Available online at www.sciencedirect.com

ScienceDirect

Information and Software Technology 49 (2007) 32–43

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Predicting software defects in varying development lifecycles using Bayesian nets

Norman Fenton [a,b,*], Martin Neil [a,b], William Marsh [a], Peter Hearty [a], David Marquez [a], Paul Krause [c,d], Rajat Mishra [d]

[a] *Department of Computer Science Queen Mary, University of London, Mile End Road, London, UK*
[b] *Agena Ltd., London, UK*
[c] *Department of Computing, University of Surrey, Guildford, Surrey, UK*
[d] *Philips Software Centre, Bangalore, India*

Available online 16 October 2006

Abstract

An important decision in software projects is when to stop testing. Decision support tools for this have been built using causal models represented by Bayesian Networks (BNs), incorporating empirical data and expert judgement. Previously, this required a custom BN for each development lifecycle. We describe a more general approach that allows causal models to be applied to any lifecycle. The approach evolved through collaborative projects and captures significant commercial input. For projects within the range of the models, defect predictions are very accurate. This approach enables decision-makers to reason in a way that is not possible with regression-based models.

Keywords: Causal models; Dynamic Bayesian networks; Software defects; Decision support

1. Introduction

In [8] we reviewed the various approaches to software defect prediction. We concluded that traditional statistical approaches, such as using regression modelling alone, were inadequate. We proposed that ***causal*** models were needed for more accurate predictions. We described a simple Bayesian Net (BN) as an example of the kind of causal model needed for this purpose. Since then a number of authors, for example [3,7,24], have used BNs in various related aspects of software engineering management. Our own research has extended the use of BNs to more general software project management [13]. However, the original motivation of more accurate software defect prediction has continued to be an important focus of our research and this paper describes some of the most recent results in this area.

Following on from the early ideas presented in [8], in [12] we have shown how BNs can be used to predict the number of software defects remaining undetected after testing. This work led to the AID tool [22] developed in partnership with Philips, and used to predict software defects in consumer electronic products. Project managers use a BN-based tool such as AID to help decide when to stop testing and release software, trading-off the time for additional testing against the likely benefit.

Rather than relying only on data from previous projects, this work uses causal models of the Project Manager's understanding, covering mechanisms such as:

- poor quality development increases the number of defects likely to be present,
- high quality testing increases the proportion of defects found.

[*] Corresponding author. Tel.: +44 20 7882 7896; fax: +44 20 7882 7860. *E-mail address:* norman@dcs.qmul.ac.uk (N. Fenton).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.09.001

Causal models are important because they allow all the evidence to be taken into account, even when different evidence conflicts. Suppose that few defects are found during testing – does this mean that testing is poor or that development was outstanding and the software has few defects to find? Regression-based models of software defects are little help to a Project Manager who must decide between these alternatives [8]. BN models allow back propagation of evidence to show the most likely causes of a given outcome. Data from previous projects are used to build the BN, with expert judgements on the strength of each causal mechanism.

In this paper, we extend the earlier work by describing a much more flexible and general method of using BNs for defect prediction. We also describe how the AgenaRisk [1] toolset is used to create an effective decision support system from the BN. An important limitation of the earlier work was the need to build a different BN for each software development lifecycle – to reflect both the differing number of testing stages in the lifecycle and the differing metrics data available. Given the work required to build a BN, this severely limits the practicality of the approach. To overcome this limitation, we describe a BN that models the creation and detection of software defects without commitment to a particular development lifecycle. We then show how a software development organisation can adapt this BN to their development lifecycle and metrics data with much less effort than is needed to build a tailored BN from scratch.

The contents of the remainder of the paper are as follows: in Section 2 we introduce BNs and show how they are used for causal modelling in software engineering. Section 3 introduces the idea of a 'phase' as a sub-part of a software lifecycle and shows how several phase models can be combined to model different lifecycles. The phase model is described in detail in Section 4; Section 5 shows how it is adapted to different development lifecycles. An experimental validation of defect predictions is described in Section 6.

2. Defect prediction with BNs

2.1. Bayesian nets

A Bayesian net [14] (BN) is a directed acyclic graph (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. We have adopted the convention in this paper that a dotted margin around a node indicates that it is a "link" node. The semantics of these nodes will be discussed in Section 3.2.

The BN of Fig. 1 forms a causal model of the process of inserting, finding and fixing software defects. The variable 'effective KLOC implemented' represents the complexity-adjusted size of the functionality implemented: as the amount of functionality increases the number of potential defects rises. In this version of the model, KLOC is used as a surrogate for Function Points (FP). Function points are the preferred measure of program size since they can be estimated on the basis of a functional specification. However, most of the companies involved in the validation of this model did not use FPs and, since the validation was retrospective, KLOC measures were readily available. The model can deal with either KLOC or FPs using appropriate conversions (based on empirical data by Jones [15,16]).

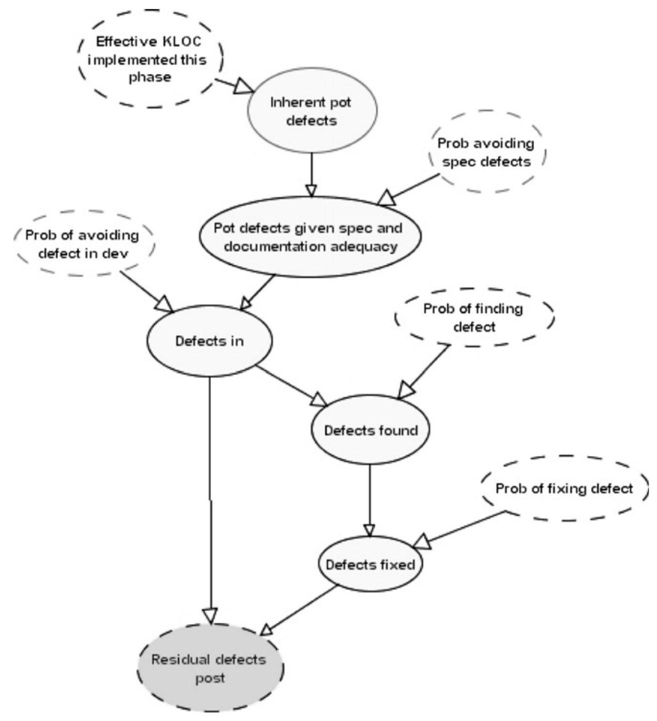
Effective KLOC implemented this phase
Inherent pot defects
Prob avoiding spec defects
Prob of avoiding defect in dev
Pot defects given spec and documentation adequacy
Defects in
Prob of finding defect
Defects found
Prob of fixing defect
Defects fixed
Residual defects post

Fig. 1. BN for defect prediction.

