as there on the general topic of software engineering. Most computing/software engineering courses world-wide now include some compulsory material on software metrics. In other words the subject area has now been accepted as part of the mainstream of software engineering.

There has been a significant increase in the level of software metrics activity actually taking place in industry. Most major IT companies, and even many minor ones, have in place software metrics 'programs', even if they are not widely recognised as such. This was not happening 10 years ago.

Now for the bad news:

There is no match in content between the increased level of metrics activity in academia and industry. Like many other core subjects within the software engineering mainstream (such as formal development methods, object-oriented design, and even structured analysis), the industrial take-up of most academic software metrics work has been woeful. Much of the increased metrics activity in industry is based almost entirely on metrics that were around in the early 1970s.

Much academic metrics research is inherently irrelevant to industrial needs. Irrelevance here is at two levels.

1. *Irrelevance in scope*: Much academic work has focused on metrics which can only ever be applied/computed for small programs, whereas all the reasonable objectives for applying metrics are relevant primarily for large systems. Irrelevance in scope also applies to the many academic models which rely on parameters which could never be measured in practice.

2. *Irrelevance in content*: Whereas the pressing industrial need is for metrics that are relevant for process improvement, much academic work has concentrated on detailed code metrics. In many cases these aim to measure properties that are of little practical interest. This kind of irrelevance prompted Glass to comment:

> What theory is doing with respect to measurement of software work and what practice is doing are on two different planes, planes that are shifting in different directions (Glass, 1994).

Much industrial metrics activity is poorly motivated. For all the warm feelings associated with the objectives for software metrics (improved assessment and prediction, quality control and assurance, etc.) it is not at all obvious that metrics are inevitably a 'good thing'. The decision by a company to put in place some kind of a metrics program is inevitably a 'grudge purchase'; it is something done when things are bad or to satisfy some external assessment body. For example, in the US the single biggest trigger for industrial metrics activity has been the CMM (Humphrey, 1989); evidence of use of metrics is intrinsic for achieving higher levels of CMM. Just as there is little empirical evidence about the effectiveness of specific software development methods (Fenton et al., 1994) so there is equally little known about the effectiveness of software metrics. Convincing success stories describing the long-term payback of metrics are almost non-existent. What we do know is that metrics will always be an overhead on your current software projects (we found typically it would be 4–8%, Hall and Fenton, 1997). When deadlines are tight and slipping it is inevitable that the metrics activity will be one of the first things to suffer (or be stopped completely). Metrics are, above all, effective primarily as a management tool. Yet good metrics data is possible only with the commitment of technical staff involved in development and testing. There are no easy ways to motivate such staff in this respect.

Much industrial metrics activity is poorly executed: We have come across numerous examples of industrial practice which ignores well-known guidelines of best practice data-collection and analysis and applies metrics in ways that were known to be invalid 20 years ago. For example, it is still common for companies to collect defect data which does not distinguish between defects discovered in operation and defects discovered during development (Fenton and Neil, 1999).

The raison d'être of software metrics is to improve the way we monitor, control or predict various attributes of software and the commercial software production process. Therefore, a necessary criterion to judge the success of software metrics is the extent to which they are being routinely used across a wide cross section of the software production industry. Using this criterion, and based solely on our own consulting experience and the published literature, the only metrics to qualify as *candidates* for successes are:

The enduring LOC metric. Despite the many convincing arguments about why this metric is a very poor 'size' measure (Fenton and Pfleeger, 1996) it is still routinely used in its original applications: as a normalising measure for software quality (defects per KLOC); as a means of assessing productivity (LOC per programmer month) and as a means of providing crude cost/effort estimates. Like it or not LOC has been a 'success' because it is easy to compute and can be 'visualised' in the sense that we understand what a 100 LOC program looks like.

Metrics relating to defect counts (see Gilb and Graham, 1994; Fenton and Pfleeger, 1996 for extensive examples). Almost all industrial metrics programs incorporate some attempt to collect data on software defects discovered during development and testing. In many cases, such programs are not recognised explicitly as 'metrics programs' since they may be regarded simply as applying good engineering practice and configuration management.

McCabe's cyclomatic number (McCabe, 1976). For all the many criticisms that it has attracted this remains exceptionally popular. It is easily computed by static analysis (unlike most of the more discriminating complexity metrics) and it is widely used for quality control purposes. For example, Grady (1992) reports that, at Hewlett Packard, any modules with a cyclomatic complexity higher than 16 must be re-designed.

