Bayesian Graphical Models for Software Testing

David A. Wooff, Michael Goldstein, and Frank P.A. Coolen

Abstract—This paper describes a new approach to the problem of software testing. The approach is based on Bayesian graphical models and presents formal mechanisms for the logical structuring of the software testing problem, the probabilistic and statistical treatment of the uncertainties to be addressed, the test design and analysis process, and the incorporation and implication of test results. Once constructed, the models produced are dynamic representations of the software testing problem. They may be used to drive test design, answer what-if questions, and provide decision support to managers and testers. The models capture the knowledge of the software tester for further use. Experiences of the approach in case studies are briefly discussed.

Index Terms—Bayesian graphical models, expert judgment, knowledge capture, software reliability, software testing, statistical methods, test design.

1 Introduction

In a recent article, Redmill [17] discusses a number of deficiencies of available software testing approaches. For example, he states that "testing can prove imperfection by finding a single fault, but it cannot prove perfection" and "given that we can never prove perfection, we want to avoid testing beyond the point of significantly diminished returns—while still achieving the desired level of confidence." He emphasizes that testing is a risk management activity where "we need to choose test cases carefully, to achieve the necessary coverage while avoiding replication." According to Redmill, "there is no definitive answer to the question of what is the minimum level of testing needed to secure the desired level of confidence. In fact, usually there is no knowledge of what level of confidence is desired. Indeed, the issue is almost never addressed." The Bayesian graphical model (BGM) approach presented within this paper provides a natural logical and probabilistic framework to software testing which allows all these issues to be resolved.

The theory of BGMs [3], [10], [13], [16], [24] has led to many new applications of uncertainty modeling, in particular, to complex problems where a large number of factors contribute to overall uncertainty. BGMs derive from Bayesian statistical methodology, which is characterized by providing a formal framework for the combination of data with the judgments of experts such as software testers. For the application area of software testing, we demonstrate how the problem should be structured and how the resulting models may be used. We illustrate the methodology with case studies arising from applying the approach to large-scale software testing problems for a major UK company.

In Section 2, we briefly present the mathematical theory underpinning the approach. Our presentation throughout is targeted at the software testing community so that we limit statistical technicalities as far as is practicable. In Section 3, we describe practical aspects of the BGM approach, including the structural modeling and the assessment of the relevant probabilities. In Section 4, we describe the use of the BGM for testing and sensitivity analyses. In Section 5, we summarize the application of the methodology to two case studies. Finally, in Section 6, some related issues are briefly discussed.

1.1 Uncertainty in Software Testing: Related Approaches

General issues in software reliability have attracted attention from statistical researchers: Overviews and references can be found in [12], [22]. However, much of this work attempts to fit problems related to software reliability within existing mathematical frameworks rather than attempting carefully to model the actual uncertainties occurring in software testing and the process of learning from tests. We make case studies central to the development of the methods described in this paper as the emphasis of our approach is in modeling the actual testing process and, thus, contributing to better testing.

Yamaura [27] describes the relevance of careful documentation of test cases. One advantage is the possibility of repeating the same tests, perhaps by another tester. Two further stated advantages are easy validation of the quality of the test cases and estimation of the quality of the target software. However, Yamaura does not specify how to perform such validation and estimation. These issues and requirements are clearly addressed by the BGM approach. Another important test advocated, but not described, by Yamaura is a 48-hour continuous operation test, which is aimed at revealing faults related to memory leakage, deadlock, and connection time-out. We acknowledge the

• *The authors are with the Department of Mathematical Sciences, University of Durham, Durham, DH1 3LE, UK.*
E-mail: {d.a.wooff, michael.goldstein, frank.coolen}@durham.ac.uk.

Manuscript received 13 Dec. 1999; revised 9 Apr. 2001; accepted 23 Aug. 2001.
Recommended for acceptance by J. Bechta-Dugan.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111102.

importance of such testing. It requires sophisticated modeling, but can be handled by the BGM approach. However, we do not discuss this further in this paper.

Smidts and Sova [23] present another approach for software reliability quantification, placing the functional architecture of the software centrally in their model. They suggest that their model "encourages a testing philosophy directed toward the triggering of failure modes and removal of related faults," but they do not provide further guidelines for testing at the input partition level. It would be interesting to consider if such, or related, models could be used in the process of creating BGMs for software testing.

Frankl et al. [5] identify two main goals in testing software: to achieve adequate quality (*debug testing*) and to assess existing quality (*operational testing*). The objective for debug testing is to probe software for defects so that these can be removed. The objective for operational testing is to gain confidence that the software is reliable. They examine the relationship between these testing goals via a probabilistic analysis in which the effectiveness of testing is based on the reliability of a program after testing. Both approaches are based on subjective arguments: Debug testing relies on insights on where faults are likely to be; operational testing depends on knowledge and assumptions on operational profiles. Both approaches have advantages, depending on the practical situation, and both depend on partitioning of the input space. The BGM approach also requires partitioning the input space. Our partitioning is driven by focusing on differing software actions (SAs), which we regard as essential to test complex software. Frankl et al. carefully discuss the difficult problem of defining "faults" that are responsible for failures and suggest avoiding the term "bug" because of its often vague definition. They conclude that a formal treatment of "faults" is not available and suggest using "failure regions" of the input space, where one such region is a set of failure points that is eliminated by a program change. We stay close to this in our analysis by probabilistically tracking back an observed failure in the graphical model to see to which specific SA the cause of failure is likely related, to guide attempts to fix the fault. Detailed operational testing enables a statistical analysis, when based on a large number of tests, of the reliability of the software in operation. While we support such testing whenever possible, resource constraints for our applications, together with vague knowledge about operational profiles, often prevent us from applying such testing. However, some aspects of operational profiles are reflected in utilities which influence the design of test suites.

One aspect shared by most practical software testing approaches is partitioning of the input domain into subdomains such that any input in the same subdomain is considered to have the same effect on the system. The testing task then becomes the task of selecting at least one representative from each partition. The art of testing then becomes that of defining the partitions. Such partitions can be created either from a black-box (see [4]) view of the system utilizing knowledge of the software requirements and specification, through a white-box view utilizing a structural analysis of the code, or by a combination of these two views. While there are a number of techniques that can be used, the process is often thought to be still fraught with difficulties and problems [7], [8], [25], [26]. Hierons and Wiper [9] discuss the estimation of failure rate on the basis of both random and partition testing. Again, their work depends strongly on an assumed operational profile. In our approach, we also partition the input domain using groups of similar inputs, where the similarity is a subjective assessment with regard to the software actions shared. We carefully discuss how this is executed and the effects actual tests have on our confidence in the quality of nontested inputs, both in the same subdomain as tested inputs and in other subdomains.

Regression testing is the process of testing a program after changes have been made to it to ensure that the changes have been effective and have not introduced further faults. For such testing, there already exists a set of prior test cases. It is not usually an option to rerun all of these tests, so some method for regression test selection needs to be devised. Rothermel and Harrold [19], [20] evaluate current regression test methods. All of the methods analyzed are based on information about the source code before and after modification and yet none of the methods attempt to capture expert knowledge about the software and the tests. Regression testing is not explicitly addressed in this paper, but is deferred to a subsequent report in which we will demonstrate formal logical and probabilistic mechanisms for regression testing within the BGM approach and which enables a routine and automatic treatment of regression testing.

Many statistical methods have been suggested or used in attempts to create better software. Burr and Owen [2] describe the possible use of methods from classical statistical quality control, although the adaptation to software problems is rather vague. Our first case study was far too complex to allow any standard classical statistical methods to be used in a straightforward fashion and this appears to be typical for testing problems. Singpurwalla and Wilson [21], [22] give overviews of an active area of statistical research in software reliability. Mostly, these are contributions to the theory of stochastic processes, linking reliability metrics to assumed behavior of processes with which failures occur, both while testing and in operation. They also discuss testing aspects related to such assumed models. While such approaches are potentially interesting, it seems that direct application is currently only possible to software of rather restricted complexity.

A classical approach to experimental design can be useful in some software testing situations when choosing test suites. Dalal and Mallows [4] discuss factorial designs which seem promising for smaller applications. In complex situations, such as our first case study, straightforward adaption of their approach is not feasible. Our automatic test design algorithm (Section 4.2) uses simple stepwise methods to arrive at efficient test suites according to appropriate criteria. These methods offer a great improvement on current practice, although there is scope for refinement, which is the subject of ongoing research. Compared to the approach by Dalal and Mallows, an advantage of our approach is that it insists on proper structuring and delivers outputs which make testing of functionality in multiple areas both feasible and desirable and that these outputs drive appropriate experimental design. The nature of our case studies is such that we treat the software being tested as black-box in that the structures relate to the detailed knowledge of the tester. If detailed code analysis information is available, it can be embedded in our approach.

BGMs, also called Bayesian belief networks (BBN), have been applied to different problems on software quality. Neil and Fenton [14] use them to predict software quality, taking into account a diversity of factors such as effort and complexity of design, skills of people involved in the process of development and testing, and costs. While this is an interesting approach to give an overall idea of the density of defects in a piece of software, these BBNs are not aimed directly at assisting testers. The most typical use of BBNs in this application area is in inferring models for reliability from large databases. The SERENE project (http://www.hugin.dk/serene/) presents interesting applications of BBNs in the area of safety and risk evaluation and some of the work within this project also takes software into account [1], albeit without actual support at the test design level. Our usage of BGM rather than BBN reflects the terminology used in the wider statistics literature and helps to emphasize that our BGMs model testers' judgments and are not the results of inferring models from large databases.