The 'probability of avoiding defect in development' determines 'defects in' from 'Potential defects given specification and documentation adequacy'. This number represents the number of defects (before testing) that are in the new code that has been implemented. However, inserted defects may be found and fixed: the residual defects are those remaining after testing.

There is a probability table for each node, specifying how the probability of each state of the variable depends on the states of its parents. Some of these are deterministic (in the sense that they introduce no *new* uncertainty): for example 'Residual defects' is simply the numerical difference between 'Defects in' and 'Defects fixed'. In other cases, we can use standard statistical functions: for example the process of finding defects is modelled as a sequence of independent experiments, one for each defect present, using the 'Probability of finding a defect' as a characteristic of the testing process.

$Defects\ found = B(Defects\ inserted, Prob\ finding\ a\ defect)$

where $B(n,p)$ is the Binomial distribution for n trials with probability p.

Some nodes are defined as *ranked* nodes. These have a discrete set of states such as: "very low", "low", "medium", "high", "very high". Such nodes are useful when capturing expert judgement, where a simple, qualitative description is required. However, because this judgment is also intended to indicate degree, it is represented by an underlying real number in the range [0..1]. An example is the 'Quality of spec and doc PRE' node in Fig. 4. Its children incorporate the [0..1] value of their parent into expressions which determine their conditional probability tables. However, the correspondence between qualitative degree and quantitative value is not always straightforward. For example, the 'Quality of spec and doc POST' node in Fig. 4 uses a partitioned expression on one of its parents to create distinct conditional probability tables corresponding to each parent state.

For variables without parents the table just contains the prior probabilities of each state.

The BN represents the complete joint probability distribution – assigning a probability to each combination of states of all the variables – but in a factored form, greatly reducing the space needed. When the states of some variables are known, the joint probability distribution can be recalculated conditioned on this 'evidence' and the updated marginal probability distribution over the states of each variable can be observed.

The quality of the development and testing processes is represented in the BN of Fig. 1 by four variables over the 0 to 1 interval:

- probability of avoiding specification defects,
- probability of avoiding defects in development,
- probability of finding defects,
- probability of fixing defects.

The BN in Fig. 1 is a simplified version of the BN at the heart of a decision support system for software defects, discussed below. None of these probability variables (or the 'Effective KLOC implemented' variable) are entered directly by the user: instead, these variables have further parents modelling the causes of process quality as we describe in Section 4.

2.2. Decision support with BNs

Although the underlying theory (Bayesian probability) has been around for a long time, executing realistic BN models was only first made possible in the late 1980s as a result of breakthrough algorithms and software tools that implement them [14]. Methods for building large-scale BNs are even more recent [9,21] but it is only such work that has made it possible to apply BNs to the problems of software engineering.

Drawing on this work in various commercial projects with Agena, Fenton and Neil have built BN-based applications that have proved the technology is both viable and effective. Several of these applications have been related to systems or software assessment. Especially significant was the TRACS tool [20] to assess vehicle reliability for QinetiQ (on behalf of the UK Ministry of Defence) and the AID tool [11,22] to predict software defects in consumer electronic products for Philips. Much of the modelling work described here was done as part of the MODIST project [13], which extends the ideas in AID. The toolset implementation has been based on Agena's AgenaRisk technology that was extended to incorporate recent developments in building large-scale BNs that was undertaken in the SCULLY, SIMP and SCORE projects [9].

Three features of AgenaRisk are especially critical for building this kind of model:

- Continuous nodes do not have to be discretised manually. Part of the problem in creating BN models is determining the discretisation intervals for numeric nodes. After the MODIST project, the AgenaRisk toolset was updated to automatically discretise – allocating more refined intervals to greater probability masses.
- The notion of 'ranked nodes' with a range of pre-defined functions makes it easy for domain experts to build very large tables that otherwise would have to be constructed manually.
- Probability tables are generated from numerical and statistical expressions by simulation. The expression given above using the binomial distribution is not only the conceptual model but also how the model is specified. A user specifies that a node has an NPT that has a binomial distribution dependent upon its parents. The simulation algorithm then converges upon a suitably discretised approximation to the true continuous distribution [23].

2.3. Building the BN model

Like all BNs, the defect model was built using a mixture of data and expert judgements. Understanding cause and effect is a basic form of human knowledge underlying our decisions. For example, a project manager knows that more rigorous testing increases the number – and proportion of – defects found during testing and therefore reduces the number remaining in the delivered software.

It is obvious that the relationship is not the other way round. However, it is equally obvious that we need to take into account whatever evidence we have about: the likely number of defects in the software following development; the capabilities of the team; and the adequacy of the time allowed. The expert's understanding of cause and effect is used to connect the variables of the net with arcs drawn from cause to effect.

To ensure that our model is consistent with these empirical findings, the probability tables in the net are constructed using data, whenever it is available. However, when there is missing data, or the data does not take account of all the causal influences, expert judgement must be used as well.

2.4. Object Oriented Bayesian nets

Creating large Bayesian nets, consisting of many repetitions of similar collections of nodes, is a straightforward but highly laborious process. Object Oriented Bayesian nets (OOBN) simplify this task by creating predefined subnets, known as *Classes*. Instances of these net classes are known as *Risk Objects*. The theory underlying this approach was presented by Koller and Pfeffer [18], and the practical implications outlined by Bangsø and Wuillemin [2]. The utility of OOBNs will be demonstrated in our approach to modelling software lifecycles.

3. Varying the lifecycle

When we describe defects being inserted in 'implementation' and removed in 'testing' we are referring to the activities that make up the software development lifecycle. We need to fit a decision support system to the lifecycle being used, but practical lifecycles vary greatly. In this section, we describe how this can be achieved without having to build a bespoke BN for every different lifecycle. The solution has two steps: the idea of a lifecycle 'phase' modelled by a BN and a method of linking separate phase models into a model for an entire lifecycle.

3.1. A lifecycle phase

We model a development lifecycle as made up from 'phases', but a phase is not a fixed development process as in the traditional waterfall lifecycle. Instead, a phase can consist of any number and combination of development activities. For example, in the 'incremental delivery' approach the phases could correspond to the code increments. Each phase then includes all the development activities: specification, design, coding and testing. Even in a traditional waterfall lifecycle it is likely that a phase includes more than one activity with, for example, the testing phase involving some new design and coding work.