To a less extent we should also include:

Function points (Albrecht, 1979; Symons, 1991). Function point analysis is very hard to do properly (Jeffery and Stathis, 1996) and is unnecessarily complex if used for resource estimation (Kitchenham, 1995). Yet, despite these drawbacks function points appear to have been widely accepted as a standard sizing measure, especially within the financial IT sector (Onvlee, 1995).

This is not an impressive list, and in Section 4 we will explain why the credibility of metrics as a whole has been lowered because of the crude way in which these kinds of metrics have been used. However, as we will show in Section 5, it is the models and applications which have been fundamentally flawed rather than the metrics.

4. Limitations of metrics for predicting software quality

Ultimately one of the major objectives of software metrics research has been to produce metrics which are good 'early' predictors of reliability (the frequency of operational defects). We assume that readers already acknowledge the need to distinguish between defects, which are discovered at different life-cycle phases. Most notably defects discovered in operation (we call these *failures*) must always be distinguished from defects discovered during development (which we refer to as faults since they *may or may not* lead to failures in operation).

It is known that stochastic reliability growth models can produce accurate predictions of the reliability of a software system providing that a reasonable amount of failure data can be collected for that system in representative operational use (Brocklehurst and Littlewood, 1992). Unfortunately, this is of little help in those many circumstances when we need to make predictions before the software is operational. In Fenton and Neil (1999) we reviewed the many studies advocating statistical models and metrics to address this problem. Our opinion of this work is not high – we found that many methodological and theoretical mistakes have been made. Studies suffered from a variety of flaws ranging from model mis-specification to use of inappropriate data. We concluded 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.

In Fenton and Ohlsson (1999) we used case study data from two releases of a major commercial system to test a number of popular hypotheses about the basic metrics. The system was divided into many hundreds of modules. The basic metrics data were collected at the module level (the metrics were: *defects found* at four different testing phases, including in operation, *LOC* and a number of complexity metrics including *cyclomatic complexity*). We were especially interested in hypotheses that lie at the root of the popularity for the basic metrics. Many of these are based around the idea that a small number of the modules inevitably have a disproportionate number of the defects; the assumption is then that metrics can help us to identify early in the development cycle such fault and failure prone modules.

The hypotheses and results are summarised in Table 1. We make no claims about the generalisation of these results. However, given the rigour and extensiveness of the data-collection and also the strength of some of the observations, we feel that there are lessons to be learned by the wider community.

The result for hypothesis 4 is especially devastating for much software metrics work. The rationale behind this hypothesis is the belief in the inevitability of 'rogue modules' – a relatively small proportion of modules in a system that account for most of the faults and which are likely to be fault-prone both pre- and post-release. It is often assumed that such modules are somehow intrinsically complex, or generally poorly built. This also provides the rationale for complexity metrics. For example, Munson and Khosghoftaar asserted:

> 'There is a clear intuitive basis for believing that complex programs have more faults in them than simple programs' (Munson and Khoshgoftaar, 1992).

Not only was there no evidence to support hypothesis 4, but there was evidence to support a converse hypothesis. In both releases almost all of the faults discovered in pre-release testing appear in modules which subsequently revealed almost no operational faults.

These remarkable results are also closely related to the empirical phenomenon observed by Adams (1984) – that most operational system failures are caused by a small proportion of the latent faults. The results have major ramifications for the commonly used *fault density* metric as a de-facto measure of user perceived *software quality*. If fault density is measured in terms of pre-release faults (as is common), then at the module level this measure tells us worse than nothing about the quality of the module; a high value is more likely to be an indicator of extensive testing than of poor quality. Modules with high fault density pre-release are likely to have low fault-density post-release, and vice versa. The results of hypothesis 4 also bring into question the entire rationale

