Software Metrics: Roadmap

Norman E Fenton
Computer Science Department
Queen Mary and Westfield College,
London E1 4NS
UK
44 (0)208 530 5981
norman@dcs.qmw.ac.uk

Martin Neil
Computer Science Department
Queen Mary and Westfield College
London E1 4NS
UK
44 (0)1784 491588
martin@dcs.qmw.ac.uk

ABSTRACT
Software metrics as a subject area is over 30 years old, but it has barely penetrated into mainstream software engineering. A key reason for this is that most software metrics activities have not addressed their most important requirement: to provide information to support quantitative managerial decision-making during the software lifecycle. Good support for decision-making implies support for risk assessment and reduction. Yet traditional metrics approaches, often driven by regression-based models for cost estimation and defects prediction, provide little support for managers wishing to use measurement to analyse and minimise risk. The future for software metrics lies in using relatively simple existing 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. Our recommended approach is to handle the key factors largely missing from the usual metrics approaches, namely: *causality*, *uncertainty*, *and combining different (often subjective) evidence*. Thus the way forward for software metrics research lies in causal modelling (we propose using Bayesian nets), empirical software engineering, and multi-criteria decision aids.

Keywords
Software metrics, risk assessment, Bayesian belief nets, causal models, multi-criteria decision aid.

1 INTRODUCTION

Since this paper is intended to be a roadmap for software metrics it seems reasonable to motivate the core message with a motoring analogy. Data on car accidents in both the US and the UK reveal that January and February are the months when the fewest fatalities occur.

Thus, if you collect a database of fatalities organised by months and use this to build a regression model, your model would predict that it is safest to drive when the weather is coldest and roads are at their most treacherous. Such a conclusion is perfectly sensible *given the data available*, but intuitively we know it is wrong. The problem is that you do not have all the relevant data to make a sensible decision about the safest time to drive. Regression models often lead to misunderstanding about *cause* and *effect*. A correlation, such as that between *month of year* and *number of fatalities*, does not provide evidence of a *causal* relationship.

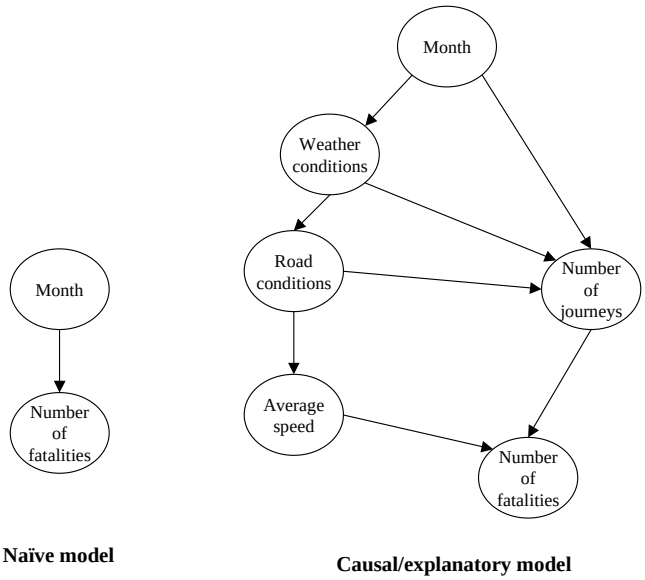

Figure 1: Predicting road fatalities

Of course the weather is more likely to be bad during the winter months and bad weather can cause treacherous road conditions. In these circumstances fewer people drive their cars and when they do they tend to drive more slowly. The

combination of fewer car journeys and slower speed is what leads to fewer fatalities.

Figure 1 summarises the above argument and shows the distinction between the original naïve regression-based approach and the latter causal-modelling approach. The causal model is *telling the story* that is missing from the naïve approach—you can use it to help you make intelligent decisions for risk reduction, and identify factors that you can control or influence. If you decide to make as many car journeys in February as in July, and if you decide to drive as fast irrespective of the road conditions then you are more likely to experience a fatal accident in February. The naïve regression model would not be able to provide you with this important information. The causal model would.

So what has this got to do with software metrics? Well, software metrics has been dominated by statistical models, such as regression models, when what is really needed are causal models.

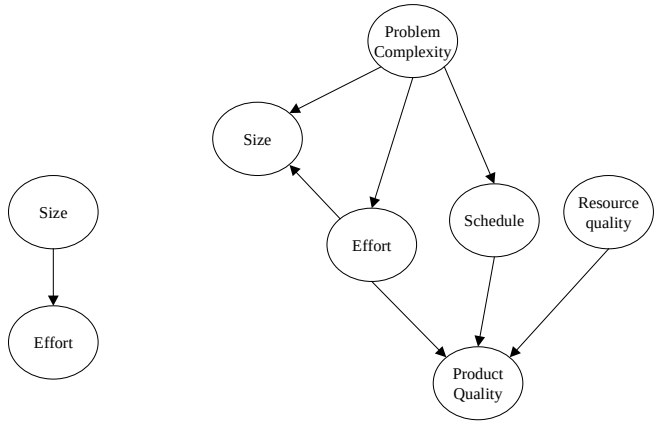

Figure 2 Software resource estimation

For example, much of software metrics has been driven by the need for resource prediction models. Usually this work has involved regression-based models of the form

$$effort = f(size)$$

Ignoring the fact that solution size cannot possibly *cause* effort to be expended, such models cannot be used effectively to provide decision support for risk assessment. This is because they do not have an explanatory framework. Without this managers do not know how to act upon the model's outputs to improve things. Causal modelling can provide an explanatory structure to explain events that can then be quantified.