The incremental and waterfall models are just two ends of a continuum. To cover all parts of this continuum, we consider all phases to include one or more of the following development *activities*:

- Specification/documentation: This covers any activity whose objective is to understand or describe some existing or proposed functionality. It includes: requirements gathering, writing, reviewing, or changing any documentation (other than comments in code).
- Development (or more simply coding): This covers any activity that starts with some predefined requirements (however vague) and ends with executable code.
- Testing and rework: This covers any activity that involves executing code in such a way that defects are found and noted; it also includes fixing known defects.

The phase BN includes all these activities, allowing the extent of each activity in any actual phase to be adjusted. In the most general case, a software project will consist of a combination of these phases. In Section 4 we describe the BN model for one phase in more detail. First, in the next section, we describe how multiple instances of the BN are linked to model an arbitrary lifecycle.

3.2. Linking phases: Dynamic BNs

Whatever the development lifecycle, the main objective is: given information about current and past phases we would like to be able to predict attributes of quality for future phases. We therefore think of the set of phases as a time series that defines the project overall. This is readily expressed as a Dynamic Bayesian Network (DBN) [2]. Multiple instances of a single risk object class are chained together, with instances representing consecutive time frames. A DBN allows time-indexed variables: in each time frame one of the parents of a time-indexed variable is the variable from the previous time frame. Fig. 2 shows how this is applied when the quality attribute is the number of residual defects.

The dynamic variable is shown with a dashed boundary. We construct the DBN with two nodes for each time-indexed variable: the value in the previous time frame is the 'input' node (here 'Residual defects pre') and it has no parents in the net. The node representing the value in this time frame is called the 'output node' (here 'Residual defects post'). Note that the variable for the current time frame 'Residual defects post' depends on the one for the previous time frame, but as an ancestor rather than as a parent since it is clearer to represent the model with the intermediate variable 'Total defects in'.

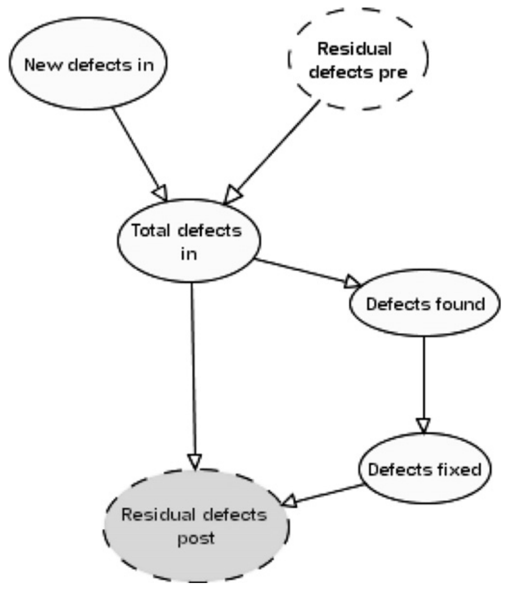

Fig. 2. A dynamic BN modelling a software lifecycle.

As well as defects, we also model the documentation quality as a time-varying quality attribute. Recall that documentation includes specification, which even in iterative developments is often prepared in one phase-level and implemented in a later phase. We consider specification errors as defects so a phase in which documentation is the main activity may lead to an important incremental change in documentation quality that is passed on to the next phase.

4. Modelling a single phase

We describe the 'phase-level BN', which models a single software development phase, first giving an overview and then describing two parts of the BN in more detail.

4.1. Overview

The phase BN is constructed from five classes.

- One of three *activity* classes: specification and documentation, design and development, or test and rework.
- The scale of "New functionality" developed in this phase.
- The defect prediction model.

Fig. 3 shows a single object instantiation of each of these classes. This object view of the single phase model represents the BN in abstract terms. The inner details of each class are not shown – only the input and output nodes are visible. In this view, a class is represented by its interface to other classes.

Triangular arrow heads represent input nodes within a class, whereas rounded arrow tails represent output nodes. Lines represent input node instantiation, i.e. the output node of one object instantiates (replaces) the input node of the connected object. Input nodes effectively act as parameters for a BN class.

Note that not all input nodes are instantiated by output nodes from another object. Input nodes have a default probability distribution associated with them. However this is rarely used. More often, unattached input nodes are initialised using explicit observations.

For example "Residual defects pre" is used to account for defects remaining from previous phases. If this is the first or only phase, then it should be explicitly initialised to zero.

The BN classes are:

- *New Functionality Implemented.* Since we are to build and test some software we may be implementing some new functionality in this phase. This class provides a measure of the size of this functionality.
- *Specification and Documentation*. This class is concerned with measuring the amount of specification and documentation work in the phase, the quality of the specification process and determining the change in the quality of the documentation as a result of the work done in the phase (modelled as a time-indexed variable).
- *Design and Development.* This class models the quality of the design and development process, which influences the probability of inserting each of the potential defects into the software.
- *Testing and Rework.* This class models the quality of the testing process and the rework process, influencing the probabilities of finding and fixing defects.

Defect Insertion and Discovery. This class follows the pattern already described in Section 2.1, adapted to handle changes to the number of defects using a time-indexed variable. The amount of 'new functionality implemented' will influence the inherent number of defects in the new code. We distinguish between potential defects from poor specification and 'inherent potential defects', which are independent of the specification.