Table 1
Support for the hypotheses provided in this case study

Number	Hypothesis	Case study evidence?
1a	A small number of modules contain most of the faults discovered during pre-release testing	Yes – evidence of 20–60 rule
1b	If a small number of modules contain most of the faults discovered during pre-release testing then this is simply because those modules constitute most of the code size	No
2a	A small number of modules contain most of the operational faults	Yes – evidence of 20–80 rule
2b	If a small number of modules contain most of the operational faults then this is simply because those modules constitute most of the code size	No – strong evidence of a converse hypothesis
3	Modules with higher incidence of faults in early pre-release likely to have higher incidence of faults in system testing	Weak support
4	Modules with higher incidence of faults in all pre-release testing likely to have higher incidence of faults in post-release operation	No – strongly rejected
5a	Smaller modules are less likely to be failure prone than larger ones	No
5b	Size metrics (such as LOC) are good predictors of number of pre-release faults in a module	Weak support
5c	Size metrics (such as LOC) are good predictors of number of post-release faults in a module	No
5d	Size metrics (such as LOC) are good predictors of a module's (pre-release) fault-density	No
5e	Size metrics (such as LOC) are good predictors of a module's (post-release) fault-density	No
6	Complexity metrics are better predictors than simple size metrics of fault and failure-prone modules	No (for cyclomatic complexity), but some weak support for metrics based on SigFF
7	Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system	Yes
8	Software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases	Yes

for the way software complexity metrics are used and validated. The ultimate aim of complexity metrics is to predict modules that are fault-prone post-release. Yet, most previous 'validation' studies of complexity metrics have deemed a metric 'valid' if it correlates with the (pre-release) fault density. Our results suggest that valid metrics may therefore be inherently poor at predicting what they are supposed to predict.

What we did confirm was that complexity metrics are closely correlated to size metrics like LOC. While LOC (and hence also the complexity metrics) are reasonable predictors of absolute number of faults, they are very poor predictors of fault density.

5. New directions

The results in Section 4 provide mixed evidence about the usefulness of the commonly used metrics. Results for hypotheses 1, 2, 3, 7 and 8 explain why there is still so much interest in the potential of defects data to help quality prediction. But the other results confirm that the simplistic approaches are inevitably inadequate. It is fair to conclude that:

- Complexity and/or size measures alone cannot provide accurate predictions of software defects.
- On its own, information about software defects (discovered pre-release) provides no information about likely defects post-release.
- Traditional statistical (regression-based) methods are inappropriate for defects prediction.

In addition to the problem of using metrics data in isolation, the major weakness of the simplistic approaches to defect prediction has been a misunderstanding of the notion of *cause and effect*. A correlation between two metric values (such as a module's size and the number of defects found in it) does *not* provide evidence of a causal relationship. Consider, for example, our own result for hypothesis 4. The data we observed can be explained by the fact that the modules in which few faults are discovered during testing may simply not have been tested properly. Those modules that reveal large numbers of faults during testing may genuinely be very well tested in the sense that *all* the faults really are 'tested out of them'. The key missing explanatory data in this case is, of course, *testing effort*, which was unfortunately not available to us in this case study. As another example, hypothesis 5 is the popular software engineering assumption that small modules are likely to be (proportionally) less failure prone. In other words small modules have a lower defect density. In fact, empirical studies summarised in Hatton (1997) suggest the opposite effect: that large modules have a lower fault density than small ones. We found no evidence to support either hypothesis. Again this is because the association between size and fault density is not a causal one. It is for this kind of reason that we recommend more complete models that enable us to augment the empirical observations with other explanatory factors, most notably, *testing effort* and *operational usage*. If you do not test or use a module, you will not observe faults or failures associated with it.

Thus, the challenge for us is to produce models of the software development and testing process which take account of the crucial concepts missing from traditional statistical approaches. Specifically we need models that can handle:

- diverse process and product evidence,
- genuine cause and effect relationships,
- uncertainty,
- incomplete information.

At the same time the models must not introduce any additional metrics overheads, either in terms of the amount of data-collection or the sophistication of the metrics.

After extensive investigations during the DATUM project 1993–1996 into the range of suitable formalisms (Fenton et al., 1998) we concluded that Bayesian belief nets (BBNs) were by far the best solution for our problem. The only remotely relevant approach we found in the software engineering literature was the process simulation method of Abdel-Hamid (1996), but this did not attempt to model the crucial notion of uncertainty.