2 The BGM Approach to Software Testing

2.1 Introduction

Suppose that the function of a piece of software is to process an input number, such as a credit card number, in order to perform an action and that this action might be carried out correctly or incorrectly; for example, the action might be to check whether the account corresponding to the number is in credit. The various tests that we may run correspond to choosing various numbers and checking that the *software action* (SA) is performed correctly for each number. Usually, we will not be able to check all possible inputs, but instead we will check a subset from which we will, hopefully, be able to conclude that the software is performing reliably. Already we can see that, whether we quantify the uncertainties or not, a subjective judgment must be made about the functionality of the software over the collection of all inputs that have not been tested. Therefore, we must choose whether we explicitly quantify the uncertainties concerning further failures given our test results or whether we are content to make a purely informal qualitative judgment for such uncertainties. In many areas of risk and decision analysis, uncertainties are routinely quantified as subjective probabilities representing the best assessments of uncertainty of the experts carrying out the analysis. In this paper, we will argue the benefits of quantifying and analyzing our uncertainties for software failure as subjective probabilities. Such an analysis involves a certain, possibly substantial, investment of effort on our part in thinking carefully about our prior knowledge. However, the reward from this investment is that we may make precise probabilistic statements having seen the test results which describe our confidence in the functionality of the software, rather than making an informal guess as to what the result might be of carrying out the exact calculations. We may also gain further advantage from the practical uses of the fully specified probabilistic model, which may play a central role, for example, in risk management for the software release and as a decision support tool for creating and analyzing proposed suites of software tests.

If we accept that, in principle, the probabilistic analysis is the correct way to proceed, then we must consider, first, whether the work involved in constructing the belief model is worth the gains in precision that we may obtain and, second, whether this analysis requires the tester to make judgments that are beyond his ability to make. The first question is strongly context dependent as it takes the same amount of effort to build the probabilistic description if the consequences of software failure are negligible or if they are catastrophic. In a further paper, we will show how to adjust the level of detail in the formal analysis to its potential benefit. We will address the second question by showing how uncertainties for software failure may be systematically modeled in the form required for a probabilistic analysis of the testing process.

The simplest case occurs when the tester judges all possible test results to be *exchangeable*. This expresses the judgment that we have no information about the differences in software reliability for different subsets of the numbers. Thus, the subjective probability that an individual test will succeed is the same whichever number is chosen to test, the probability that any pair of tests will both succeed is the same whichever pair of nonidentical numbers is chosen to test, and so forth.

Qualitatively, exchangeability is a simple judgment for the tester to make: Either there are features of the set of possible inputs which cause him to treat some subsets of test outcomes differently from other subsets or, for him, the collection of outcomes is exchangeable. In this example, suppose that the software is intended to cope with both short and long numbers (in one of our case studies, number length could be anything from 8 to 16 digits) and that he judges test success to be exchangeable for all short numbers and test success to be exchangeable for all long numbers; for example, it might be that the first release of the software only dealt with credit cards with short numbers, but, in a subsequent release, cards with long numbers have also been incorporated.

As we increase the complexity of the example, it becomes increasingly important to have methods for systematic description and analysis of the uncertainties that arise. We will use BGMs for this purpose. These models provide a flexible and powerful way to organize the belief modeling, quantification, and analysis for software testing. We now describe how such graphical models may be constructed.

2.2 Modeling a Software Action

In our example, suppose that we are primarily interested in the probability of the event N, that the software contains no faults. We decompose this event by considering the two overlapping events, S, that at least one of the short numbers is processed incorrectly and, L, that at least one of the long numbers is processed incorrectly. We wish to model our beliefs over S, L, N and we intend to use a BGM.

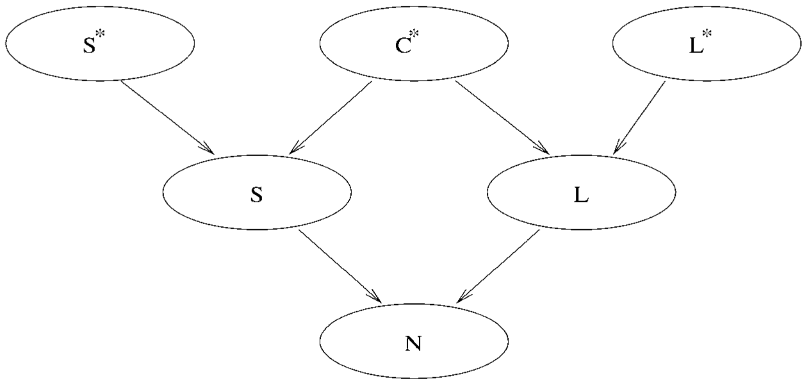

Fig. 1. A Bayesian graphical model showing the structure underlying the handling of number faults and judgments as to the causes and their relationship.

In a BGM, we represent each uncertain event by a node. Some nodes may be joined by directed arrows. If a directed arc goes from node A to node B, then node A is termed a parent of B and node B is termed a child of A. We term node A a root node if A has no parents. Node B is a descendent of node A if there is a directed path from A to B. Informally, a directed arc from node A to B indicates that the probability for node B is influenced by the value of node A. More strongly, the collection of parents of a node determines the probability at the node in the following sense: For any pair of nodes B and C which are connected via parent nodes, if we specify the outcomes of all of these parent nodes, then B is conditionally independent of C.

To construct a graphical model, we draw a directed, acyclic graph to represent the qualitative relationships between the nodes. We then quantify all of the probabilities for the events in the graph as follows: First, we determine the probability of occurrence of each event represented by a root node. Second, for each child node, we determine the probability of occurrence of the event represented by the node conditional on the occurrence or nonoccurrence of each parent of that node. These specifications determine the probabilities for all combinations of events represented on the graph. Because the full joint probability distribution has been specified, each time we observe an event (for example, the success of a software test), we may update all of the probabilities for all of the remaining events in the model (for example, the event that there are no faults in the software) by probabilistic conditioning. Further, because of the structure of the graphical model, it is straightforward to carry out such belief updating, even for large models, using principles of local computation. General treatments of BGMs can be found in [3], [10], [13], [16], [24].

In our example, we have three events, S, L, and N. One way to think about the relationship between events S and L is to consider that there are three possibilities for faults in the code: problems which only affect short numbers, problems which only affect long numbers, and problems common to all numbers. We introduce three further events, S^*, L^*, and C^*, to express these possibilities so that S^* is the event that there is a problem in the code which may produce faults for short but not long numbers, L^* corresponds to faults in long but not short numbers, and C^* corresponds to faults which may occur in any number. These events are not directly observable, but they are helpful in explaining our beliefs relating to the observable events S and L. We will treat S^*, L^*, and C^* as independent.

Our model is given as Fig. 1. There are three independent root nodes, S^*, C^*, and L^*. Node S has parents S^* and C^*, node L has parents C^* and L^*, and node N has parents S and L. Thus, S and L are independent given S^*, L^* and C^* and N is independent of S^*, C^*, and L^* given S and L.

We now quantify beliefs over the model. The model can be quantified directly or by adopting an elicitation process, such as is described in Section 3.5. Here, we first specify probabilities for the root nodes, beginning by ranking the relative failure probabilities. We let $\mathrm{P}(C^*)$ be that probability that there is at least one fault in the software corresponding to C^*. Suppose that the tester judges $\mathrm{P}(C^*)$ to be much larger than $\mathrm{P}(L^*)$, which is much larger than $\mathrm{P}(S^*)$, which is judged to be small. While we may find it difficult to give precise values for these probabilities, we will usually be able to give values which appear to be of reasonable orders of magnitude. When we carry out the full probabilistic analysis, we check the sensitivity of our conclusions by varying the probabilistic inputs. Usually, the conclusions will not be sensitive to small variations in the probabilistic assignments. However, if our conclusions for the reliability of the software do depend critically on certain input values, then the inescapable conclusion is that these values must be considered carefully. In this way, we identify those aspects of the quantification where it is important to be precise and which have strong implications for the testing process.

We now specify conditional probabilities for each child node given each parent node. In the present example, suppose that we consider the probability that S occurs to be one if either S^* or C^* or both occur and 0 otherwise and the probability that L occurs to be one if either L^* or C^* or both occur and 0 otherwise. Similarly, the probability that N occurs is one if neither S nor L occur and is 0 otherwise. In this example, all of the links are deterministic, i.e., if the parent event occurs, then the child event will also occur. In general, if a parent root node in our structure has a single child, then we will usually assign a probability of one that the child inherits the problem from the parent. In other cases, the links may be probabilistic; for example, if the common problem C^* could potentially affect many child

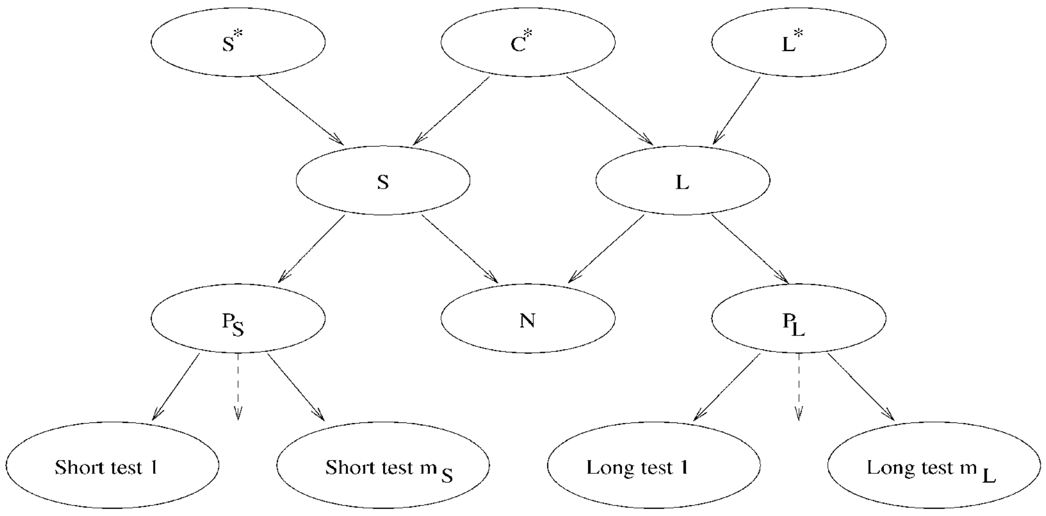

Fig. 2. Linking multiple possible test outcomes to the basic BGM.

nodes, then we might identify some child nodes that would definitely be affected by the common node, some that would be affected with high probability because they shared many characteristics with these nodes, and some that would be affected with low probability as they shared few characteristics.