4.2. Specification and Documentation

Fig. 4 shows the Specification and Documentation class. Before implementing any functionality there is assumed to be some specification of it. If we are lucky this specification

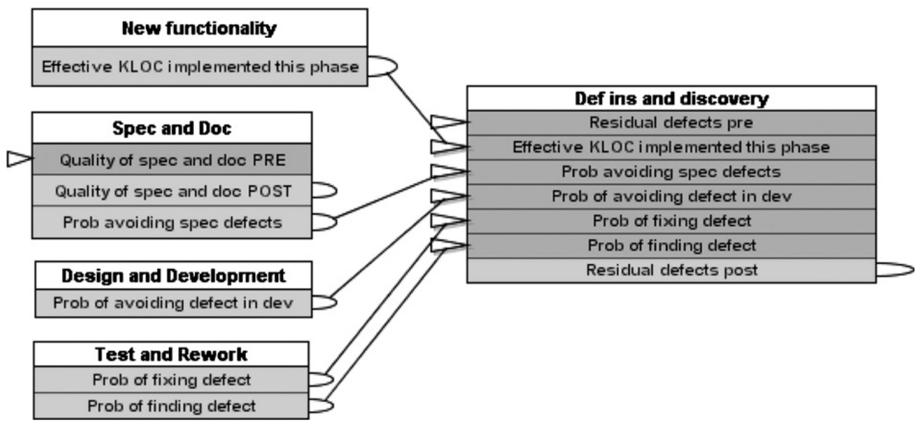
New functionality
Effective KLOC implemented this phase
Spec and Doc
Quality of spec and doc PRE
Quality of spec and doc POST
Prob avoiding spec defects
Design and Development
Prob of avoiding defect in dev
Test and Rework
Prob of fixing defect
Prob of finding defect
Def ins and discovery
Residual defects pre
Effective KLOC implemented this phase
Prob avoiding spec defects
Prob of avoiding defect in dev
Prob of fixing defect
Prob of finding defect
Residual defects post

Fig. 3. Objects in the phase BN.

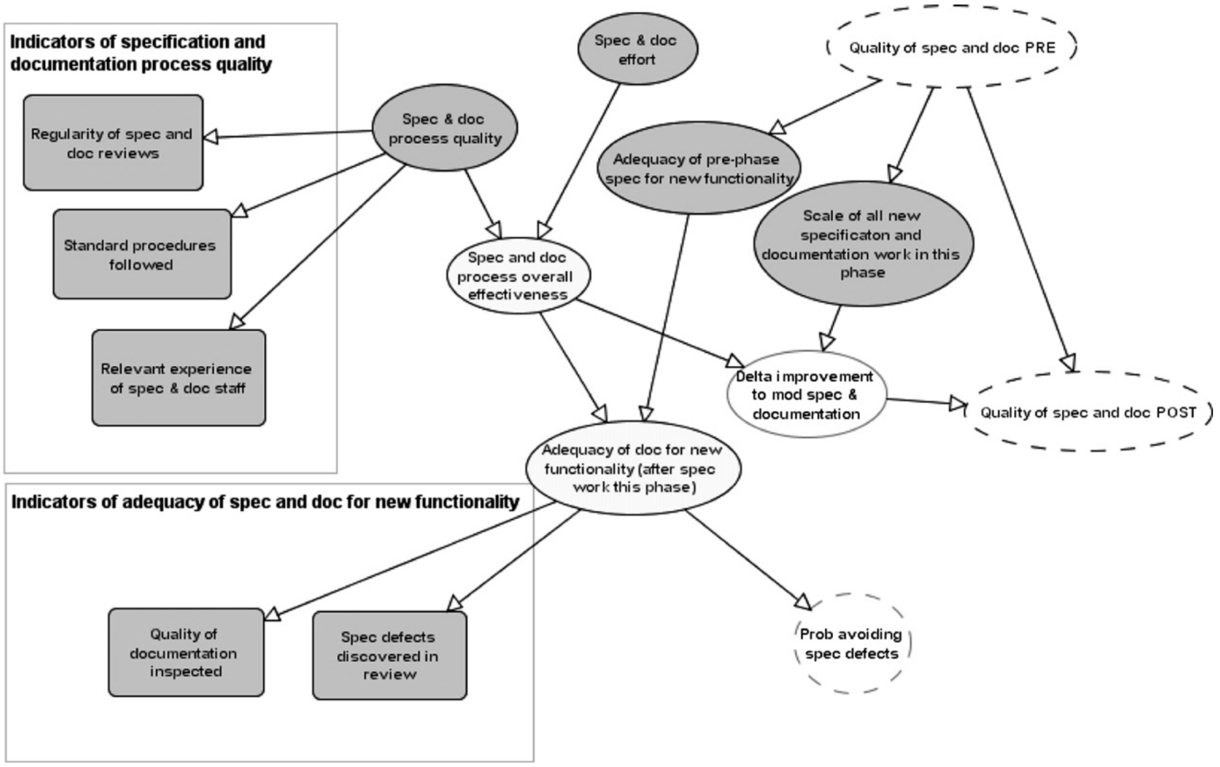
Indicators of specification and documentation process quality
Regularity of spec and doc reviews
Standard procedures followed
Relevant experience of spec & doc staff
Spec & doc process quality
Spec & doc effort
Quality of spec and doc PRE
Adequacy of pre-phase spec for new functionality
Scale of all new specificaton and documentation work in this phase
Spec and doc process overall effectiveness
Delta improvement to mod spec & documentation
Quality of spec and doc POST
Adequacy of doc for new functionality (after spec work this phase)
Indicators of adequacy of spec and doc for new functionality
Quality of documentation inspected
Spec defects discovered in review
Prob avoiding spec defects

Fig. 4. Specification and documentation class.

will be a well-written document at the appropriate level of detail. However, in many cases it may be nothing more than a vague statement of requirements. Generally, therefore, there may be work that needs to be done on the specification as part of this lifecycle phase.

The ‘scale of all new specification and documentation work in this phase’ and ‘spec & doc process quality’ will determine the ‘adequacy of documentation for new functionality (after spec work this phase)’ that is being implemented in this phase. If, for example, there is very little new functionality (and so the ‘scale of new specification and documentation work’ is low) then, even if the ‘spec & doc process quality’ is poor, it is likely that adequacy of documentation will be sufficient. On the other hand, if there is a lot of new functionality the scale of new specification and documentation work is likely to be high, which means that the process quality will need to be good in order for the documentation to be adequate.

This class shows the use of ‘indicator’ nodes: for example the experience of the staff is an indicator of the process quality. Indicators can easily be tailored to match the information available in the software development environment – see Section 5.

4.3. Testing and rework

Fig. 5 shows the testing and rework class. The better the testing process the more likely we are to find defects. We may or may not decide to fix the defects found in testing in this phase; the success of such fixes will depend on the ‘probability of fixing defect’. The two probabilities are used to update the number of residual defects in the ‘Defect Insertion and Discovery’ class and to predict the number of residual defects at the start of any subsequent phase in which further development and/or testing of the software takes place.

4.4. Variations on the phase model

We can easily construct phase models that exclude any of the development activities already described. For example, a phase that includes only specification or documentation is modelled by an instance of the “Specification and documentation” class connected to an instance of the “Defect Insertion and Discovery” class.