For example, software managers would like to be able to use a cost model to answer the following kind of questions:

> "For a specification of this complexity, and given these limited resources, how likely am I to achieve a product of suitable quality?"
>
> "How much can I scale down the resources if I am prepared to put up with a product of specified lesser quality?"
>
> "The model predicts that I need 4 people over 2 years to build a system of this kind of size. But I only have funding for 3 people over one year. If I cannot sacrifice quality, how good do the staff have to be to build the systems with the limited resources?"

The causal model in Figure 2 could potentially be used to answer these questions. The regression model cannot.

There are the many generic claimed benefits of software metrics (see for example the books [16, 23, 27, 36]). However, the most significant is that they are supposed to provide information to support quantitative managerial decision-making during the software lifecycle [Stark]. Good support for decision-making implies support for risk assessment and reduction. The thesis of this paper is that the future of software metrics must be driven by this objective. Unfortunately, metrics research has failed largely to address it. It is for this reason above all others, we believe, that software metrics has failed to achieve a pivotal role within software engineering.

The aim of the paper is to explain how we can build on the existing body of knowledge to tackle the real objectives for software metrics. In section 2 we review the key metrics activities and explain how two activities (namely resource prediction and quality prediction) have driven the entire software metrics subject area since its inception. In Section 3 we explain the limitations of the usual regression-based approaches. Although the empirical data from regression-type models provide a valuable empirical basis, the models lack any causal structure, which is necessary if they are to be used for quantitative risk management. Thus, in Section 4 we explain how causal models, based on classical metrics and empirical results, provide a way forward. In Section 5 we map out a strategy for incorporating causal models, multi-criteria decision aids, and empirical software engineering

2 SIZE, RESOURCES, AND QUALITY: THE HISTORY OF SOFTWARE METRICS

Although the first dedicated book on software metrics was not published until 1976 [25], the history of active software metrics dates back to the mid-1960's when the *Lines of Code* metric was used as the basis for measuring programming productivity and effort. Despite a history that therefore officially predates "software engineering" (recognised as late as 1968), there is still little awareness of what software metrics actually is.

In fact *software metrics* is a collective term used to describe the very 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.

Since software engineering is ultimately an empirical subject, software metrics (as defined above) should by now have achieved a pivotal role within it. Yet, metrics continue to lie at the margins of software engineering. Moreover, they are often misunderstood, misused and even reviled. Theory and practice in software metrics have been especially out of step:

> "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" [26]

The recent increased industrial software metrics activity has not necessarily been the result of companies being convinced of their true efficacy. In fact, because the true objectives (and hence benefits) of software metrics have never been fully realised the decision by a company to put in place some kind of a metrics program is more often than not a 'grudge purchase'. It is something done when things are bad or when there is a need to satisfy some external assessment body. For example, in the US the single biggest trigger for industrial metrics activity has been the CMM [33], since evidence of use of metrics is intrinsic for achieving higher levels of CMM.

Building on the definition of software metrics given above, we can divide the subject area into two components:

1. The component concerned with defining the actual measures (in other words the numerical 'metrics')
2. The component concerned with how we collect, manage and use the measures.

Table 1, which is an adapated version of a table that appeared in [18], provides a classification of software metrics as far as the first component is concerned. This classification has now been reasonably widely adopted. Essentially any software metric is an attempt to measure or predict some attribute (internal or external) of some product, process, or resource. The table provides a feel for the broad scope of software metrics, and clarifies the distinction between the attributes we are most interested in knowing or predicting (normally the external ones) and those which ultimately are the only things we can control and measure directly (the internal ones).

Although there are literally thousands of metrics that have been proposed since the mid 1960's (all of which fit into the framework of Table 1) the rationale for almost all individual metrics has been motivated by one of two activities:

1. The desire to assess or predict effort/cost of development processes;
2. The desire to assess or predict quality of software

ENTITIES	ATTRIBUTES	
	Internal	***External***
Products		
Specifications	size, reuse, modularity, redundancy, functionality, syntactic correctness, ...	comprehensibility, maintainability, ...
Designs	size, reuse, modularity, coupling, cohesiveness, inheritance, functionality, ...	quality, complexity, maintainability, ...
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness, ...	reliability, usability, maintainability, reusability
Test data	size, coverage level, ...	quality, reusability, …
...	...	...
Processes		
Constructing specification	time, effort, number of requirements changes, ...	quality, cost, stability, ...
Detailed design	time, effort, number of specification faults found, ...	cost, cost-effectiveness, ...
Testing	time, effort, number of coding faults found, ...	cost, cost-effectiveness, stability, ...
...	...	...
Resources		
Personnel	age, price, ...	productivity, experience, intelligence, ...
Teams	size, communication level, structuredness, ...	productivity, quality, ...
Organisations	size, ISO Certification, CMM level	Maturity, profitability, …
Software	price, size, ...	usability, reliability, ...
Hardware	price, speed, memory size, ...	reliability, ...
Offices	size, temperature, light, ...	comfort, quality, ...
...	...	...