2.2.1 Linking Test Outcomes to Domain Nodes

To complete the graphical model, we must link test outcomes to the graph. Consider tests of short numbers. If there is at least one fault in handling a short number, then we must consider the proportion, p_S, of short numbers which will give correct responses when tested. The quantity p_S is unknown and, therefore, must be given a probabilistic description. A simple but useful form is to consider the probability distribution for p_S to be a mixture of two components. The first component is a point mass q_S at the value $p_S = 0$. Therefore, q_S is the probability that, given that there is a fault in at least one short number, then this fault will occur in whichever short number that we choose to test. In many cases, this will be sufficient as a single test success or failure will determine whether the node will always or never be in error. For more general cases, the remaining probability, $q'_S = 1 - q_S$, is represented by a continuous probability density on (0, 1). A natural choice would be to suppose that p_S has a Beta distribution, $Be(\alpha_S, \beta_S)$, so that the probability density of p_S is

$$f(p) = \frac{\Gamma(\alpha_S + \beta_S)}{\Gamma(\alpha_S)\Gamma(\beta_S)} p^{\alpha_S - 1}(1-p)^{\beta_S - 1}, \quad 0 \le p \le 1,$$

with parameters α_S, β_S, which we must select to represent our beliefs about the shape of the prior distribution. Values $\alpha_S = \beta_S = 1$ correspond to the uniform distribution on [0, 1] and changing the ratio $\alpha_S/(\alpha_S + \beta_S)$ moves the center of the distribution toward 0 or 1, while large values of one or both of these parameters correspond to small variation for the distribution. The choice of appropriate values can most easily be carried out with a computer-based elicitation tool.

We select the Beta distribution as this is the standard conjugate prior distribution for binomial sampling. That is, the conditional distribution for p_S, given that $p_S \neq 0$, if we observe a collection of test outcomes, is also a Beta distribution. The implication of a test pass is to update the parameters α_S, β_S to values $\alpha_S + 1, \beta_S$, while a test failure updates the values to $\alpha_S, \beta_S + 1$. This greatly simplifies the calculations for updating probabilities. Note that each test pass moves the probability distribution for p_S toward one as we increase the overall probability for observing test successes. We similarly construct a model for tests on long numbers, selecting values q_L, α_L, β_L.

Suppose that we carry out m_S tests on different short numbers and m_L tests on long numbers. We add to the graph (Fig. 1) nodes P_S and P_L to represent the models for observed tests for each observable, together with the nodes for each actual test, giving Fig. 2. We have only shown the nodes for the first and last test in each group. As the nodes P_S, P_L are continuous random quantities, we must specify the conditional probability of each child node, namely, each test outcome, conditional on each possible value of the parent node. This is straightforward in this case as the probability that a test fails, for example, for a short number, is equal to the numerical value of the parent node P_S. Notice that, in Fig. 2, the only parent that we have chosen for P_S is the event S. In a more complex model, we might have several links into P_S.

2.2.2 Simplified Representations

Fig. 2 is a full description of the model. It can be convenient to show the model in a reduced form. Partly, this is so that greater complexity can be accommodated while retaining a visualization of the essential structure without becoming mired in irrelevant details. In addition, this also reflects the way in which graphical modeling software handles the computations, for which we wish to avoid including any nodes which are not strictly necessary to the computational description of the process. There are three essential simplifications. First, we do not need to describe the test nodes so extensively. All that is important is that there is a test process which is connected to a single node. Second, any node, such as N, which is a deterministic function of

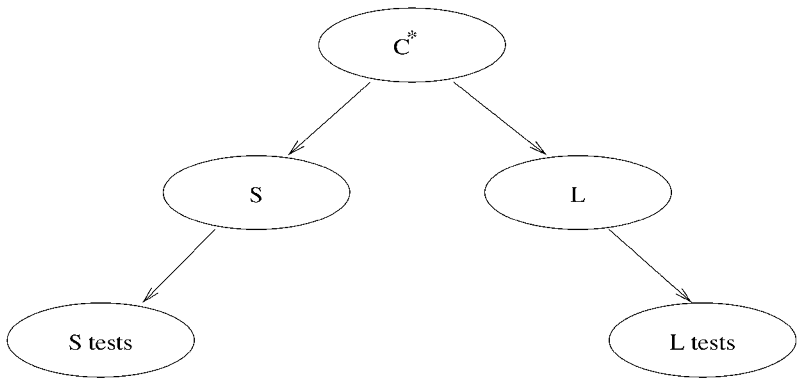

Fig. 3. A reduced form depiction of the full model shown in Fig. 2.

the parent nodes, can be removed from the diagram. Finally, any root node with a single child node can be absorbed into that child node. For example, if we absorb node S^* into S, then the only modification that we must make is to increase the conditional probability that S occurs if C^* occurs and, if C^* does not occur, to take into account the probability that S occurs given that S^* occurs. Therefore, we have a reduced form representation of Fig. 2 as Fig. 3. This reduction is particularly helpful as we increase the number of characteristics which may influence the software failure. We illustrate this process by introducing a further failure condition.

2.3 Combinations of Software Problems

In our example, suppose we realize that, in addition to problems with short and long numbers, there is a potential problem with numbers which begin with a zero as we suspect that the software can sometimes incorrectly strip out a leading zero (for one of our case study models, there were five different potential problems involving the value of the starting and ending digit). We construct the further events Z, the event that at least one number beginning with a zero is processed incorrectly due to problems with handling zeros as the first digit of the number; X, the event that at least one number not beginning with a zero is processed incorrectly due to problems with handling nonzero first digits. Suppose that we see no reason to suppose that correct initial digit processing should be related to correct number length processing so that we judge pair Z, X to be independent of pair S, L.

We have four subgroups of tests, which we abbreviate as $[S, Z]$ (short numbers beginning with zero), $[S, X]$ (short numbers not beginning with zero), $[L, Z]$ (long numbers beginning with zero), and $[L, X]$ (long numbers not beginning with zero). Failures for numbers within each group are judged to be exchangeable. Our graphical model is given in Fig. 4. We are using the reduction that we introduced in Fig. 3 for which we only display parent nodes with more than one child. Thus, we show the node I^* which is the event that there are initial digit faults which may occur in numbers starting with any digit, but we do not show the event Z^* which corresponds to faults solely in numbers beginning with zero. Similarly, each of the individual test nodes, for example $[L, Z]$, may be linked to a corresponding root node describing possible problems which may manifest only for that particular combination of test factors.

As before, we specify probabilities for each root node and conditional probabilities for each child node given the outcomes for each parent node. For example, for the event

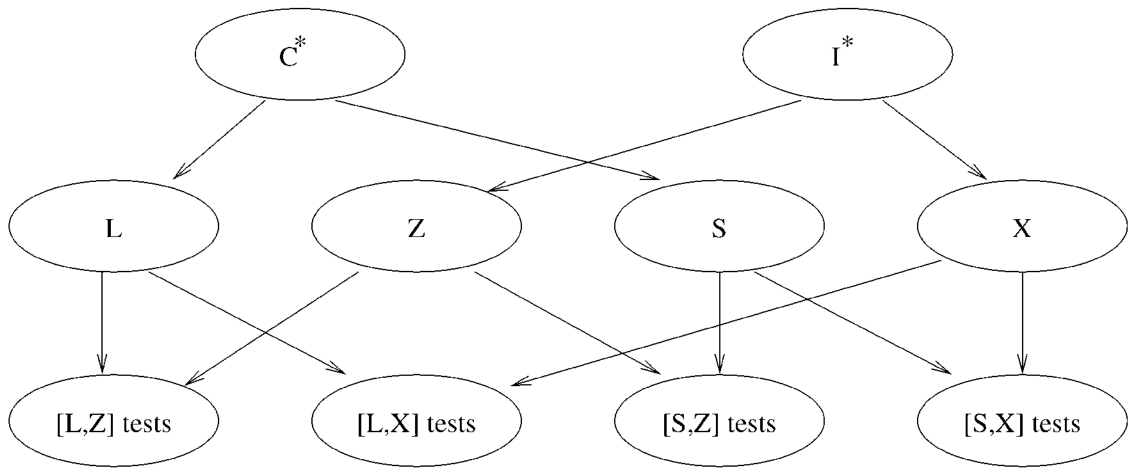

Fig. 4. A reduced form depiction of the full model for a software problem with a combination of two possible failure modes.

that there is at least one fault in the treatment of $[S, Z]$ numbers, we would have to specify four conditional probabilities: the conditional probability that this event occurs if events S and Z both occur, if S occurs and Z does not, if Z occurs and S does not, and if neither S nor Z occur. It may be suitable to judge these effects as independent, in which case, the probability of at least one fault for $[S, Z]$ if S and Z both occur is simply the sum of the probabilities of a fault given S and a fault given Z minus the product of these two probabilities. However, we have complete freedom to construct each conditional probability table according to whatever interactions we deem appropriate.

The conditional probabilities may also be influenced by higher order nodes expressing further hierarchical structure in the software. For example, the software may process the input number to perform a variety of functions and we may construct nodes to express the relations between faults in one function, such as adding a new card number to a database, and faults in another function, such as modifying the value of the number.

2.4 The Test Procedure

The test procedure for the model is as follows: We select a number to test. If the test fails, then the software is modified and retested. The BGM for the software is adjusted to reflect information about the nature of the software failure and our beliefs about the likely success in fixing the problem without introducing new faults. For example, the tester may judge that previously successful tests in areas which are very different from that in which the problem has been identified are unlikely to reveal any new faults, while those in closely related areas are likely to require extensive retesting. This adjustment to the BGM may either be formally modeled or carried out informally by the tester. Whichever is the case, the adjusted model provides the appropriate knowledge base for both further testing and regression testing.