The new functionality implemented is set to zero, and the development, testing and rework effort to zero. This ensures that the information about defects is not changed (since without coding or testing defects are neither introduced nor removed).

However, it is irksome for users to enter dummy information to ensure that certain variables are set to zero, so we introduced a predefined set of variants of the phase BN model. These explicitly model the cases where at least one of the software development activities is not undertaken.

1. Specification/documentation and development carried out in the phase, but not testing.
2. Specification/documentation and testing carried out in the phase, but not development.

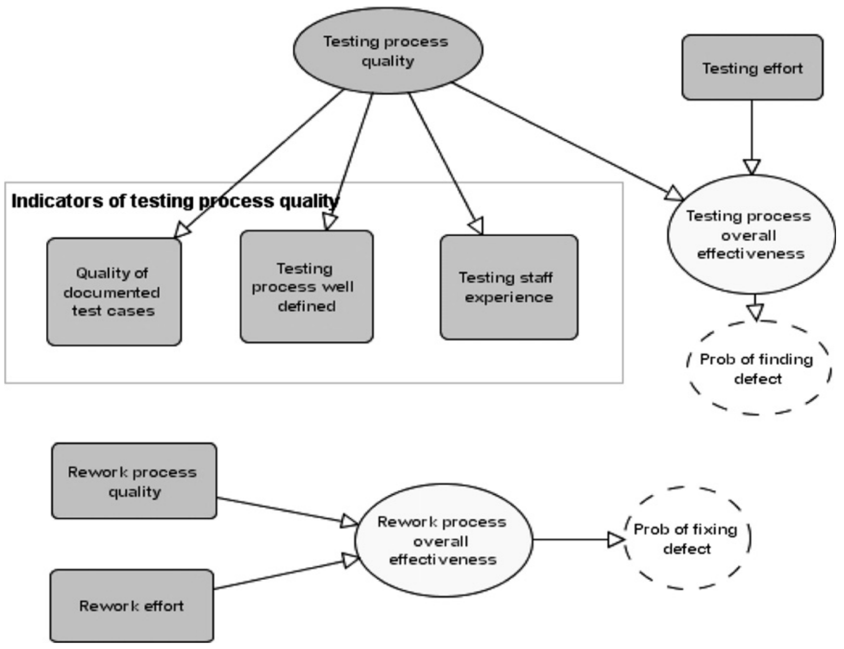
Testing process quality
Testing effort
Indicators of testing process quality
Quality of documented test cases
Testing process well defined
Testing staff experience
Testing process overall effectiveness
Prob of finding defect
Rework process quality
Rework effort
Rework process overall effectiveness
Prob of fixing defect

Fig. 5. Testing and rework class.

3. Development and testing carried out in the phase, but not specification/documentation.
4. Only specification/documentation carried out in the phase.
5. Only development carried out in the phase, and
6. Only testing carried out in the phase.

These BNs are constructed by selecting the relevant classes and omitting those that are irrelevant. The BN modelling the general case is known as the 'all activities' phase BN.

5. Application methodology

There are two steps for applying the defect prediction model to a specific software development environment.

1. Choose the 'indicators' used to judge the qualities of the different processes.
2. Link together phase BNs to model the full lifecycle.

5.1. Quality indicators

Indicator variables used in the BN can be customised to match the indicators used within the organisation. As well as editing names given to an indicator in the questionnaire, its probability table can be adjusted. The formula language of the AgenaRisk toolset makes this feasible. Consider, for example, the 'Testing process quality' (TPQ) shown in Fig. 5. The suggested indicators are:

- Quality of documented test cases.
- Testing process well defined.
- Testing staff experienced.

The process quality and the indicator values are judged on a five-point scale from 'very low' to 'very high', corresponding to an underlying 0..1 numeric range. Values are judged relative to the norm for the development environment. To set up the indicators, an expert need only judge its 'strength' as an indicator of the underlying quality attribute. Given that the process quality really is high, how certain is it that the staff will be experienced?

We have found the truncated normal distribution [4] useful for creating a probability expressing an expert's assessment of the 'strength' of an indicator. The truncated normal distribution is simply a normal distribution which has been truncated at both ends and the result renormalized to give a total probability of unity. The distribution has the advantage that it can model probability distributions which do not tend to zero at either extreme, as well as offering great flexibility in the variety of distribution shapes which can be achieved.

For example, suppose:

Testing process well defined = *TNormal*('*TPQ*',0.6)

Testing staff experience = *TNormal*('*TPQ*',0.2)

(the truncation points have been omitted). This expresses the judgement that the staff experience is the stronger indicator, since it has a smaller variance parameter (0.2) than the other indicator. In both cases the mean value of the indicator is given by the parent process quality.

5.2. Lifecycle modelling

We show two examples of how the phase BN and its variants can be linked to model different lifecycles.

5.2.1. Iterative development

An incremental software lifecycle is modelled by a series of the ‘all activities’ phase BN. Fig. 6 shows this as it is displayed in the AgenaRisk toolset.

Fig. 7 shows an example of the predicted defects for this model. The three diagrams show the marginal distributions for the ‘Residual defects post’ variable in each of the three iterations. All three increments include all activities. However in increment 1 most of the effort is given to specification, whereas in increment 3, most of the effort is in testing and rework.

In increment 1, the defects before the start of the phase is set to zero and the new functionality to 5 KLOC. As this is mostly specification, the number of defects introduced is relatively low, with a median value of 93.

The marginal distribution for ‘residual defects post’ in iteration 1 becomes the prior distribution for ‘residual defects pre’ in iteration 2. As this is mostly development (it adds 25 KLOC of new functionality), the largest number of defects are introduced during this iteration. The marginal distribution of ‘residual defects post’ in iteration 2 has a median value of 335.