Table 1: Classification of software measurement activities (measurement can be either *assessment* or *prediction*)

products.

The key in both cases has been the assumption that product 'size' measures should drive any predictive models.

The first key metric used to do this was the Lines of Code (LOC or KLOC for thousands of lines of code) metric. It was, and still is, used routinely as the basis for measuring programmer productivity (LOC per programmer month) and as such LOC was assumed to be a key driver for the effort and cost of developing software. Indeed, the early resource prediction models (such as those of Putnam [43] and Boehm [11]) used LOC or related metrics like *delivered source instructions* as the key size variable in predictive (regression-based) models of the form

$$\text{Effort}=f(\text{LOC})$$

In the late 1960's LOC was also used as the basis for measuring program quality (normally measured indirectly as *defects per KLOC*). In 1971 Akiyama [4] published what we believe was the first attempt to use metrics for software quality *prediction* when he proposed a regression-based model for module defect density (number of defects per KLOC) in terms of the module size measured in KLOC.

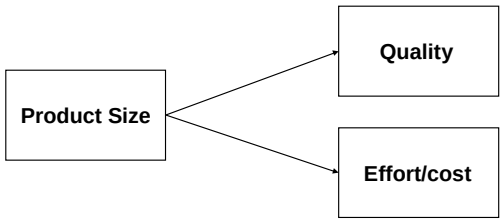

Figure 3: The historical driving factors for software metrics

In both of the key applications, therefore, LOC was being used as a surrogate measure of different notions of product *size* (including *effort*, *functionality*, or *complexity*) - the critical assumption was of size as the driver for both quality and effort, as shown in Figure 3.

The obvious drawbacks of using such a crude measure as LOC as a surrogate measure for such different notions of program size were recognised in the mid-1970's. The need for more discriminating measures became especially urgent with the increasing diversity of programming languages. After all, a LOC in an assembly language is not comparable in effort, functionality, or complexity to a LOC in a high-level language. Thus, the decade starting from the mid-1970's saw an explosion of interest in measures of software complexity (pioneered by the likes of Halstead [30] and [40]) and measures of functional size (such as function points pioneered by Albrecht [5] and later by [48]) which were intended to be independent of programming language. However, the rationale for all such metrics was to use them for either of the two key applications.

Work on extending, validating and refining complexity metrics and functional metrics has been a dominant feature of academic metrics research up to the present day. This works includes defining metrics that are relevant to new paradigms such as object oriented languages [14, 31]. The subject area has also benefited from work that has established some theoretical underpinnings for software metrics. This work (exemplified by [12,18,52]) 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.

While software metrics as a subject has been dominated by component 1 (defining specific measures and models) much recent work has been concerned with component 2 (collecting, managing, and using metrics in practice). The most notable advances in this area have been:

- *Work on the mechanics of implementing metrics programs*. Two pieces of work stand out in this respect:
  1. *The work of Grady and Caswell [27] (later extended in [28]*) 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*) [6]. 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 [29]).
- *The use of metrics in empirical software engineering*: specifically we refer to empirical work concerned with benchmarking [36] and for 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 [8,9]). Recent work [34] suggests that some momentum is growing.

3 THE TRADITIONAL APPROACHES: WEAKNESSES AND NEED FOR EXTENSION

Despite the advances described in Section 2, the approaches to both quality prediction and resource prediction have remained fundamentally unchanged since the early 1980's. Figure 2 provides a schematic view of the approach which is still widely adopted The assumption is that some measure

of size/complexity is the key driver and that variations, which are determined by regression approaches, are explained by other measurable variables. For example, in the original COCOMO model 15 additional process, product and resource attributes are considered as explaining variation in the basic regression model.

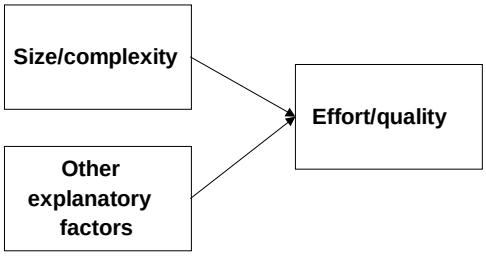

Figure 4: The basic approach to both effort prediction and quality prediction

These approaches have provided some extremely valuable empirical results. However, as discussed in Section 1, they cannot be used effectively for quantitative management and risk analysis, and hence do not yet address the primary objective of metrics.

In the case of quality prediction (see for example, [7,24,39,46]) the emphasis has been on determining regression-based models of the form

f(complexity metric) = defect density

where defect density is defects per KLOC.

In [20] we provided an extensive critique of this approach. 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. A further major empirical study [22] sheds considerable light on why the above approach is fundamentally insufficient. For example, we showed:

- Size based metrics, while *correlated* to gross number of defects, are inherently poor *predictors* of defects.
- Static complexity metrics, such as cyclomatic complexity, are not significantly better as predictors (and in any case are very strongly correlated to size metrics).
- There is a fundamental problem in the way defect counts are used as a surrogate measure of quality. Specifically, counts of defects pre-release (the most common approach) is a very bad indicator of quality.