For tests which are successful, we probabilistically propagate the implications of the success across the BGM. This reduces the current probability of software failure for many of the various nodes on the model and, particularly, for those nodes most strongly connected to the node where we have observed a test pass. We then choose a further test. We continue in this way, fixing and retesting each time we find a fault and updating our probabilities for successful tests, until we either exceed our test resources or we reach a point where our probability that the software is reliable is sufficiently high that there is judged to be no need for further testing. This criterion may be refined, if there are several different types of potential faults, to terminate with low probability for faults with major consequences but to tolerate a higher probability for faults with minor consequences.

The BGM approach provides probabilistic assessments of the reliability of the software being tested before and during the testing process. As such, these assessments provide a natural approach to test design. There are two criteria that we require for the test suite. First, we would like to judge the software acceptable if all of the tests are successful so that we want to choose the test suite which maximizes the conditional probability of software acceptability, given success for each test, subject to any resource constraints. We may assess the value of such a termination probability before carrying out the test suite and, thus, may judge a priori whether the resources are sufficient to test the software to the required level of confidence. Similarly, we can use such judgments to determine a realistic schedule for release of the software. Second, each time we find a fault, there may be some need for regression testing. We typically prefer to find most of the faults as early as possible in the testing sequence. Thus, when we have selected the test set to optimize a termination probability, we then sequence the tests so that, at each stage, we choose, subject to any practical constraints, the test with maximum probability of finding a fault given that each previous test has been successful.

We may therefore use the probabilistic model either to generate and sequence the test suite in a purely automatic fashion or to function as a decision support system for evaluating and choosing how to sequence a suite suggested by the tester and to check for additional tests which may have been overlooked. We shall address test design issues more fully in a separate paper. Briefly, within the BGM approach, it is straightforward to design tests to take account of differing levels of fault consequence and to show how regression testing and the test-retest cycle can be accommodated and resolved.

3 Practical Aspects

We have used the BGM approach in several case studies undertaken for a major UK company. In this section we discuss the main practical aspects of the approach. In Section 4, we describe the kinds of analysis that are enabled by the approach. In Section 5, we summarize results from the application of the approach to two case studies.

3.1 Structuring Prior to Modeling

Before the software system can be modeled graphically, it must be organized in ways which we describe below. Sometimes the organization we describe can be obtained directly from the functional specification for the software, either from documentation or as a by-product of the software design process. In cases where software is sourced from external suppliers according to a functional specification, the tester can only test whether the software achieves the functional specification and the software is treated as black-box in that it is possible only to observe what outputs are given by specified inputs and it is not generally possible to observe at the programming level how those outputs are determined. Thus, in order to test the software efficiently, it is necessary for the tester to relate the possible tests to his subjective assessment of what the black-box software must do to achieve the functional specification.

Such structuring stages are extremely valuable, regardless of whether Bayesian graphical modeling is applied to the structures that result, as it is important to have a clear and structured appreciation of how the different functionality interacts, what the SAs are, what the possible inputs are, and so forth. With respect to such interactions, as there is usually only resource available to test a small number of the combinations of possible inputs, confidence in test results depends strongly on beliefs about such relationships. The structuring of a software system typically includes the following stages.

3.1.1 *Initial Structuring*

We begin by listing the *transactions* which the software system performs. By a transaction, we mean a major software function, such as adding a credit card to a database, modifying a customer's details, and so forth. Each transaction will typically involve many SAs (such as processing a card number), which may or may not be shared with other transactions. The main point is to identify any features, such as shared code, that relate the transactions.

3.1.2 *Identifying the Software Actions*

Each transaction involves a possibly large number of distinct software actions (SA). An SA is defined to be software code responsible for a particular piece of processing and which can be specifically tested. From our case studies, one example was the SA that encrypted a credit card number. A second example is provided by the SA which was responsible for transmitting a record from one of the databases to another. With respect to the level of detail involved, we consider an SA to be an action such that, at the testing phase, from the tester's point of view, there is no value or intention (for whatever reason) in deconstructing into further SAs.

3.1.3 *Relationships between Software Actions*

It is important to establish which SAs are common to other transactions and to record whether any of the SAs are related to other SAs. There are three situations of interest. Suppose that there are two SAs, A and B, forming parts of two transactions (or occurring at different points within the same transaction). A and B are *common* if A and B are the same piece of code so that a test of A with a given set of inputs necessarily also tests B if the details are identical. A and B are *related* if, before you carry out a test of A, you expect the test of A to also give you some information about the reliability of B. A and B are *independent* if you believe that a test of A cannot give you any information about the reliability of B. In making such judgments, we assume that any earlier processing which needed to have been carried out before entry to this SA has worked perfectly. That is, any failure in this SA has not occurred because of a failure upstream. Software actions may be related for various reasons: For example, they may be variants of the same piece of code, they might be different code but using similar algorithms (e.g., decryption), or it might be known that the code has been supplied by the same software provider, perhaps notoriously incompetent. In such cases, the software actions remain distinct, but related.

3.1.4 *Sequencing the Software Actions*

A chronological order for the various SAs must be established. This is vital to the design and interpretation of tests and the elicitation of failure probabilities. To do this, it is necessary to establish, for each transaction, which observable SAs *must have been completed* before the next SA is attempted. The point here is that a test failure within an SA must clearly be identified as reflecting a fault within that SA and not a simple consequence of a fault occurring upstream.

3.1.5 *Partitioning the Input Spaces of Software Actions*

Once the list of SAs has been established, it is necessary to consider the input spaces for each. For some SAs, the input space simply consists of one input. Other SAs need to work correctly for a variety of different inputs (such as card number). We wish to arrive at partitions of the input spaces of SAs which satisfy the requirements of exchangeability noted in Section 2.1. We do this by taking each SA and separating its inputs into groups such that the inputs in a group satisfy the following judgments: 1) The inputs have the same probability of failure and 2) the implication of observing a test for one input is the same for all other inputs in the node. To achieve this aim, we proceed through the scheme shown in Fig. 5. Typically, input spaces are partitioned according to certain *characteristics* (such as number of digits in a card number and whether or not the number starts with a zero). The partitions that we arrive at, together with the associated SAs, comprise the basic testable and observable features for our approach. We shall term these groupings *nodes*, as each will be represented as a distinct observable node for the BGM. Essentially, the tester's judgments are such that he should test at this level of detail and, so, the testing process will focus on making observations at this level of detail.

3.2 Advantages of Formal Structuring

The focus of the initial structuring is to arrive at this decomposition into characteristics and nodes. We will show shortly how the BGMs are constructed from these. However, irrespective of whatever approach is undertaken from this point, the initial structuring of the software testing to arrive at these characteristics and nodes (which are the logical consequences of the tester's judgments) is obviously valuable in providing the foundation for any formal testing process. For our case studies, we found that this proper structuring process caused testers to think more carefully about the nature of the software being tested and, indeed, led them to consider issues they had previously overlooked.

3.3 Converting Software Actions into BGMs

The initial structuring will have resulted in the definition of a number of observable nodes, which we term *domain* nodes, each consisting of an SA plus a distinct group of inputs. It is necessary next to convert these structures into BGMs according to shared characteristics. This is undertaken as described in Section 2.2. Additionally, we must handle any SAs which have been judged as being *related*, for example, because of similarity of transaction or because of a tester's judgment that SAs share general background operating mechanisms which, it is felt, are not worth modeling in more detail. We handle these as follows: We form a distinct *cluster* of SAs that are interrelated. That is, if SAs A, B, and C are related and SAs D and E are related, we work with distinct *clusters* {A, B, C} and {D, E}, where the clusters are independent in the sense that SAs within one cluster are independent of SAs in any other cluster. We think of each *cluster* as a composite SA which is capable of carrying out any or all of its component SAs and we introduce a marker characteristic to denote the component SAs. From this point, we simply treat the marker characteristic in the same way as any other input characteristic and partition accordingly.