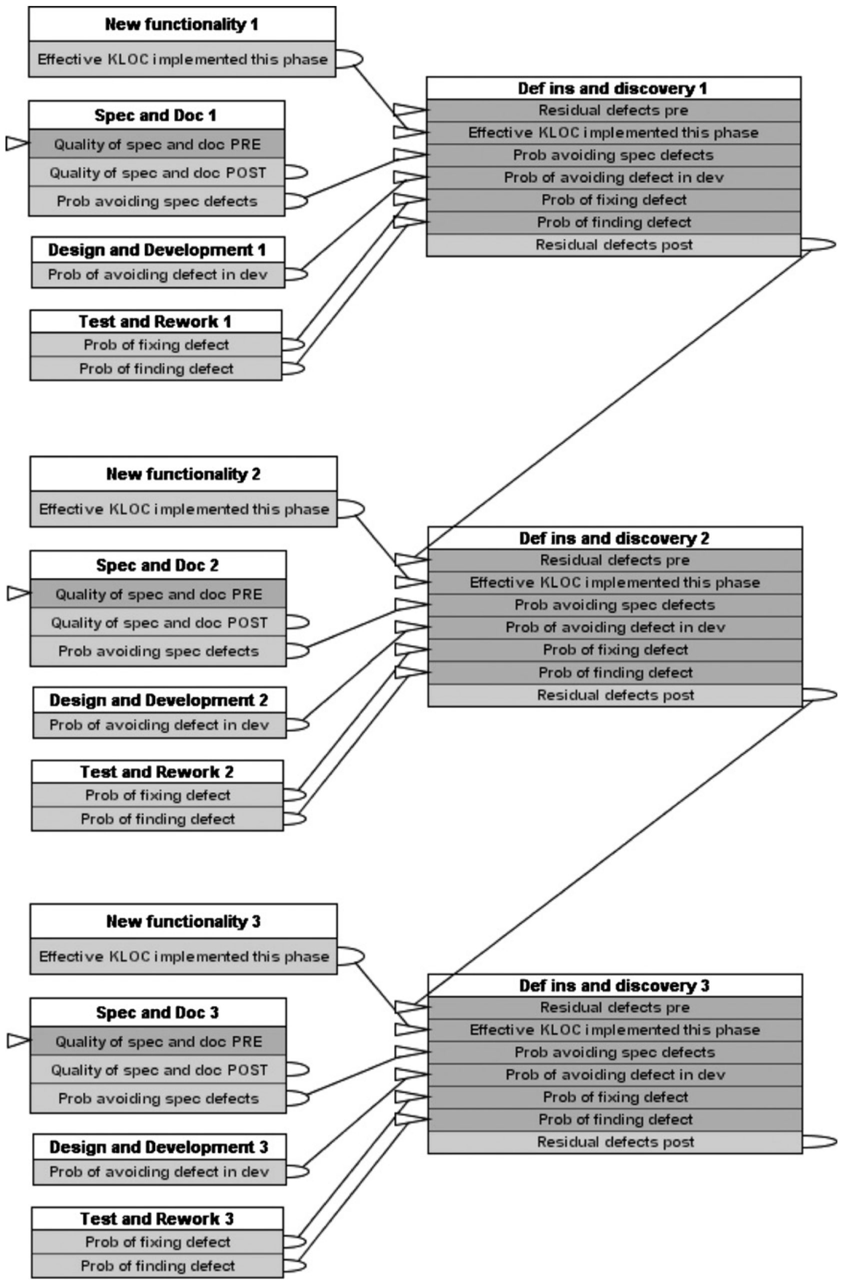

Fig. 6. An incremental development lifecycle.

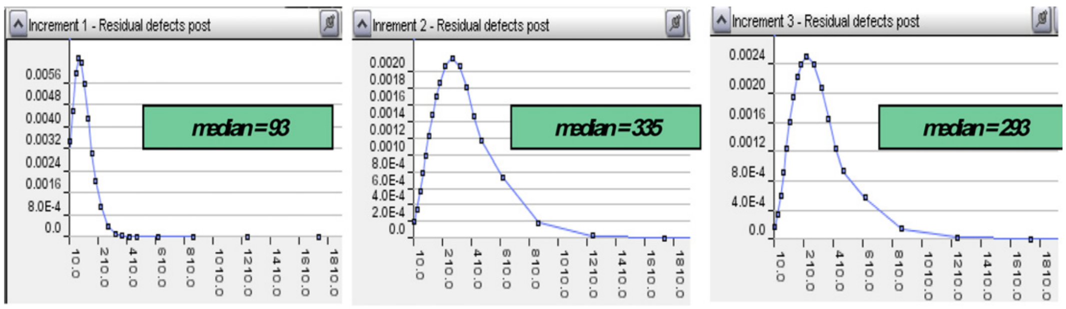

Fig. 7. Defects predicted at each increment of the incremental lifecycle.

This marginal distribution in turn becomes the prior distribution for the ‘residual defects pre’ in iteration 3. Only 5 KLOC of new functionality is introduced in this increment. The number of residual defects falls from increment 2 to increment 3 as a result of the testing effort.

5.2.2. *A waterfall example with integration*

The example in Fig. 8 shows a waterfall lifecycle but with initial development of modules 1 and 2, including some low-level testing, done by two separate teams, for example modelling development at different sites or the use of subcontractors.

The initial development follows different lifecycles: the two phases for module 1 being ‘specification, development but no testing’ followed by ‘testing only’, while module 2 has a specification only initial phase. This difference may represent the different way that metrics data is gathered at the two sites as well as actual lifecycle differences.

The ‘join’ class combines the defect estimates for the two modules, taking account of their relative size, before two phases of testing applied to the system as a whole. This example also shows that user trials can be modelled as a ‘testing only’ phase.

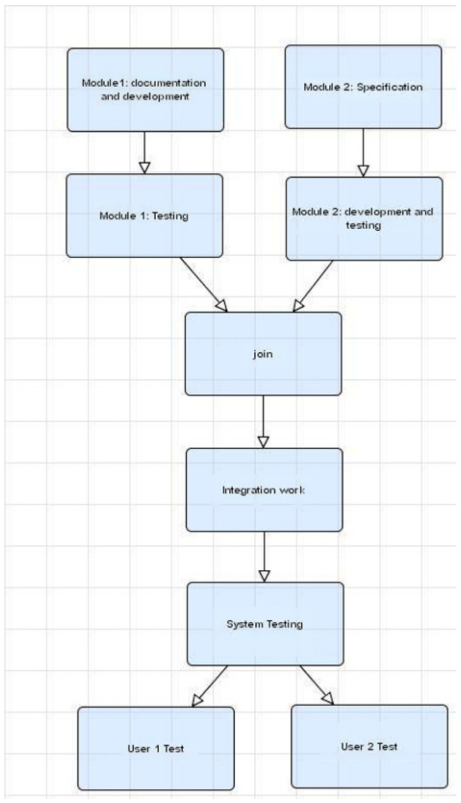

Fig. 8. A more complex lifecycle with two teams.