BBNs have attracted much recent attention as a solution for problems of decision support under uncertainty. Although the underlying theory (Bayesian probability) has been around for a long time, building and executing realistic models has only been made possible because of recent algorithms (see Jensen, 1996) and software tools that implement them (Hugin, 1989). 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. A number of recent projects in Europe, most of which we have been involved in, have pioneered the use of BBNs in the area of software assessment (especially for critical systems) (Fenton et al., 1998). Other related applications include their use in time-critical decision making for the propulsion systems on the Space Shuttle (Horvitz and Barry, 1995).

A BBN is a graphical network (such as that shown in Fig. 1) together with an associated set of probability tables. The nodes represent uncertain variables and the arcs represent the causal/relevance relationships between the variables. The probability tables for each node provide the probabilities of each state of the variable for that node. For nodes without parents these are just the marginal probabilities while for nodes with parents these are conditional probabilities for each combination of parent state values. The BBN in Fig. 1 is a simplified version of the one that was built in collaboration with the partner in the case study described in Section 4.

It is simplified in the sense that it only models one pre-release testing/rework phase whereas in the full version there are three. Like all BBNs the probability tables were provided by a mixture of: empirical/benchmarking data and subjective judgements. We have also developed (in collaboration with Hugin A/S) tools for (a) generating quickly very large probability tables (including continuous variable nodes) and (b) building large BBN topologies. Thus, we feel we have to a large

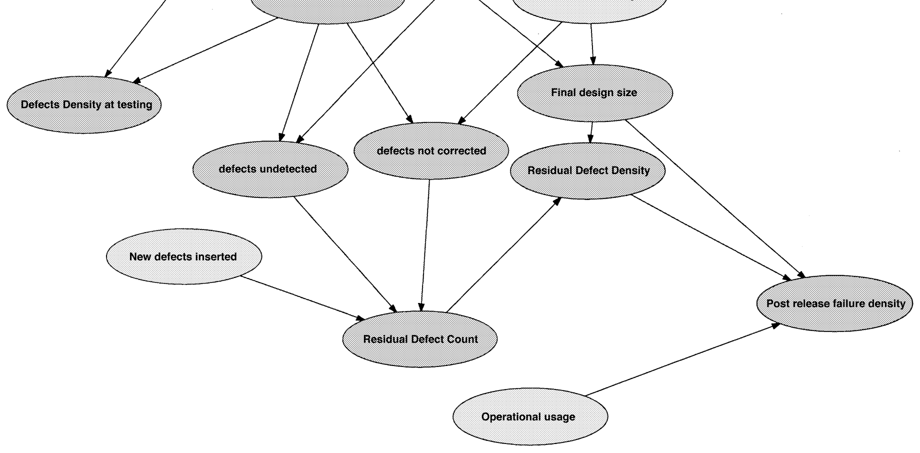

Fig. 1. A BBN that models the software defects insertion and detection process.

extent cracked the problem (of tractability) that inhibited more widespread use of BBNs. In recent applications we have built BBNs with several hundred nodes and many millions of probability values. It is beyond the scope of this paper to describe either BBNs or the particular example in detail (see Fenton and Neil, 1999 for a fuller account). However, we can give a feel for its power and relevance.

First note that the model in Fig. 1 contains a mixture of variables we might have evidence about and variables we are interested in predicting. At different times during development and testing different information will be available, although some variables such as 'number of defects inserted' will never be known with certainty. With BBNs, it is possible to propagate consistently the impact of evidence on the probabilities of uncertain outcomes. For example, suppose we have evidence about the number of defects discovered in testing. When we enter this evidence all the probabilities in the entire net are updated.

In Fig. 2 we provide a screen dump from the Hugin tool of what happens when we enter some evidence into the BBN. In order for this to be displayed in one screen we have used a massive simplification for the range of values in each node (we have restricted it to three values: low, medium, high), but this is still sufficient to illustrate the key points. Where actual evidence is entered this is represented by the dark coloured bars. For example, we have entered evidence that the design complexity for the module is high and the testing accuracy is low. When we execute the BBN all other probabilities are propagated. Some nodes (such as 'problem complexity') remain unchanged in its original uniform state (representing the fact that we have no information about this node), but others are changed significantly. In this particular scenario we have an explanation for the apparently paradoxical results on pre- and post-release defect density. Note that the defect density discovered in testing (pre-release) is likely to be 'low' (with probability 0.75). However, the post-release failure density is likely to be 'high' (probability 0.63).

This is explained primarily by the evidence of high operational usage, low testing accuracy, and to a lesser extent the low design effort.