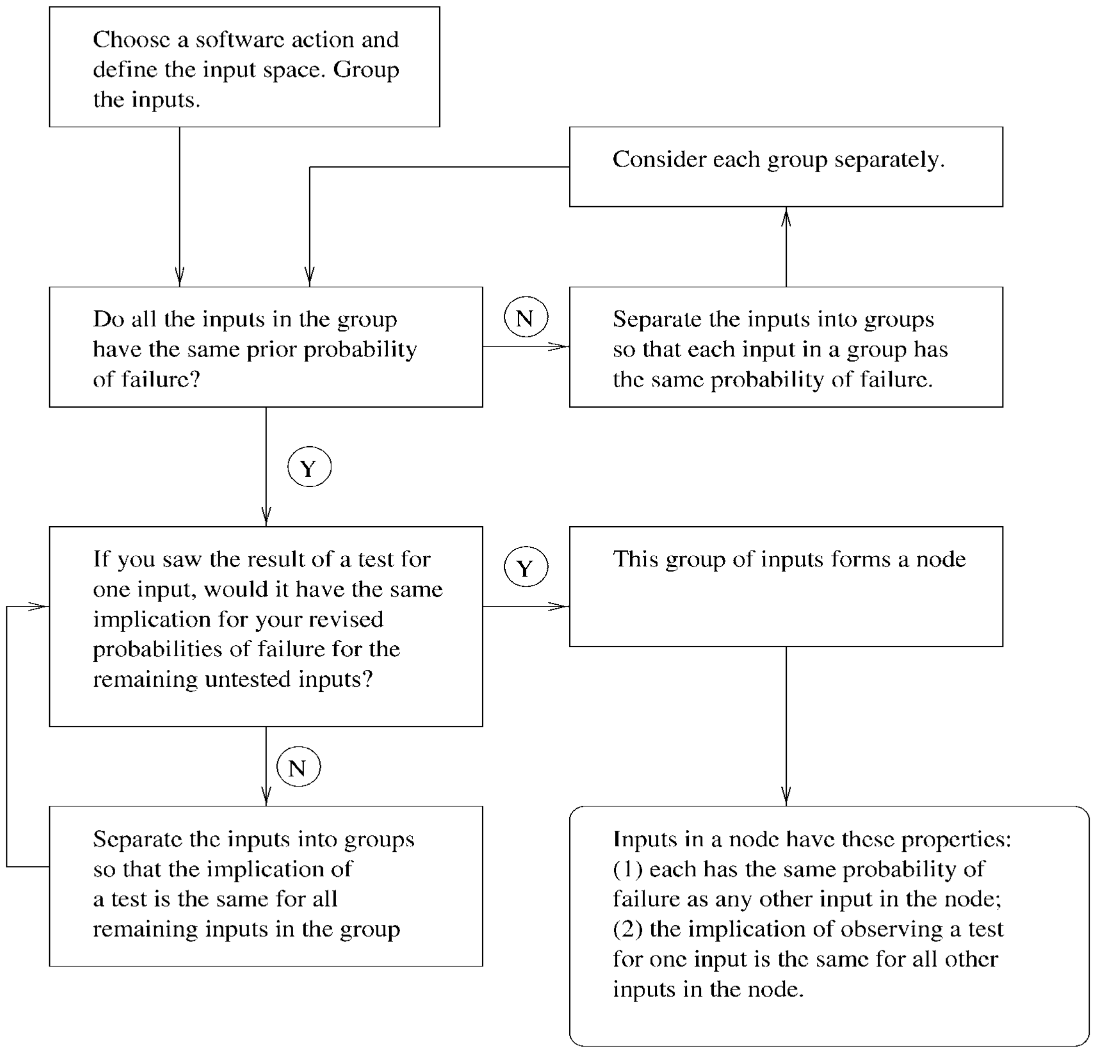
Choose a software action and define the input space. Group the inputs.
Consider each group separately.
Do all the inputs in the group have the same prior probability of failure?
N
Separate the inputs into groups so that each input in a group has the same probability of failure.
Y
If you saw the result of a test for one input, would it have the same implication for your revised probabilities of failure for the remaining untested inputs?
Y
This group of inputs forms a node
N
Separate the inputs into groups so that the implication of a test is the same for all remaining inputs in the group
Inputs in a node have these properties: (1) each has the same probability of failure as any other input in the node; (2) the implication of observing a test for one input is the same for all other inputs in the node.

Fig. 5. Partitioning the input space.

3.4 Assessing Prior Specifications

To enable modeling of the tester's judgments, the specifications described below are to be elicited. Here, and in Section 3.5, we briefly describe the methods we used for elicitation in our case studies.

3.4.1 Prior Distributions for Tests for Domain Nodes

For each domain node, the tester assesses q_S, the probability that, assuming a test for one input fails, all other inputs would also fail. If $q_S < 1$, we must select a probability distribution $f(p)$ for the proportion of the remaining inputs that would pass. To do so, we employ Beta distributions, which are convenient for this situation and allow sufficient flexibility.

3.4.2 Assessing Parental Influence

The graphical models that we construct can be interpreted as having parent nodes, where faults occur, and child nodes, where faults show up. Informally, if we assume that there are faults in the parent node (but no faults anywhere else), the implication of an arc is to show (using a probability) the potential of a parent node to induce a fault in the child node.

We treat a parent as being independently able to pass its fault to each child. An arc between parent and child nodes on the graph is labeled by a value which represents the probability that, if the parent has at least one fault, it will transmit a fault to the child. To specify the tables of conditional probabilities of child nodes given parent nodes, we employ the following scheme, termed noisy OR-gate (disjunctive interaction, [16]). Let $X = 0,\ X = 1$ be the events that node X represents software with no fault and at least one fault, respectively. For a node X with parents Y_i, $i = 1, \dots k$, specify probabilities

$$P(X = 0 | Y_1 = 0 \cap \dots \cap Y_i = 1 \cap \dots \cap Y_k = 0) = v_i$$

so that we think of $c_i = 1 - v_i$ as the probability that the parent Y_i transmits its fault. Then, for any configuration of parents Y, $P(X = 0|Y) = \Pi_{\{i:Y_i=1\}} v_i$. Finally, as all the causes of failure have been explicitly modeled, $P(X = 0|Y = 0) = 0$. The underlying assumption is that $Y_1, \dots, Y_k$ fail to transmit their faults independently.

3.4.3 Elicitation of Root Probabilities

In specifying root node probabilities, it may be difficult to arrive at precise judgments. However, calibration and sensitivity analyses will be run after the initial specification in order to achieve greater precision.

Information which is directly relevant to the elicitation process includes historic information, such as previous test results or test results for similar software areas. Further issues for the tester to consider include the following:

1. How complex is the code for the node? Perhaps, the greater the complexity, the higher the chance of a fault.
2. Comparatively, do you judge this code more or less reliable than other pieces of code for which you have a better idea of the reliability?
3. Is the code old code, which has passed many previous tests, or is it essentially new code to undertake new functionality, or is it somewhere in between?
4. What, historically, has been the reliability of the software author responsible for this piece of code? A well-respected software house or an entirely new provider?
5. Has similar code been tested in the past and found to be reliable or unreliable?

In Section 3.5, we briefly describe a general process for elicitating root node probabilities. This process is especially helpful when the tester's knowledge is quite vague. Elicitation is an important topic in applied Bayesian statistics [15] and we shall provide additional methods and guidance for elicitation related to the BGM approach in the future.

3.4.4 Measuring Consequences and Prioritization of Faults

Software reliability can be judged in many different ways; for example, the natural output of our BGM approach is to deliver the probability that a node, or any network formed from a group of nodes, has at least one fault. Another measure is the expected number of domain nodes containing at least one fault. One natural expression of the reliability of the modeled software is via a utility scale: The Bayesian approach provides a natural framework for combination of probabilities and utilities for decision making [6]. For example, every domain node either passes or fails a test and there is a consequence to the software owner if a particular node fails. Such consequences can be ascribed numerical values, perhaps reflecting the expected cost to the company of using the software given that it contains that fault.

A possible grading of faults is as follows: *Priority faults* have a catastrophic effect on software operations and would damage the company if they passed undetected. *Major faults* have an important effect on software operations and would cause the company some embarrassment if they passed undetected. *Minor faults* have a local effect and may or may not be fixed when they are discovered. *Cosmetic faults* would not normally be fixed unless a major revision of the software were undertaken. For a risk analysis, the domain nodes may be categorized according to fault priority and probabilities of failure and other summary measures calculated separately for each category. For a decision analysis, it would be more natural to combine measures on a utility scale. For example, the software owner may judge that the cost to the company for each fault category is, respectively, 100,000, 20,000, 1,000, and 1 unit of loss. Such utilities play an important role in designing test suites as they allow us to focus on the more relevant faults.

3.5 A Procedure for Probabilistic Specification of the BGM

We now describe a formal procedure for probabilistic specification of the BGM. A particular advantage of the procedure is that it can generate the quantification from vague knowledge, but is also sufficiently flexible to allow the quantification of detailed knowledge. The procedure has two stages. First, we obtain a reasonable approximation to the initial specification by asking the tester a few simple questions to assess the relative reliabilities of different areas of the software system. Second, we take the initial specification and refine it in the light of calibration analyses and other relevant information.

Typically, the SAs will be separated into several unconnected BGMs. We begin by asking the tester to rank all the BGMs in terms of expected reliability before testing. Next, each BGM has been constructed by partitioning SAs according to characteristics, such as number length and starting digit. Thus, the tester is asked to rank, within every BGM separately, characteristics in terms of expected reliability before testing. Next, each characteristic has been partitioned into groups of inputs; for example, the characteristic *number length* might be partitioned into 8-digit, 9-digit, and 10-16-digit numbers. For the root nodes for the constructed BGMs, we typically must assess reliability for general software problems handling the characteristic and for problems occurring for specific partitions of the characteristic. Thus, the tester is asked to rank, within every characteristic separately, the common and specific causes in terms of expected reliability before testing. Finally, any root nodes formed from the combinations of characteristics used to partition an input space are ranked among the common cause and specific partition quantities.

Once the ranking has been completed, we specify two initial tuning parameters: p reflects the general level of unreliability in the least reliable BGM and d reflects the general level of unreliability in the most reliable BGM compared to the most unreliable BGM. The parameter p equates to the probability that there is at least one fault in the most unreliable root node in the most unreliable BGM; dp equates to the probability that there is at least one fault in the most unreliable root node in the most reliable BGM. It is important to note that these are initial judgments: They provide a basis for further calibration and sensitivity analysis and there is full scope for revising the judgments in line with guidance from such analysis.

To quantify the BGM initially, proceed as follows: Label the n independent graphical models $G_1, G_2, \dots, G_n$. Label the n_i characteristics in model G_i as $C_{i1}, C_{i2}, \dots, C_{in_i}$. Next, label the n_{ij} partitions of characteristic C_{ij} as $L_{ij1}, L_{ij2}, \dots, L_{ijn_{ij}}$ so that L_{ijk} is the kth partition of the jth characteristic of the ith graphical model. Similarly, let r_i, r_{ij}, r_{ijk} be the ranks assigned to G_i, C_{ij}, L_{ijk}, respectively. Let p_{ijk} be the probability that there is at least one fault in the root node L_{ijk}. The aim of the elicitation is to establish the probabilities $\{p_{ijk}\}$ for the root nodes $\{L_{ijk}\}$. The approach is to take

$$\begin{aligned} p_i &= p - \tfrac{(r_i-1)p(1-d)}{n_i-1}, & & i=1,\dots,n. \\ p_{ij} &= p_i - \tfrac{(r_{ij}-1)p_i(1-d_i)}{n_{ij}-1}, & & i=1,\dots,n,\ j=1,\dots,n_i. \\ p_{ijk} &= p_{ij} - \tfrac{(r_{ijk}-1)p_{ij}(1-d_{ij})}{n_{ijk}-1}, & & i=1,\dots,n,\ j=1,\dots,n_i, \\ & & & k=1,\dots,n_{ij}. \end{aligned}$$

It remains to choose suitable values for d_i and d_{ij}. For characteristics, d_i corresponds to d for networks in the sense that it gives a rough indication for model G_i as to the relative reliabilities of the most and least unreliable characteristics within model G_i. For partitions within characteristics, d_{ij} corresponds to d for networks and d_i for characteristics and gives a rough indication for characteristic C_{ij} as to the relative ratio between most unreliable and least unreliable partition within characteristic C_{ij}. It is appropriate initially to choose $d_i = d$ for all i and $d_{ij} = d$ for all i, j. The probability for a root node thus consists of its reliability ranking within its local area, multiplied by a probability tuning parameter for that local area, modified by a tuning parameter expressing the difference in reliability between the least and most reliable root nodes for that local area.