The latter problem is the most devastating of all for the classical approach to defect modelling. To explain and motivate this result it is useful to think of the following analogy:

> Suppose you know that a person has eaten a very large lunch at 1.00pm. Is it likely that this person will also eat a very large dinner at 6.00pm?

On the one hand you might reason that a person who eats a very large lunch is inherently a 'big eater' or 'greedy' and hence that such a person is predisposed to eating a large dinner. On the other hand it seems more sensible to assume that the person will be so full up from the very large lunch that the last thing they will want to do is eat a very large dinner.

So now consider the following:

> Suppose you know that a large number of defects are found in a software module prior to release. Is it likely that this module will reveal many defects post-release?

A 'yes' answer here is no more reasonable than a 'yes' answer for a person likely to eat a large dinner after a large lunch. Yet, it is popularly believed that modules with higher incidence of faults in pre-release testing are likely to have higher incidence of faults in post-release operation. In fact, empirical evidence shows this to be an invalid hypothesis. Figure 5 shows the results of plotting pre-release faults against post-release faults for a randomly selected set of modules in the systems described in [23]. There is strong evidence that modules that are very fault-prone pre-release are likely to reveal very few faults post-release. Conversely, the truly 'fault-prone' modules post-release are the ones that revealed no faults pre-release. Why are these results devastating for classical fault prediction models? Because many of those models were 'validated' on the basis of using pre-release fault counts as a surrogate measure for operational quality.

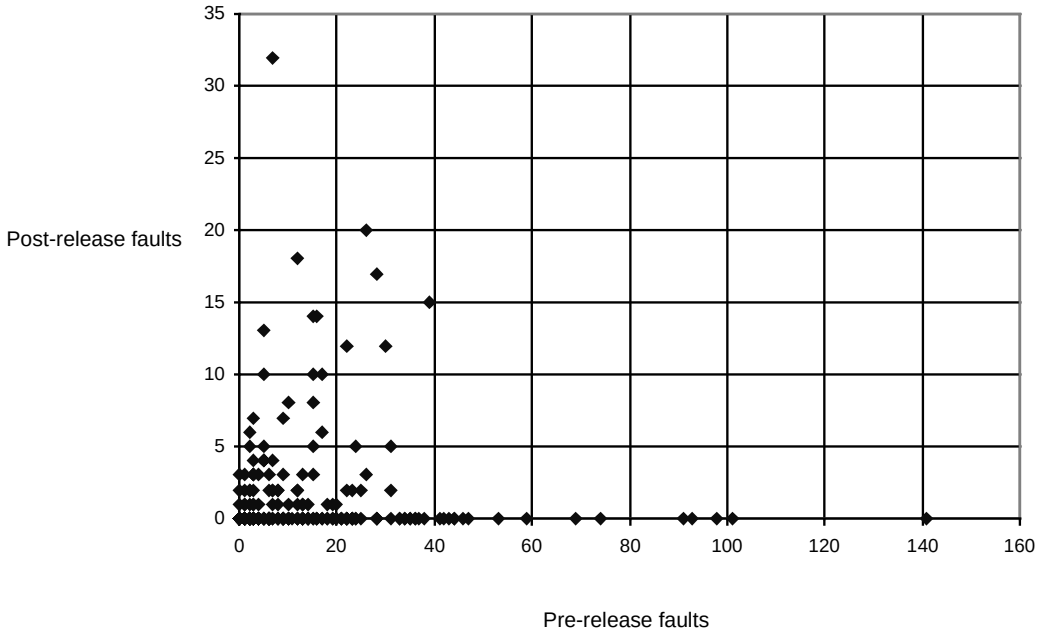

Figure 5 Scatter plot of pre-release faults against post-release faults for a major system (each dot represents a module)

The result is also devastating for much software complexity metrics work. The rationale behind much of this work has been the assumption of 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. For example, Munson and Khosghoftaar, [41] asserted:

> 'There is a clear intuitive basis for believing that complex programs have more faults in them than simple programs'

One of the key aims 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 software quality.

These remarkable results are also closely related to the empirical phenomenon observed by Adams [2] 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.

There are, of course, very simple explanations for the phenomenon observed in Figure 5. Most of the modules that had high number of pre-release, low number of post-release faults just happened to be very well tested. The amount of testing is therefore a very simple explanatory factor that must be incorporated into any predictive model of defects. Similarly, a module that is simply never executed in operation, will reveal no faults no matter how many are latent. Hence, operational usage is another obvious explanatory factor that must be incorporated.

The absence of any *causal* factors to explain variation is a feature also of the classic regression-based approach to resource prediction. Such models fail to incorporate any true causal relationships, relying often on the fundamentally flawed assumption that somehow the solution size can influence the amount of resources required. This contradicts the economic definition of a production model where output = f(input) rather input = f(output). Additional problems with these kind of models are:

- They are based on limited historical data of projects that just happened to have been completed. Based on typical software projects these are likely to have produced products of poor quality. It is therefore difficult to interpret what a figure for effort prediction based on such a model actually means.
- Projects normally have prior resourcing constraints. These cannot be accommodated in the models. Hence the 'prediction' is premised on impossible assumptions

and provides little more than a negotiating tool.