The example BBN is based only on metrics data that was either already being collected or could easily be provided by project personnel. Specifically it assumes that the following data may be available for each module:

- defect counts at different testing and operation phases,
- size and 'complexity' metrics at each development phase (notably LOC, cyclomatic complexity),
- the kind of benchmarking data described in hypotheses 7 and 8 of Table 1,
- approximate development and testing effort.

At any point in time there will always be some missing data. The beauty of a BBN is that it will

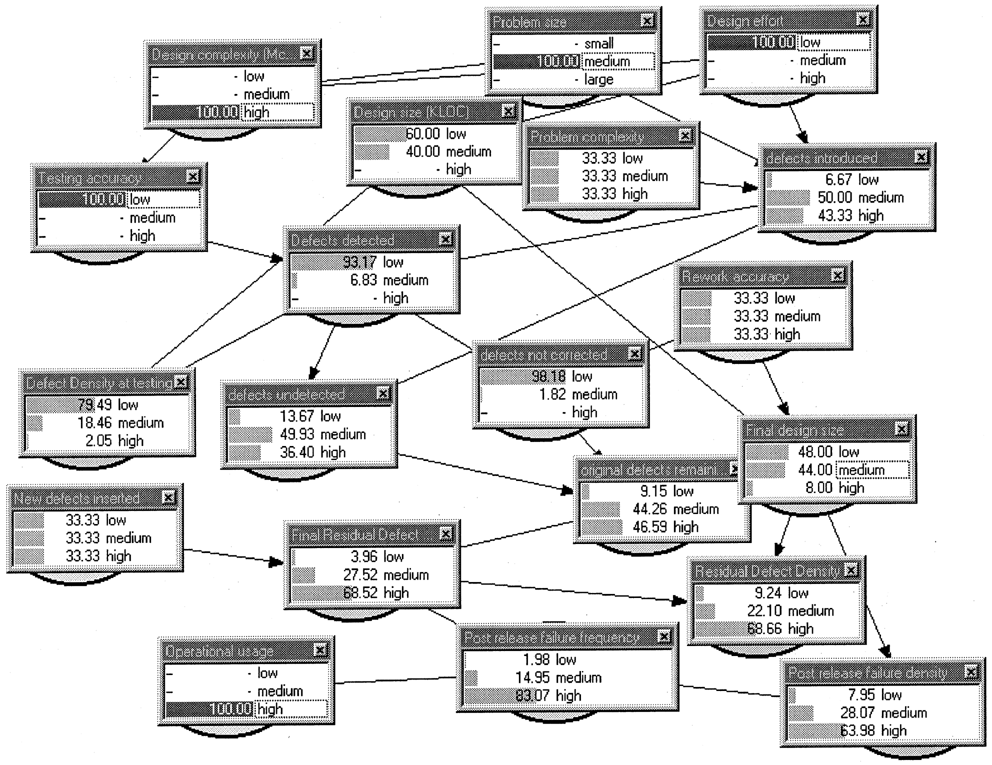

Fig. 2. Entering evidence into the BBN – an explanation for the 'paradox' of low defect density pre-release but high defect density post-release.

compute the probability of every state of every variable irrespective of the amount of evidence. Lack of substantial hard evidence will be reflected in greater uncertainty in the values.

In addition to the result that high defect density pre-release may imply low defect density post-release, we have used this BBN to explain other empirical results that were discussed in Section 4, such as the scenario whereby larger modules have lower defect densities.

The benefits of using BBNs include:

- Explicit modelling of 'ignorance' and uncertainty in estimates, as well as cause-effect relationships.
- Makes explicit those assumptions that were previously hidden – hence adds visibility and auditability to the decision-making process.
- Intuitive graphical format makes it easier to understand chains of complex and seemingly contradictory reasoning.
- Ability to forecast with missing data.
- Use of 'what-if?' analysis and forecasting of effects of process changes.
- Use of subjectively or objectively derived probability distributions.
- Rigorous, mathematical semantics for the model.
- No need to do any of the complex Bayesian calculations, since tools like Hugin do this.

Clearly the ability to use BBNs to predict defects will depend largely on the stability and maturity of the development processes. Organisations that do not collect the basic metrics data, do not follow defined life-cycles or do not perform any forms of systematic testing will not be able to apply such models effectively. This does not mean to say that less mature organisations cannot build reliable software, rather it implies that they cannot do so predictably and controllably.