3.5.1 *Refining and Calibrating the Model*

The ranking process, together with initial tuning parameters, provides an initial probability specification for the graphical model. It is straightforward to use the outputs of the models to help the tester realize the consequences of his judgments and thereby to revise or calibrate the BGM until the outputs are in accord with his judgments. For example, the model can calculate how many nodes are expected to contain at least one fault. Testers sometimes have a good feel for this quantity so that, if outputs from the constructed BGM disagree with this judgment, the BGM can be revisited in order to tune it. For example, it is simple to calculate an automatic scaling for the root nodes in the BGM so that the calculated output from the BGM exactly matches some specified number of nodes expected to contain faults. This process of calibration is to whatever level of detail is required by the tester. In addition to general calibration of the model to match the tester's judgments, the tester is free to add direct specifications wherever desired. For example, the tester may wish to construct a model where he has fairly vague knowledge about most of the software to be tested, but where he also has detailed knowledge for some parts of the software. This is straightforward to accommodate. Calibration taking into account possible test suites is discussed in Section 4.2.

4 Using the BGMs

4.1 Prior Analysis

Before any tests are run, we calculate prior descriptions of reliability across the software and, if desired, at the level of individual networks. These can be used for various purposes, including sensitivity and calibration analyses, obtaining systematic summaries of current software reliability, and to drive test design. The prior probabilities are, in general, only computable once the network has been defined and follow entirely from prior probabilities for root nodes, structural relationships, and probabilities on the arcs. Three kinds of summary measure that can be employed are: 1) the probability that the software contains at least one fault; 2) the expected number of domain nodes with low, medium, and high priority faults; and 3) a utility measure of the kind discussed in Section 3.4 which, informally, reflects the penalty to the company of releasing the software at a given point. The former pair can be used to focus risk analyses; for example, to direct testing toward high priority faults. The last is more naturally employed within a decision analysis framework. These computations must normally be performed using suitable software (Section 4.2).

4.1.1 *Sensitivity Analysis*

We undertake sensitivity analyses to explore how changes in the initial specification affect the prior (pretesting) and posterior (posttesting) summaries of reliability. Prior sensitivity analysis is useful in helping to provide a good initial model. Posterior sensitivity analysis, which we describe in Section 4.3, is useful when a specific test suite is in mind. The particular aspect of sensitivity analysis which is important prior to testing is the identification of areas where it is important to be precise and, so, where it may be worth putting extra effort into the process of specifying the model and, on the other hand, areas where quite crude specification is adequate. This is possible in a number of ways (see, for example, [11]). A simple approach is to examine the effect of increasing the specified probability of failure for every root node singly by comparing appropriate summary statistics before and after the change.

4.2 Using the BGM for Testing

The operational use of BGMs is fully described elsewhere [3], [10], [13], [16], [24]. Thus, we shall only briefly describe how updating the models operates. Instead, we shall focus on the ways in which we exploit the models in the context of software testing.

4.2.1 *Mapping of Domain Nodes to Tests*

Any test that we carry out will result in observation of a subset of the domain nodes across the various graphical models constructed to represent the software testing problem. Thus, it is necessary to map the tests to the domain nodes. Each test is likely to test several aspects of a problem. One simple way of forming the mapping is to list all the possible domain nodes (generally, all the child nodes from the graphical models), list all the possible tests, and form a matrix showing which tests result in an observation. This is straightforward when there already exists a given test suite. Otherwise, the focus will be on test design. To avoid unnecessary duplication, the mapping of domain nodes to tests is a mapping to potential tests and must be carried out at the SA level, i.e., the stage reached before partitioning the input space (Section 3.1). Which of the partitions is then observed by a given test is choosable and becomes one of the features addressed by test design.

4.2.2 *Computation*

All the results required by the approach are computationally straightforward using any package capable of performing the basic algorithms of Bayesian graphical modeling, such as *Netica*™ (Norsys Software Corporation, Vancouver, Canada) and *HUGIN*™ (Hugin Expert A/S, Aalborg, Denmark), which provide, *inter alia*, libraries of C routines providing BGM tools. Other suppliers provide similar BGM packages, see [3] for an overview. For our case studies, we wrote additional C surround programs.

4.2.3 Preposterior Analysis Assuming an Existing Test Suite

Preposterior analysis of a given test suite provides an assessment of the posterior reliability of the software, assuming that it passes the given test suite, and is the key to assessing the efficiency of a particular test suite. Thus, before running any tests, we can examine the implications for the models of all tests passing. It is straightforward to compute these implications for the various reliability measures we employ. Similarly, it is straightforward to explore the implications of test failures and alternative sequences of the proposed test suite so that we may use such assessments to compare and select between different potential test suites.

4.2.4 Simplifying and Sequencing Test Suites

Given a test suite, we may calculate the implication of any subset of tests and any sequence of tests. This enables a study of overlap of tests; for example, it may show that some tests do not increase utility and do not lead to reduction in the probability of at least one fault in any network, implying that, according to the tester's judgments, there is no value in running such a test at that moment in the test sequence, except perhaps to satisfy a coverage requirement. (Such apparently valueless tests can play an important diagnostic role for examining the assumptions of the model.) It is easy to sequence a given test suite; for example, for early selection of tests which test software areas with high remaining probability of failure.

4.2.5 Design and Analysis of Tests Additional to a Test Suite

Preposterior analysis provides measures of the reliability of the software given that it will be tested by a given test suite. If we assume that this test suite has not revealed any failures, an obvious question is to determine whether there remain substantial failure probabilities anywhere and, if so, how they might be tested. It is straightforward to use the outputs of the BGM to identify software areas which would contain substantial unreliability even if all of the tests in the test suite were to be successful and, so, to design extra tests to tackle these areas.

4.2.6 Design of New Test Suites

Just as we may design new tests to add to an existing test suite, the BGM approach can be used to design test suites *ab initio*, using whatever criteria are desired. Some of the technical issues are as follows: Typically, we choose criteria based on probability or utility calculations, which should take account of physical time and cost constraints, and management risk policies. For example, managers may wish to specify several fixed levels of software reliability (in terms of a performance measure such as probability of failure or utility) for which the test design process would yield, for each level of reliability, a test suite, together with its costs, from which managers could choose.

A simple approach is to choose tests in order to reduce the probability of at least one fault remaining in the software being tested by considering all possible tests in turn and assessing the implication for the model assuming a successful test for each test singly. This will produce one (or more) tests with the biggest gain in the chosen performance measure. This test is selected as the first test to be run. We continue this procedure iteratively by choosing, at each stage, the best remaining test and then repeating the procedure until the desired criterion has been achieved. This simple algorithm may not produce an optimal test design, but should produce one which is near optimal.

More advanced criteria and other technical considerations are the subject of ongoing research; for example, whether tests are chosen to take account of the difficulty of fixing any faults arising or to take account of faults which would prevent further testing. Test design must take account of constraints such as sequencing. It is simple to update the BGM using composite tests which do take account of sequencing and, in principle, easy to take account of batch constraints for test design. A further consideration in efficient test design is that it is useful to be able to sequence tests according to some criterion. For example, it may be cost-effective to design a sequence of tests so that the probability of finding faults during early tests is highest. This is due to the need for partial retesting, as guided by the graphical model, once faults are found.

There are obvious benefits from such a test design process allied to the Bayesian graphical modeling approach. Not only do we have all the advantages of the clear probabilistic representation of the expert and other knowledge, but also the careful analysis given by the test design process should arrive at smaller test suites with better coverage.

4.3 Posterior Sensitivity Analysis

It is important to explore sensitivity in relation to the testing process in particular because it helps to inform decisions as to when the software will be ready for release, given a specific test suite. It is a general feature of Bayesian analysis that differences in prior beliefs are often largely resolved by observing data and what matters is the effect of changes in prior belief on our posterior probability that the software is ready for release.

Consider the situation where there is a particular test suite envisaged, either historic or to be generated automatically using the BGM, and where we wish to examine the sensitivity of the initial specification in relation to this test suite. A simple approach, based on one-off changes to root nodes is as follows:

1. Specify and calibrate the model.
2. Apply the given test suite to the model and calculate the desired posterior summaries, assuming that all tests are successful.
3. Leaving all other root nodes unchanged, for each root node in the model, in turn, we take the probability of at least one fault and alter it on an appropriate scale.
4. We then apply the given test suite to the altered model and calculate the desired posterior summaries.
5. The differences between the posterior summaries for the initial and altered models measure the sensitivity of our conclusions to the probability specification for the altered root node.

We can also assess sensitivity globally by varying tuning parameters such as p and d (Section 3.5) and measuring the changes in the outputs of the BGMs. Other approaches to measuring sensitivity to prior specifications are discussed, for example, in [11].

5 Case Studies

The case studies we use to illustrate the approach are based on a larger ongoing study being undertaken for a major UK company. Some details have been altered or fictionalized to preserve confidentiality. For each case study, we briefly describe overall aspects of the implementation of our approach.