- They are essentially black box models that hide crucial assumptions from potential users.
- They cannot handle *uncertainty* either in the model inputs or the outputs. This is crucial if you want to use such models for risk assessment. We know the models are inaccurate so it is not clear what a prediction means if you have no feel for the uncertainty in the result. And there is no reason why a highly uncertain input should be treated as if it were certain. This has been recognised by pioneers of cost modelling such as Boehm and Puttnam who have both recently been attempting to incorporate uncertainty into their models [15,44].

So the classical models provide little support for risk assessment and reduction

For all the good work done in software metrics, it provides only a starting point when it comes to assessing real systems, especially the critical systems that were important to many of our industrial collaborators. The classic assessment problem we were confronted with was "Is this system sufficiently reliable to ship?"

The kind of information you might have available or would want to use in arriving at a decision is:

- measurement data from testing, such as information about defects found in various testing phases, and possibly even failure data from simulated operational testing.
- empirical data about the process and resources used, e.g. the reliability of previous products by this team
- subjective information about the process/resources - the quality and experience of the staff etc
- very specific and important pieces of evidence such as the existence of a trustable proof of correctness of a critical component.

The problem is that even when you do have this kind of information it is almost certainly the case that none alone is going to be sufficient for most systems. This is especially true for systems with high reliability targets like 10^{-9} probability of failure on demand. So in practice the situation is that you may have fragments of very diverse information.

The question is how to combine such diverse information and then how to use it to help solve a decision problem that involves risk. One way or another decisions *are* made and they inevitably involve expert judgement. If that expert judgement is *good* we should be incorporating it into our assessment models. If it is not then our models should be able to expose its weaknesses.

In the next section we will show that causal models, using Bayesian nets can provide relevant predictions, as well as incorporating the inevitable uncertainty, reliance on expert judgement, and incomplete information that are pervasive in software engineering.

4 CAUSAL MODELS

The great challenge for the software metrics community is to produce models of the software development and testing process which take account of the crucial concepts missing from classical regression-based approaches. Specifically we need models that can handle:

- diverse process and product variables;
- empirical evidence *and* expert judgement;
- 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 [19] 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 [1], but this did not attempt to model the crucial notion of uncertainty.

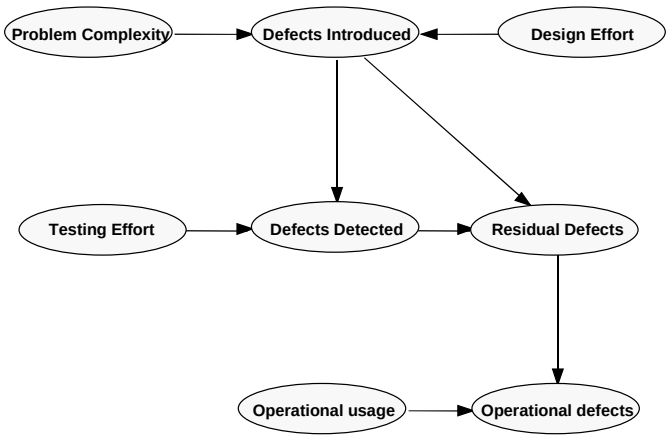

Figure 6 Defects BBN (simplified)

A BBN is a graphical network (such as that shown in Figure 6) 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.

Although the underlying theory (Bayesian probability) has been around for a long time, building and executing realistic BBN models has only been made possible because of recent algorithms (see [35]) and software tools that implement them [32]. 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.

CSR, in a number of research and development projects since 1992, has pioneered the use of BBNs in the broad area of assessing dependability of systems. For example, in recent collaborative projects we have used BBNs to:

- provide safety or reliability arguments for critical computer systems (the DATUM, SHIP, DeVa and SERENE projects have all addressed this problem from different industrial perspectives;
- provide improved reliability predictions of prototype military vehicles (the TRACS project, [3]);
- predict general software quality attributes such as defect-density and cost (the IMPRESS project [21]).

In consultancy projects [3] we have used BBNs to:

- assess safety of PES components in the railway industry;
- provide predictions of insurance risk and operational risk;
- predict defect counts for software modules in consumer electronics products.

The BBN in Figure 6 is a very simplified version of the last example. Like all BBNs the probability tables were built using a mixture of empirical data and expert 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 [42]. Thus, we feel we have to a certain 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 combinations of probability values. It is beyond the scope of this paper to describe either BBNs or the particular example in detail (see [21] for a fuller account). However, we can give a feel for their power and relevance.

Like any BBN, the model in Figure 6 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 introduced' will never be known with certainty.