6. Conclusions

The increased industrial software metrics activity seen in the last 10 years may be regarded as a successful symptom of an expanding subject area. Yet, this industrial practice is not related in content to the increased academic and research activity. The metrics being practised are essentially the same basic metrics that were around in the late 1960s. In other words these are metrics based around LOC (or very similar) size counts, defect counts, and crude effort figures (in person months). The mass of academic metrics research has failed almost totally in terms of industrial penetration. This is a particularly devastating indictment given the inherently 'applied' nature of the subject. Having been heavily involved ourselves in academic software metrics research the obvious temptation would be to criticise the short-sightedness of industrial software developers. Instead, we recognise that industrialists have accepted the simple metrics and rejected the esoteric for good reasons. They are easy to understand and relatively simple to collect. Since the esoteric alternatives are not obviously more 'valid' there is little motivation to use them. We are therefore using industrial reality to fashion our research. One of our objectives is to develop metrics-based management decision support tools that build on the relatively simple metrics that we know are already being collected. These tools combine different aspects of software development and testing and enable managers to make many kinds of predictions, assessments and trade-offs during the software life-cycle, without any major new metrics overheads. Most of all, our objective is to handle the key factors largely missing from the usual metrics models: *uncertainty and combining different* (*often subjective*) *evidence*. Traditional (largely regression-based) models are inadequate for this purpose. We have shown how the exciting technology of Bayesian nets (whose complexities are hidden to users) helps us meet our objective. The approach does, however, force managers to make explicit those assumptions that were previously hidden in their decision-making process. We regard this as a benefit rather than a drawback. The BBN approach is actually being used on real projects and is receiving highly favourable reviews. We believe it is an important way forward for metrics research.

Acknowledgements

The work was supported, in part, by the EPSRC-funded project IMPRESS, and the ESPRIT-funded projects DEVA and SERENE. We are also indebted to Niclas Ohlsson for his work on the case study described in this paper.

References

Abdel-Hamid, T.K., 1996. The slippery path to productivity improvement. IEEE Software 13 (4), 43–52.

Adams, E., 1984. Optimizing preventive service of software products. IBM Research Journal 28 (1), 2–14.

Akiyama, F., 1971. An example of software system debugging. Inf Processing 71, 353–379.

Albrecht, A.J., 1979. Measuring Application Development. Proceedings of IBM Applications Development joint SHARE/GUIDE symposium. Monterey CA, pp. 83–92.

Bache, R., Neil, M., 1995. Introducing metrics into industry: a perspective on GQM. In: Fenton, N.E., Whitty, R.W., Iizuka, Y. (Eds.), Software Quality Assurance and metrics: A Worldwide Prespective. International Thomson Press, pp. 59–68.

Basili, V.R., Rombach, H.D., 1988. The TAME project: Towards improvement-oriented software environments. IEEE Transactions on Software Engineering 14 (6), 758–773.

Basili, V.R., Reiter, R.W., 1981. A controlled experiment quantitatively comparing software development approaches. IEEE Transactions on Software Engineering 7 (3).

Basili, V.R., Selby, R.W., Hutchens, D.H., 1986. Experimentation in software engineering. IEEE Transactions on Software Engineering 12 (7), 733–743.

Boehm, B.W., 1981. Software Engineering Economics. Prentice-Hall, New York.

Briand, L.C., Morasca, S., Basili, V.R., 1996. Property-based software engineering measurement. IEEE Transactions on Software Engineering 22 (1), 68–85.

Brocklehurst, S., Littlewood, B., 1992. New ways to get accurate software reliability modelling. IEEE Software, pp. 34–42.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineeing 20 (6), 476–498.

DeMarco, T., 1982. Controlling Software Projects. Yourden Press, New York.

Fenton, N.E., 1991. Software Metrics: A Rigorous Approach. Chapman & Hall, London.

Fenton, N.E., Littlewood, B., Neil, M., Strigini, L., Sutcliffe, A., Wright, D., 1998. Assessing dependability of safety critical systems using diverse evidence. IEE Proceedings Software Engineering 145 (1), 35–39.

Fenton, N.E., Neil, M., 1999. A critique of software defect prediction models. IEEE Transactions on Software Engineering.