5.1 Credit Card Database Management

5.1.1 Software Area

The software being tested relates to all the software required for operations on products which can be taken to be credit cards. For example, the company has a database which contains customer details such as name and address, credit card number, credit card brand, current amount owing on the card, and so forth. A second database handles online referrals and, so, must hold details such as credit card number, credit limit, amount owing, and any restrictions. For example, a card may be suspended for fraud or because a credit limit has been exceeded. There are various further databases. Additionally, some of these databases may be physically disaggregated; for example, there are several online referral databases for specific regions or countries. There are typically several databases involved not only for operational reasons, but also because databases which involve personal or sensitive details (such as PIN) need to be isolated as far as is practicable. The testing process must test not only processing that occurs within databases, but must also test that databases communicate correctly.

5.1.2 Software Novelty

The software being tested represented a major update of existing software to provide new functionality and fix faults.

5.1.3 Tester Expertise

The senior tester had considerable expertise in the functionality of the software and his testing team had already constructed (but not run) a test suite.

5.1.4 Scale

This case study was medium scale. There were 16 separate major testable areas. The initial structuring process led to the tester identifying 168 different domain nodes. These 168 nodes were contained within 54 distinct (independent) BGMs. The tester's test suite consisted of 233 tests, which were then mapped to the domain nodes.

5.1.5 Sequential Analysis

Analysis of the tester's test suite, analyzed in the order the tester intended the tests to be run, showed no gain in information through running the last 57 tests. (Some tests are run purely for coverage purposes.) The sequential analysis of this test suite is shown in Fig. 6 and illustrates that it is a consequence of the tester's judgments that the vast majority of the tests are not expected to remove uncertainty about the reliability of the software. The tester found this initially surprising.

5.1.6 Test Efficiency

The tester's original test suite reduced the probability of at least one fault remaining from 0.336514 to 0.011580 and, so, reduces the initial probability by 96.56 percent. Of the 233 original tests, 66 turned out to be completely redundant in the sense that their test coverage is fully covered by (some combination) of the remaining 167 tests. Of the 168 observables, 55 were not tested at all.

5.1.7 Test Selection and Scheduling

It is simple to select which subset of the tests in an existing test suite is most efficient at reducing the probability of there being at least one fault remaining in the software. Of the original 233 tests, the best 11 tests reduced the prior probability of at least one fault by 95.07 percent. The next best 156 tests further reduced this probability by 1.49 percent. The remaining 66 tests do not further reduce this probability. This rescheduling ignores some of the sequencing constraints, but there is no practically significant difference in doing so.

5.1.8 Test Design

It was straightforward to design extra tests to cover gaps in the coverage provided by the existing test suite. For example, we designed one extra test which had the effect of reducing the residual probability of failure by 57.1 percent. The senior tester agreed that this would have been a useful test to run.

5.1.9 Conclusion

Numerical summaries of the efficiency of the tester's test suite, in probability and utility measures, are given in Table 1, together with the impact of one extra new test. Overall, the tester designed a fairly comprehensive test suite. Given that all these tests run successfully, the probability of at least one failure remaining is 3.44 percent of what it was before testing. However, the BGM approach could have 1) helped the tester choose far fewer tests with at least equal coverage, 2) helped the tester design extra tests to fill gaps, and 3) helped the tester to schedule the tests optimally in order to maximize the chance of picking up faults at an early stage of testing. When the tester's test suite was run, faults were found in the areas for which the BGMs had correspondingly relatively high probability of failure at that point in the test suite. Finally, the BGM approach provides probabilistic information concerning this software's reliability which allows managers to decide whether or not it is reasonable to release the software or whether to allocate more resources to further testing.

5.2 Record Renumbering

5.2.1 Software Area

The software to be tested is intended to carry out renumbering of all records in a database because the present number of digits is insufficient to meet an expansion in customer demand.

5.2.2 Software Novelty

Such renumbering has been carried out in the past for different databases, but this is newly created software which will be used once.

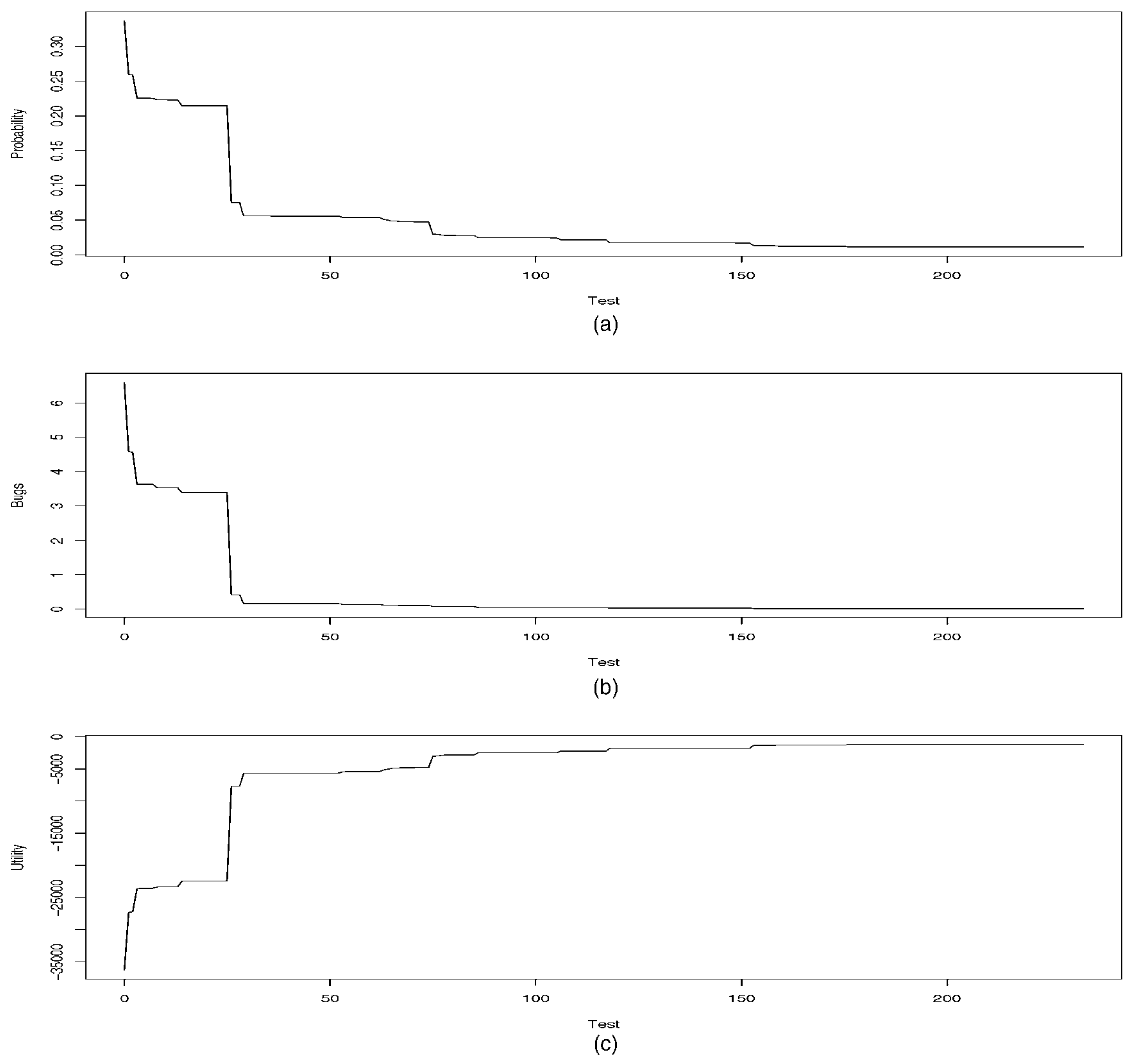

Fig. 6. Sequential analysis of the tester's test suite, analyzed in the order the tester intended the tests to be run. The utility scale is as described in Section 3.4.4.

5.2.3 Tester Expertise

The sole tester had no prior expertise in the reliability of the software to be tested and only vague notions as to the software operations needed for the software to do what it was intended to do.

5.2.4 Scale

This case study was small scale, with three separate major testable areas. The tester identified 40 domain nodes, contained in three independent BGMs. In this case study, although the tester must choose an input from a possibly large range of allowable inputs, each such test is expected to fully test the domain node. The tester's test suite consisted of 20 tests, constructed (but not run) before the BGMs were elicited. The elicitation process described in Section 3.5 was used to quantify the models and the expected number of faults remaining was chosen to match the models' outputs to the tester's judgments.

5.2.5 Sequential Analysis

Sequential analysis of the tester's test suite, analyzed in the order the tester intended the tests to be run, showed no gain in information through running the last eight tests.

5.2.6 Test Efficiency

The tester's original test suite reduced the probability of at least one fault remaining from 0.801101 to 0.209129 and, so, reduces the original probability by 73.89 percent. Of the 20 original tests, at least 11 are completely redundant in the sense that their test coverage is fully covered by (some combination) of the remaining nine tests. It is simple to analyze the scheduling for the tester's test suite, but this is of no further interest for this case study as a replacement test suite can be designed automatically.

5.2.7 Test Design

For this case study, it was possible to design, automatically, *ab initio*, an efficient test suite, given only the tester's basic specification of the operations to be tested and the BGM so constructed. The automatically designed test suite contains only six tests, but fully tests the software. The tester agreed that the automatically designed test suite was more efficient at testing the software and agreed that it tested an area he had missed. The tester had not originally considered that this area needed testing, but did agree with hindsight that it was sensible to test this area, which came to light only through the initial BGM structuring process.