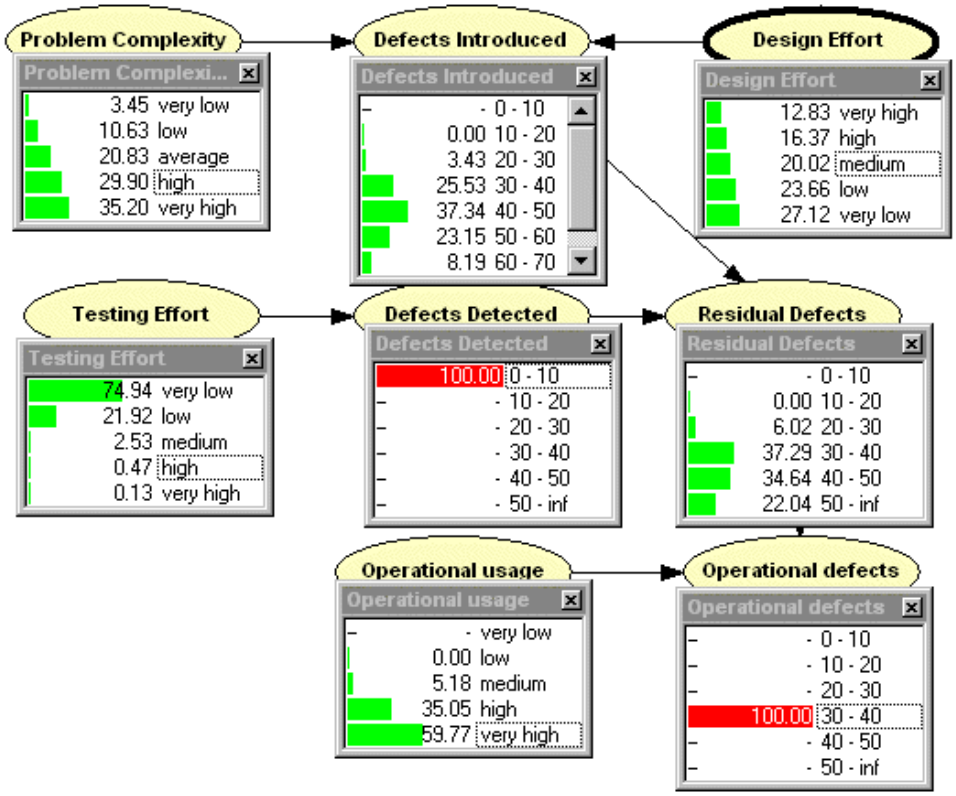

Figure 7 State of BBN probabilities showing a scenario of many post-release faults and few pre-release faults

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

Thus, Figure 7 explores the common empirical scenario (highlighted in Section 2) of a module with few pre-release defects (less than 10) and many post-release (between 30 and 40). Having entered this evidence, the remaining probability distributions are updated. The result *explains* this scenario by showing that it was very likely that a 'very low' amount of testing was done, while the operational usage is likely to be 'very high'. The problem complexity is also more likely to be high than low.

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

The BBN approach directly addresses the weaknesses described in Section 3 of the traditional approach to defect modelling and prediction. In [21] (which also contains more extensive examples of the above BBN) we have described in detail a BBN approach that directly addresses the weaknesses of the traditional approach to resource modelling and prediction. Again, it is beyond the scope of this paper to present the approach in any detail. However, conceptually the model is quite simple and is shown in Figure 8.

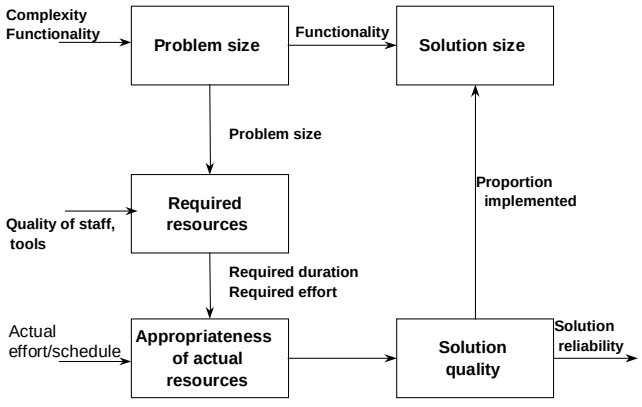

Figure 8 Causal model for software resources

Each of the boxes in Figure 8 represent subnets. For example, there is a subnet concerned with *problem size*, which contains variables such as *functionality* and *complexity*. *Problem size* influences both *solution size* and *required resources*. The subnets that make this kind of model so different from the classical approach are the subnets *appropriateness of actual resources* and *solution quality*. The basic idea is that the *required resources* for a given problem are always some kind of an ideal based on best practice. What matters are how well these ideal resources match the actual resources available. This match (which is calculated in the *appropriateness of actual resources* subnet) determines the solution quality.

The crucial thing about the resulting BBN [21] is that is can be used to answer all of the types of problems listed in Section 1—the very types of problem that cannot be handled by traditional approaches.

5 A WAY FORWARD FOR SOFTWARE METRICS

We have argued that traditional approaches to software metrics fail to address the key objective of providing quantitative support for management *decision making*. In the previous section we showed that when simple metrics are used in the context of a causal model such as a BBN, you *do* get real support for decision-making and risk assessment. However, for comprehensive decision analysis in many contexts BBNs must be supplemented with other approaches. For example, suppose you have to assess whether a new software protection system should be deployed in a nuclear plant. A BBN might help you come up with a more accurate and justifiable prediction of reliability than otherwise, because it will be able to combine many different strands of evidence (such as test results and process information). Such a BBN could even enable you to examine various trade-offs and impact of different risk factors on reliability. However, such a BBN will not be sufficient to make your decision about whether or not to deploy the system. For that you have to consider political, financial, environmental and technical criteria with preferences that are not easily modelled in a BBN. There is, in fact a large body of work on Multi-Criteria Decision Aid (MCDA) [50,51] which deals with this problem. This work has been largely ignored by the software engineering community. We feel that there is great potential in combining causal models such as BBNs with preference models such as those found in MCDA to provide a more complete solution to the quantitative decision analysis problem in software engineering.