5.3. *Toolset*

Our experience from earlier commercial projects is that project managers and other users who are not BN experts do not wish to use a BN directly via a general purpose BN editor. Instead, the BN needs to be hidden behind a more specialised user interface. The toolset provided by AgenaRisk is actually an application generator that enables toolset users to tailor both the underlying BN models and the user interface that is provided to the end-users when the application is generated.

The main functions provided to the end-user are:

1. Observations can be entered using a questionnaire interface, where questions correspond to BN variables. Each question includes an explanation and the user can select a state (if the number of states is small) or enter a number (if the states of the variable are intervals in a numeric range). Answers given are collected into ‘scenarios’ that can be named and saved. At least one scenario is created for each software development project but it is possible to create and compare multiple scenarios for a project.
2. Predictions are displayed as probability distributions and as summary statistics (mean, median, variance). Distributions are displayed either as bar charts or as line graphs (see Fig. 7) depending on the type of variable and the number of states. The predictions for several scenarios can be superimposed for ease of comparison. Summary statistics can be exported to a spreadsheet.

The questionnaires shown to the end user can be configured widely. For example, questions can be grouped and ordered arbitrarily and the question text is fully editable. Not all variables need have a question, allowing any BN variable to be hidden from the end user.

6. Validation

The toolset and models have been widely trialled by various commercial organisations, including those involved in the MODIST project [13], namely Philips, Israel Aircraft Industries (Tel Aviv) and QinetiQ (Malvern). In addition, Philips has recently completed a retrospective trial of 30 projects carried out at Bangalore.

6.1. Aim and methodology

The aim of the recent Philips trial (2004–2005) was to evaluate the accuracy of the defect prediction BN capabilities in software projects. Initially, 116 consumer electronics software projects completed between 2001 and 2004 were assessed for inclusion in the trial against the following criteria:

- reliable data was available,
- project resulted in a commercial product,
- the size of the project was within the scope of the model's data ranges,
- some key people from the project were still available for interview,
- the projects should represent a range of product domains and a variety of design layers, including user interface, intermediate and driver layers.

Thirty projects were identified as suitable for the trial, based on these criteria.

A questionnaire, based on the AgenaRisk form for entering observations, was used to collect qualitative and quantitative information from key project members. These data were entered into the model to predict the number of defects found in testing. These predictions were then compared with the actual number of defects found in all testing phases. Data were collected in two rounds: in the second round a more detailed interview was conducted with the 'Quality Leaders' for each project resulting in improved data and improved predictions.

The trial used a variant of the 'all-activities' phase-level net. It is important to note that the model was developed in one development centre but trialled in another. These two development centres did not share a common corporate culture or development methodology. Apart from the involvement of their parent company, they were effectively separate organisations.

We initially assumed that the data received from Philips applied to software projects which consisted of a single development phase.

6.2. Results

Fig. 9 shows the predicted versus the actual defect counts for the projects in the initial trial.

Although our model shows a strong correlation with actual defects, its absolute predictive accuracy is poor.

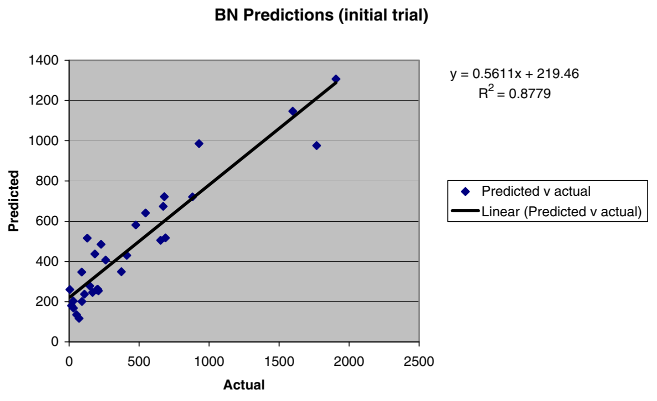

Fig. 9. Predicted verses actual defects (initial trial).

Perfect accuracy would result in all of the data points in Fig. 9 lying on a line with unit slope and zero intercept. Our model results differ significantly from the ideal. This indicates a systematic error in our model.

Our investigation into this error showed the need to ensure that the model closely matches the situation. For example, the inaccuracies for projects outside the range of the default model are largely explained by the 'defects pre' variable, representing the number of defects before the (one and only) development phase. Unless a value is explicitly entered here, a default distribution is assumed, which heavily biased the defect predictions upwards for the smaller projects and may also bias the prediction downwards for larger projects.

Although it is easy to enter a value in the AgenaRisk toolset, we did not provide a systematic method to determine the appropriate value. Many of the projects in the trial enhanced existing software, so the initial defects was not expected to be zero. This problem was easily overcome by explicitly modelling the pre-existing code, using an initial stub phase (no specification, development or testing).

This led to the final trial results shown in Fig. 10. As can be seen, the two phase model resulted in a significantly more accurate defect prediction model.

It would be perfectly possible to construct a regression model using the same data set. First we would perform

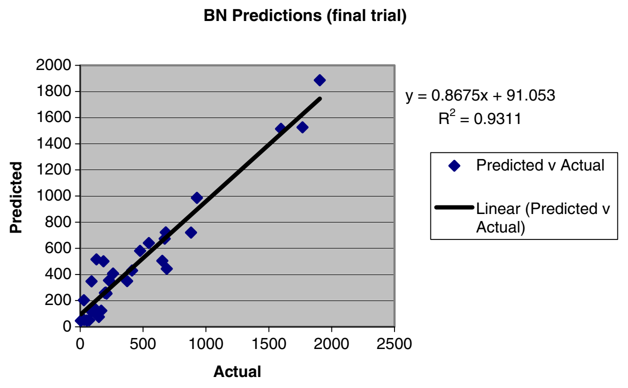

Fig. 10. Two phase prediction result.

an analysis of each variable to determine if the distribution of its values was symmetric or asymmetric, with a logarithmic transformation performed on those with asymmetric distributions. Variables which were strongly correlated with one another would then be identified and a decision made on which to eliminate. If the number of data sets was still lower than the number of variables, then further elimination candidates would be identified. Finally we would be in a position to identify the coefficients of our regression model based on some "best fit" criteria.

Defect models constructed this way generally produce good results. We can therefore expect our regression model to fit the data fairly well. However, we would have no real understanding of the applicability of such a model outside of the data set used to create it.