TABLE 1
Numerical Summaries of the Reliability of the Software before Testing, after One Successful Test, after Running the Full Test Suite, and after One New Extra Test

	Probability of at least one fault				Utility attached to consequence of failure			
Area	Before Testing	After 1 test	After 233 tests	After 1 new test	Before Testing	After 1 test	After 233 tests	After 1 new test
M_1	.002796	.002292	.000000	.000000	-279.55	-229.17	0.00	0.00
M_2	.000795	.000775	.000075	.000075	-76.11	-75.10	-6.70	-6.70
M_3	.000058	.000058	.000028	.000028	-5.78	-5.80	-2.83	-2.83
M_4	.004493	.000000	.000000	.000000	-449.34	0.00	0.00	0.00
M_5	.007485	.005000	.005000	.000000	-748.53	-500.00	-500.00	0.00
M_6	.016390	.011259	.000000	.000000	-1639.02	-1125.91	0.00	0.00
M_7	.003576	.003576	.000000	.000000	-357.56	-357.56	0.00	0.00
M_8	.007885	.007885	.003202	.003202	-788.51	-788.51	-320.23	-320.23
M_9	.002197	.002197	.000502	.000502	-203.83	-203.83	-50.24	-50.24
M_{10}	.000581	.000151	.000000	.000000	-58.10	-15.07	0.00	0.00
M_{11}	.006624	.005849	.000267	.000266	-663.47	-585.67	-26.71	-26.62
M_{12}	.009781	.009781	.000000	.000000	-978.06	-978.06	0.00	0.00
M_{13}	.005696	.001478	.000104	.000104	-522.00	-107.86	-10.08	-10.08
M_{14}	.009799	.008293	.002445	.002445	-979.90	-829.26	-244.53	-244.53
M_{15}	.003815	.000000	.000000	.000000	-381.53	0.00	0.00	0.00
M_{16}	.006382	.000000	.000000	.000000	-639.99	0.00	0.00	0.00
All	.085095	.057117	.011578	.006610	-8858.75	-5860.26	-1162.48	-662.39

Summaries are given for each of 16 distinct software areas and overall.

5.2.8 Conclusion

The tester designed a reasonably comprehensive test suite. Given that all the tests run successfully, the probability of at least one failure remaining is 26.11 percent of what it was before testing. However, the BGM approach automatically designed a test suite with fewer tests (six, compared to 20) with better (in fact, full) coverage. Further, through providing a framework for capturing knowledge about the structure of the software being tested, the BGM approach would have led the tester to consider testing areas which he omitted to consider. With regard to sensitivity analysis, none is required for this case study as, according to the tester's judgment, 100 percent coverage can be achieved. Thus, the prior specification is relevant only at the structural level.

6 Discussion

We have described an approach to the probabilistic modeling and analysis of software systems. In practice, the models that we produce may be complex. However, they will never be more complex than the considerations which the software tester must, in any case, bring to bear in assessing software reliability as all of the distinctions that are introduced into the model are there because the tester judged these distinctions to be important. All that we have done is, first, to draw a picture identifying the various distinct groups of tests which could be run and, second, to require a prior quantification of the uncertainties in the diagram. It is, in any event, good practice to carry out the first stage in this process when planning a software test suite. As for the quantification, in many cases, it will be comparatively straightforward to rank the relative probabilities for the various sources of fault. When we place precise numbers on the diagram, it may be the case that the model analyses will be relatively insensitive to variations in the prior values, provided the prior constraints are obeyed; in which case, we may be confident that the software has been tested to a high degree of reliability. Alternatively, we may find that there are plausible prior inputs for which we are unable to conclude that the software is reliable given the test results. In such cases, we must either model more carefully or test more extensively. The modeling process, combined with a careful sensitivity analysis, forces us to consider how much we can defend our judgments of software reliability. The alternative, namely, making the same judgments of software reliability without a framework for analyzing the uncertainties, is far more likely to lead to bad judgments. Further, having constructed the model, we gain the various advantages resulting from a probabilistic analysis, such as automatic generation of good test suites, a quantified approach to risk analysis for the software release, and so forth.

The model that we have described exploits the expert judgments of the tester. If these judgments are overly simplistic, then the model will not give a good representation of the faults in the software. In such cases, the model will still improve on the analysis of the tester, but with the additional advantage that the various assumptions of the tester may be explicitly scrutinized within the model by the model diagnostics for the BGM. These diagnostics are based on discrepancies between the actual test behavior and the predicted behavior according to the model; for example, the prediction of few faults in a given subarea might be contradicted by observation of several faults in that area.

Finally, note that the BGM approach captures, as far as is deemed practicable, the expertise of the tester and maintains it in usable form as a knowledge base. This represents a considerable resource to the software owner, who is routinely faced by the problem of the expertise of a tester being lost due to everyday practicalities such as personnel changes.

The relevance of the BGM approach to support software testing, from a management perspective, is addressed in [18], which also considers the circumstances under which this approach has been developed. In a future paper, we shall address how we may assess the viability of the approach in terms of the scale and complexity of the software to be tested.

Acknowledgments

The authors gratefully acknowledge the provision of funding for much of this work from British Telecommunications plc and input on its industrial direction. The probabilities and other results shown in this paper are illustrative and do not reflect the company's actual judgments. They also wish to thank the referees for their helpful comments.

References

[1] M. Bouissou, F. Martin, and A. Ourghanlian, "Assessment of a Safety-Critical System Including Software: A Bayesian Belief Network for Evidence Sources," *Proc. Ann. Reliability and Maintainability Symp., RAMS '99*, 1999.

[2] A. Burr and M. Owen, *Statistical Methods for Software Quality.* London: Thomson, 1996.

[3] R.G. Cowell, A.P. Dawid, S.L. Lauritzen, and D.J. Spiegelhalter, *Probabilistic Networks and Expert Systems.* New York: Springer, 1999.

[4] S.R. Dalal and C.L. Mallows, "Factor-Covering Designs for Testing Software," *Technometrics*, vol. 40, pp. 234–243, 1998.

[5] P.G. Frankl, R.G. Hamlet, B. Littlewood, and L. Strigini, "Evaluating Testing Methods by Delivered Reliability," *IEEE Trans. Software Eng.*, vol. 24, pp. 586–601, 1998, Erratum: vol. 25, p. 286, 1999.

[6] S. French and D. Rios Insua, *Statistical Decision Theory.* London: Arnold, 2000.

[7] D. Hamlet, "Are We Testing for True Reliability?" *IEEE Software*, pp. 21–27, July 1992.

[8] D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Software Eng.*, vol. 16, pp. 1402–1411, 1990.

[9] R.M. Hierons and M.P. Wiper, "Estimation of Failure Rate Using Random and Partition Testing," *Software Testing, Verification, and Reliability* vol. 7, pp. 153–164, 1997.

[10] F.V. Jensen, *An Introduction to Bayesian Networks.* London: UCL Press, 1996.

[11] U. Kjærulff and L.C. van der Gaag, "Making Sensitivity Analysis Computationally Efficient," *Proc. 16th Conf. Uncertainty in Artificial Intelligence*, 2000.

[12] L. Kuo, "Software Reliability," *Encyclopedia of Statistical Sciences*, S. Kotz, ed., update vol. 3, pp. 671–680, 1999.

[13] S.L. Lauritzen, *Graphical Models.* Oxford: Clarendon, 1996.

[14] M. Neil and N. Fenton, "Predicting Software Quality Using Bayesian Belief Networks," *Proc. 21st Ann. Software Eng. Workshop NASA/Goddard Space Flight Center*, 1996.

[15] A. O'Hagan, "Eliciting Expert Beliefs in Substantial Practical Applications," *The Statistician*, vol. 47, pp. 21–35, 1998.

[16] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* San Mateo, Calif.: Morgan Kaufmann, 1988.

[17] F. Redmill, "Why Systems Go Up in Smoke," *The Computer Bulletin*, pp. 26–28, Sept. 1999.

[18] K. Rees, F.P.A. Coolen, M. Goldstein, and D.A. Wooff, "Managing the Uncertainties of Software Testing: A Bayesian Approach," *Quality and Reliability Eng. Int'l*, vol. 17, pp. 191-203, 2001.

[19] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Trans. Software Eng.* , vol. 22, pp. 529–551, 1996.

[20] G. Rothermel and M.J. Harrold, "Empirical Studies of a Safe Regression Test Selection Technique," *IEEE Trans. Software Eng.*, vol. 24, pp. 401–419, 1996.

[21] N.D. Singpurwalla and S.P. Wilson, "Software Reliability Modeling," *Int'l Statistical Rev.*, vol. 62, pp. 289–317, 1994.

[22] N.D. Singpurwalla and S.P. Wilson, *Statistical Methods in Software Engineering: Reliability and Risk.* New York: Springer, 1999.

[23] C. Smidts and D. Sova, "An Architectural Model for Software Reliability Quantification: Sources of Data," *Reliability Eng. and System Safety*, vol. 64, pp. 279–290, 1999.

[24] D.J. Spiegelhalter, A.P. Dawid, S.L. Lauritzen, and R.G. Cowell, "Bayesian Analysis in Expert Systems," *Statistical Science*, vol. 8, pp. 219–283, 1993.

[25] E.J. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies," *IEEE Trans. Software Eng.*, vol. 17, pp. 703–711, 1991.

[26] E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Trans. Software Eng.*, vol. 6, pp. 236–246, 1980.

[27] T. Yamaura, "How to Design Practical Test Cases," *IEEE Software*, pp. 30–36, Nov.-Dec. 1998.

David A. Wooff is the director of the Statistics and Mathematics Consultancy Unit, University of Durham, United Kingdom. He is a chartered statistician and has lectured in statistics at Durham University since October 1991. His research interests include Bayes linear statistics, applied industrial statistics, and Bayesian approaches to software testing. He has been lead or coinvestigator for a number of commercial and UK Engineering and Physical Sciences Research Council research projects.

Michael Goldstein is a professor of statistics at the Department of Mathematical Sciences, University of Durham, United Kingdom. His research interests are concerned with foundations, methodology, and applications of the Bayesian approach to statistics and decision analysis. In particular, he has developed Bayes linear approaches which simplify both belief specification and analysis and, therefore, allow the extension of Bayesian methodology to increasingly complex problems, one such being uncertainty analysis for computer models of large scale physical systems, for which he has received both commercial and UK Engineering and Physical Sciences Research Council support.

Frank P.A. Coolen received the MSc and PhD degrees from Eindhoven University of Technology, The Netherlands. He is a reader in statistics in the Department of Mathematical Sciences, University of Durham, United Kingdom. His research interests include foundations of statistics, reliability theory, and Bayesian graphical models.