Progress in building good decision support systems along the lines proposed above, is intrinsically dependent on one other great challenge for the software metrics community in the next 10 years. This is to extend the emerging discipline *of empirical software engineering*. All the BBN models that we have produced in the context of software risk assessment, have been ultimately dependent on results from empirical studies, including some of the important emerging benchmarking studies. We would like to see an increase in empirical studies that test specific cause and effect hypotheses and establish ranges of empirical values.

For software managers seeking decision-support for quantitative risk management, tools based on industry-wide empirical studies and benchmarking data can certainly

provide a vast improvement than on using gut instinct and heuristics. However, the history of software metrics teaches us conclusively that, for greater accuracy and relevance, there is a need for company-specific data input. Hence there will continue to be a need for company specific metrics programs that address specific objectives. The research challenge is *not* to produce new metrics in this respect, but to build on the work of Basili, Grady and the likes on the managerial challenges that must be overcome in setting up such programs. The required metrics are, in fact, now reasonably well understood and accepted. Most objectives can be met with a very simple set of metrics, many of which should in any case be available as part of a good configuration management system. This includes notably: information about faults, failures and changes discovered at different life-cycle phases; traceability of these to specific system 'modules' at an appropriate level of granularity; and 'census' information about such modules (size, effort to code/test). It is also very useful if the hard metrics data can be supplemented by subjective judgements, such as resource quality, that would be available from project managers.

The final challenge is that of *technology transfer*. If metrics is ultimately a tool for quantitative managerial decision-making, then it must be made available to software managers in a form that is easily understood by them. Managers are no more likely to want to experiment with BBNs as they would with regression models or code complexity metrics. The challenge for all metrics research is to bring the results of the research to the stage whereby it is usable by managers without them having to understand any of the underlying theory. In the case of BBNs, for example, managers need only understand the basic causal model for their application domain. To achieve truly widescale adoption of the technology in software management it will be necessary to provide simple, configurable questionnaire-based front-ends that produce simple reports and recommendations as outputs.

6 CONCLUSIONS

Software metrics research and practice has helped to build up an empirical basis for software engineering. This is an important achievement. But the classical statistical-based approaches do *not* provide managers with decision support for risk assessment and hence do not satisfy one of the most important objectives of software metrics. The great challenge for the next ten years is to use software metrics in such a way that they address this key objective.

We feel that the way forward is to build and use causal models based on existing simple metrics. We have highlighted one such approach, namely BBNs, which have many advantages over the classical approaches. Specifically, the benefits of using BBNs include:

- explicit modelling of 'ignorance' and uncertainty in estimates, as well as cause-effect relationships
- enables us to combine diverse types of information
- 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

The BBN models can be thought of as risk management decision-support tools that build on the relatively simple metrics that 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. 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.

Of course BBNs alone cannot solve all the quantitative managerial decision problems that should be addressed by software metrics. Hence we have proposed an approach that involves combining BBNs with MCDA and empirical software engineering.

Clearly the ability to use these methods for accurate prediction and risk assessment still depends on some stability and maturity of the development processes. Organisations that do not collect the basic metrics data, or do not follow defined life-cycles, will not be able to apply such models effectively.

ACKNOWLEDGMENTS

The work was supported, in part, by the EPSRC-funded project IMPRESS.

7 REFERENCES

[1] Abdel-Hamid TK, "The slippery path to productivity improvement", IEEE Software, 13(4), 43-52, 1996.

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

[3] Agena Ltd, "Bayesian Belief Nets", http://www.agena.co.uk/bbn_article/bbns.html

[4] Akiyama F, "An example of software system debugging", Inf

Processing 71, 353-379, 1971.

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

[6] Basili VR and Rombach HD, The TAME project: Towards improvement-oriented software environments, IEEE Transactions on Software Engineering 14(6), pp 758-773, 1988.

[7] Basili VR and Perricone BT, "Software Errors and Complexity: An Empirical Investigation", Communications of the ACM, Vol. 27, No.1, pp.42-52, 1984.

[8] Basili VR and Reiter RW, A controlled experiment quantitatively comparing software development approaches, IEEE Trans Soft Eng SE-7(3), 1981.

[9] Basili VR, Selby RW and Hutchens DH, Experimentation in software engineering, IEEE Trans Soft Eng SE-12(7), 733-743, 1986.

[10] Bhargava HK, Sridhar S, Herrick C, Beyond spreadsheets: tools for building decision support systems, IEEE Computer, 32(3), 31-39, 1999.

[11] Boehm BW, Software Engineering Economics, Prentice-Hall, New York, 1981.

[12] Briand LC, Morasca S, Basili VR, Property-based software engineering measurement, IEEE Transactions on Software Engineering, 22(1), 68-85, 1996.

[13] Brocklehurst, S and Littlewood B, New ways to get accurate software reliability modelling, IEEE Software, 34-42, July, 1992.

[14] Chidamber SR and Kemerer CF, A metrics suite for object oriented design, IEEE Trans Software Eng, 20 (6), 476-498, 1994.

[15] Chulani S, Boehm B, Steece B, "Bayesian analysis of empirical software engineering cost models", IEEE Transactions on Software Engineering, 25(4), 573-583, 1999.