Fenton, N.E., Ohlsson, N., 1999. Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering, to appear.

Fenton, N.E., Pfleeger, S.L., 1996. Software Metrics: A Rigorous and Practical Approach, 2nd ed. International Thomson Computer Press.

Fenton, N.E., Pfleeger, S.L., Glass, R., 1994. Science and substance: a challenge to software engineers. IEEE Software 11 (4), 86–95.

Gilb, T., Graham, D., 1994. Software Inspections. Addison–Wesley, Wokingham, UK.

Gilb, T., 1988. Principles of Software Engineering Management. Addison–Wesley, Wokingham, UK.

Gilb, T., 1976. Software Metrics. Chartwell-Bratt.

Glass, R.L., 1994. A tabulation of topics where software practice leads software theory. Journal of Systems Software 25, 219–222.

Grady, R., Caswell, D., 1987. Software Metrics: Establishing a Company-wide Program. Prentice-Hall, Englewood Cliffs, NJ.

Grady, R.B, 1992. Practical Software Metrics for Project Management and Process Improvement. Prentice-Hall, Englewood Cliffs, NJ.

Hall, T., Fenton, N.E., 1997. Implementing effective software metrics programmes. IEEE Software 14 (2), 55–66.

Halstead, M., 1977. Elements of Software Science. North-Holland, Amsterdam.

Hatton, L., 1997. Reexamining the fault density – component size connection. IEEE Software 14 (2), 89–97.

Hetzel, W.C., 1993. Making Software Measurement Work: Building an Effective Software Measurement Program. QED Publishing Group, Wellesley, MA.

Horvitz, E., Barry, M., 1995. Display of information for time-critical decision making. Proceedings of 11th Conference on AI, Montreal.

Hugin, A.S., 1989. www.hugin.dk.

Humphrey, W.S., 1989. Managing the Software Process. Addison–Wesley, Reading, MA.

Jeffery, R., Stathis, J., 1996. Function point sizing: structure, validity and applicability. Empirical Software Engineering 1 (1), 11–30.

Jensen, F.V., 1996. An Introduction to Bayesian Networks. UCL Press.

Jones, C., 1991. Applied Software Measurement. McGraw-Hill, New York.

Kitchenham, B.A., 1995. Using function points for software cost estimation. In: Fenton, N.E., Whitty, R.W., Iizuka, Y. (Eds.), Software Quality Assurance and Measurement. International Thomson Computer Press, pp. 266–280.

McCabe, T., 1976. A software complexity measure. IEEE Transactions on Software Engineering 2 (4), 308–320.

Munson, J.C., Khoshgoftaar, T.M., 1992. The detection of fault-prone programs. IEEE Transactions on Software Engineering 18 (5), 423–433.

Onvlee, J., 1995. Use of function points for estimation and contracts. In: Fenton, N., Whitty, R.W., Iizuka, Y. (Eds.), Software Quality Assurance and Metrics. pp. 88–93.

Putnam, L.H., 1978. A general empirical solution to the macro software sizing and estimating problem. IEEE Transactions on Software Engineering 4 (4), 345–361.

Symons, C.R., 1991. Software Sizing and Estimating: Mark II Function point Analysis. Wiley, New York.

Zuse, H., 1991. Software Complexity: Measures and Methods. De Gruyter, Berlin.

Norman Fenton is Professor of Computing Science at the Centre for Software Reliability, City University (London) and is also Managing Director of Agena Ltd, a company specialising in risk management and measurement for critical systems. He is a Chartered Engineer (member of the IEE). Norman's 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. Norman has produced numerous books and publications on software metrics and formal methods.

Martin Neil is a Lecturer in Computing at the Centre for Software Reliability, City University, London. He holds a first degree in 'Mathematics for Business Analysis' from Glasgow Caledonian University and achieved a PhD in 'Statistical Analysis of Software Metrices' jointly from South Bank University and Strathclyde University. Before joining the CSR Martin spent three years with Lloyd's Register as a consultant and researcher and a year at South Bank University. He has also worked with JP Morgan as a software quality consultant. His research interests cover software metrices, Bayesian probability and the software process. Martin is a member of the CSR Council, the IEEE Computer Society and the ACM. Martin is also a director of Agena, a consulting company specialising in decision support and risk assessment of safety and business critical systems.