Contrast this with the causal model just described. This is independent of the data set. There is no need to identify the shape of variable distributions, nor to determine correlations between variables in order to eliminate some of them. All of the data can be incorporated, and where it is absent, prior assumptions used.

7. Conclusions

We have shown how a wide variety of software lifecycles can be modelled using a Dynamic Bayesian Net, in which each time frame is a lifecycle 'phase' combining all software development activities in different amounts. This approach allows a BN for software defect prediction to be tailored to different software development environments. The AgenaRisk toolset makes this a practical approach, providing a formula language with standard statistical distributions that can be used to change the quality indicators available in each software development team.

The approach and toolset have been extensively trialled by industrial partners in a collaborative project. Despite making little use of the available tailoring capabilities, a retrospective trial of 30 projects showed a good fit between predicted and actual defect counts. Once a small amount of tailoring was undertaken (to take account of the size of any legacy code in the projects under assessment) the predictions were outstanding (over 93% correlation between predicted and actual defects).

The AgenaRisk toolset allows the use of large variable state spaces that are necessary to achieve accurate predictions, with the formula language making the construction of very large probability tables feasible. The AgenaRisk toolset also now incorporates dynamic discretisation [17,19] to overcome the classic BN problem of discretisation errors that occur when numeric variables have a widely varying scale.

We have also used BNs to reason about software projects as a whole [13] and the trade-off between time, resources and quality. Many of the factors are common in these two models, covering both the assessment of process quality and the product quality achieved and required. In future, we hope to combine the two models into a single decision support system for software projects. Part of this is being done in the eXdecide project [6].

Acknowledgements

This report is based in part on work undertaken on the following funded research projects:

- MODIST (EC Framework 5 Project IST-2000-28749),
- SCULLY (EPSRC Project GR/N00258),
- SIMP (EPSRC Systems Integration Initiative Programme Project GR/N39234),
- SCORE (EPSRC Project Critical Systems Programme GR/R24197/01),
- eXdecide (EPSRC Grant Reference: EP/C005406/1).

We also acknowledge the contributions of the following individuals: Patrick Cates, Manesh Tailor and Simon Forey from Agena, Geert Acke and Shoaib Qureshi from Philips, Schlomo Gluck from Israel Aircraft Industries, Dave Milledge, John Elliot and Lisa Tipping from QinetiQ and David Corrall and members of the ASTUTE project team at BAe Systems.

References

[1] AgenaRisk: Advanced risk analysis for important decisions. http://www.agenarisk.com.
[2] O. Bangsø, P.H. Wuillemin, Top-down construction and repetitive structures representation in Bayesian networks, in: Proceedings of The Thirteenth International Florida Artificial Intelligence Research Symposium Conference, Florida, USA, 2000, pp. 282–286.
[3] S. Bibi, I. Stamelos, Software Process Modeling with Bayesian Belief Networks, in: Proceedings of 10th International Software Metrics Symposium (Metrics 2004) 14–16 September 2004, Chicago, USA.
[4] F. Cozman, E. Krotkov, Truncated Gaussians as Tolerance Sets, Technical Report CMU-RI-TRI, Robotics Institute, Carnegie Mellon University, 1997.
[6] eXdecide: Quantified Risk Assessment and Decision Support for Agile Software Projects, EPSRC project EP/C005406/1, www.dcs.qmul.ac.uk/~norman/radarweb/core_pages/projects.html.
[7] Fan, Chin-Feng, Yu, Yuan-Chang, BBN-based software project risk management, Journal of Systems Software 73 (2004) 193–203.
[8] N.E. Fenton, M. Neil, A Critique of Software Defect Prediction Models, IEEE Transactions on Software Engineering 25 (5) (1999) 675–689.
[9] N.E. Fenton, M. Neil, SCULLY: Scaling up Bayesian Nets for Software Risk Assessment, Queen Mary University of London, www.dcs.qmul.ac.uk/research/radar/Projects, 2001.
[11] N.E. Fenton, P. Krause, M. Neil, Probabilistic Modelling for Software Quality Control, Journal of Applied Non-Classical Logics 12 (2) (2002) 173–188.
[12] N.E. Fenton, P. Krause, M. Neil, Software Measurement: Uncertainty and Causal Modelling, IEEE Software 10 (4) (2002) 116–122.
[13] N.E. Fenton, W. Marsh, M. Neil, P. Cates, S. Forey, T. Tailor, Making Resource Decisions for Software Projects, in: Proceedings of 26th International Conference on Software Engineering (ICSE 2004), IEEE Computer Society, Edinburgh, United Kingdom, 2004, pp. 397–406.
[14] F.V. Jensen, An Introduction to Bayesian Networks, UCL Press, 1996.

[15] C. Jones, Programmer Productivity, McGraw Hill, 1986.

[16] C. Jones, Software sizing, IEE Review 45 (4) (1999) 165–167.

[17] D. Koller, U. Lerner, D. Angelov, A general algorithm for approximate inference and its application to hybrid Bayes nets, in: Proceedings of the 15th Annual Conference on Uncertainty in AI (UAI), Stockholm, Sweden, 1999, pp. 324—333.

[18] D. Koller, A. Pfeffer, Object-oriented Bayesian networks, in: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI97), Providence, Rhode Island, USA, 1997, pp. 302–313.

[19] A.V. Kozlov, D. Koller, Nonuniform dynamic discretization in hybrid networks, in: Proceedings of the 13th Annual Conference on Uncertainty in AI (UAI), Providence, Rhode Island, August 1997, pp. 314–325.

[20] M. Neil, N.E. Fenton, S. Forey, R. Harris, Using Bayesian belief networks to predict the reliability of military vehicles, IEE Computing and Control Engineering 12 (1) (2001) 11–20.

[21] M. Neil, N.E. Fenton, L. Nielsen, Building large-scale Bayesian networks, The Knowledge Engineering Review 15 (3) (2000) 257–284.

[22] M. Neil, P. Krause, N.E. Fenton, Software quality prediction using Bayesian networks, in: Software Engineering with Computational Intelligence, (Ed Khoshgoftaar TM), Kluwer, 2003 (Chapter 6).

[23] M. Neil, M. Tailor, D. Marquez, Inference in hybrid Bayesian Networks using dynamic discretisation. Accepted for publication in Statistics and Computing.

[24] I. Stamelos, L. Angelis, P. Dimou, E. Sakellaris, On the use of Bayesian belief networks for the prediction of software productivity, Information and Software Techniques 45 (1) (2003) 51–60.