[16] DeMarco T, 'Controlling Software Projects', Yourden Press, New York, 1982

[17] Fenton NE, "Software Measurement Programs", Software Testing & Quality Engineering 1(3), 40-46, 1999.

[18] Fenton NE, Software Metrics: A Rigorous Approach, Chapman and Hall, 1991.

[19] Fenton NE, Littlewood B, Neil M, Strigini L, Sutcliffe A, Wright D, Assessing Dependability of Safety Critical Systems using Diverse Evidence, IEE Proceedings Software Engineering, 145(1), 35-39, 1998.

[20] Fenton NE and Neil M, A Critique of Software Defect Prediction Models, IEEE Transactions on Software Engineering, 25(5), 675-689, 1999.

[21] Fenton NE and Neil M, "Software Metrics and Risk", Proc 2nd European Software Measurement Conference (FESMA'99), TI-KVIV, Amsterdam,ISBN 90-76019-07-X, 39-55, 1999.

[22] Fenton NE and Ohlsson N, Quantitative Analysis of Faults and Failures in a Complex Software System, IEEE Transactions on Software Engineering, to appear, 2000.

[23] Fenton NE and Pfleeger SL, Software Metrics: A Rigorous and Practical Approach (2nd Edition), International Thomson Computer Press, 1996.

[24] Gaffney JR, "Estimating the Number of Faults in Code", IEEE Trans. Software Eng., Vol.10, No.4, 1984.

[25] Gilb T, Software Metrics, Chartwell-Bratt, 1976.

[26] Glass RL, "A tabulation of topics where software practice leads software theory", J Systems Software, 25, 219-222, 1994.

[27] Grady R and Caswell D, Software Metrics: Establishing a Company-wide Program, Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[28] Grady, R.B, Practical Software Metrics for Project Management and Process Improvement, Prentice-Hall, 1992.

[29] Hall T and Fenton NE, "Implementing effective software metrics programmes", IEEE Software, 14(2), 55-66, 1997.

[30] Halstead M, Elements of Software Science, North Holland, , 1977.

[31] Harrison R, Counsell SJ, Nithi RV, "An evaluation of he MOOD set of object-oriented software metrics", IEEE Transactions on Software Engineering, 24(6). 491-496, 1998.

[32] Hugin A/S: www.hugin.com, Hugin Expert A/S, P.O. Box 8201 DK-9220 Aalborg, Denmark, 1999

[33] Humphrey WS, Managing the Software Process, Addison-Wesley, Reading, Massachusetts, 1989.

[34] Jeffery DR and Lotta LG, "Empirical Software Engineering: Guest editors special section introduction", IEEE Transactions on Software Engineering, 25(4), 435-437, 1999.

[35] Jensen FV, An Introduction to Bayesian Networks, UCL Press, 1996.

[36] Jones C, "Applied Software Measurement", McGraw Hill, 1991.

[37] Kitchenham B, Pickard L, Pfleeger SL, "Case studies for method and tool evaluation", IEEE Software, 12(3), 52-62, July, 1995.

[38] Lauritzen SL and Spiegelhalter DJ, "Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems (with discussion)", J. R. Statis. Soc. B, 50, No 2, pp 157-224, 1988.

[39] Lipow M, "Number of Faults per Line of Code", IEEE Trans. Software Eng., Vol. 8, No.4, 437-439, 1982.

[40] McCabe T, A Software Complexity Measure, IEEE Trans. Software Engineering SE-2(4), 308-320, 1976.

[41] Munson JC, and Khoshgoftaar TM, The detection of fault-prone programs, IEEE Transactions on Software Engineering,18(5), 423-433, 1992.

[42] Neil M, Fenton NE, Nielsen L, "Building large-scale Bayesian Networks", submitted Knowledge Engineering Review, 1999.

[43] Putnam LH, A general empirical solution to the macro software sizing and estimating problem, IEEE Trans Soft Eng SE-4(4), 1978, 345-361, 1978.

[44] Putnam LH, Mah M, Myers W, "First, get the front end right", Cutter IT Journal 12(4), 20-28, 1999.

[45] SERENE consortium, SERENE (SafEty and Risk Evaluation using bayesian Nets): Method Manual, ESPRIT Project 22187, http://www.csr.city.ac.uk/people/norman.fenton/serene.htm, 1999.

[46] Shen VY, Yu T, Thebaut SM, and Paulsen LR, "Identifying error-prone software - an empirical study", IEEE Trans. Software Eng., Vol. 11, No.4, pp. 317-323, 1985.

[47] Stark G, Durst RC and Vowell CW, Using metrics in management decision making, IEEE Computer, 42-49, Sept, 1994.

[48] Symons, CR, Software Sizing & Estimating: Mark II Function point Analysis, John Wiley, 1991.

[49] TRACS (Transport Reliability Assessment & Calculation System): Overview, DERA project E20262, http://www.agena.co.uk/tracs/index.html, 1999

[50] Vincke P, Multicriteria Decision Aid, J Wiley, New York, 1992.

[51] Yang JB and Singh MG, An evidential reasoning approach for multiple attribute decision making with uncertainty, IEEE Transactions Systems, Man, and Cybernetics, 24, 1-18, 1994.

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