Bayesian Graphical Models for High Complexity Testing: Aspects of Implementation

D.A. Wooff, M. Goldstein, F.P.A. Coolen
Department of Mathematical Sciences
University of Durham
Durham, DH1 3LE, UK

March 23, 2017

Synopsis

The Bayesian Graphical Models (BGM) approach to software testing was developed in close collaboration with industrial software testers. It is a method for the logical structuring of the software testing problem, where focus was on high reliability final stage integration testing. The core methodology has been published and is briefly introduced here, followed by discussion of a range of topics for practical implementation. Modelling for test-retest scenarios is considered, as required after a failure has been encountered and attempts at fault removal have been made. The expected duration of the retest cycle is also considered, this is important for planning of the full testing activity. As failures often have different levels of severity, we consider how multiple failure modes can be incorporated in the BGM approach, and we also consider diagnostic methods. Implementing the BGM approach can be a time-consuming activity, but there is a benefit of re-using parts of the models for future tests on the same system. This will require model maintenance and evolution, to reflect changes over time to the software and the testers' experiences and knowledge. We discuss this important aspect which includes consideration of novel system functionality. End-to-end testing of complex systems is a major challenge which we also address, and we end by presenting methods to assess the viability of the BGM approach for individual applications. These are all important aspects of high complexity testing which software testers have to deal with in practice, and for which Bayesian statistical methods can provide useful tools. We present the basic approaches to these problems, most of these topics will need to be carefully extended for specific applications.

Keywords: Bayesian graphical models; Diagnostics; End-to-end testing; Expert judgement; Model maintenance and evolution; Multiple failure modes; Software reliability; Software testing; Statistical methods; Test design; Test-retest; Viability.

1 The Bayesian Graphical Models approach to testing: brief overview

Bayesian Graphical Models (BGM), also known as Bayesian networks and a variety of further terms, are graphical representations of conditional independence structures for random quantities, which have proven to be powerful tools for probabilistic modelling and related statistical inference

[6]. The Bayesian Graphical Models approach for supporting software testing was developed by the authors in close collaboration with an industrial partner, the core methodology was presented in Wooff, et al. [7]. Some additional aspects of this approach were briefly discussed by Coolen, et al. [3]. The BGM approach presents formal mechanisms for the logical structuring of software testing problems, 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 are dynamic representations of the software testing problem. They may be used to answer what-if questions, to provide decision support to testers and managers, and to drive test design. The models capture the knowledge of the testers for further use. The main ingredients of the BGM approach are briefly summarized in this section and illustrated via part of a substantial case study [7].

Suppose that the function of a piece of software is to process an input number, e.g. a credit card number, in order to perform an action, and that this action might be carried out correctly or incorrectly. The tests that might be run correspond to choosing various numbers and checking that the *software action* is performed correctly for each number. Usually, it will not be possible to check all inputs, but instead one checks a subset from which one, hopefully, may conclude that the software is performing reliably. This involves a subjective judgement about the functionality of the software over the collection of inputs not tested, and the corresponding uncertainties. One must choose whether to explicitly quantify the uncertainties concerning further failures given test results, or whether one is content to make an informal qualitative judgement 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 expert carrying out the analysis [1].

In the BGM approach, the testers' uncertainties for software failure are quantified and analysed using subjective probabilities, which involves investment of effort in thinking carefully about prior knowledge and modelling at a level of detail appropriate for the project. The rewards of such efforts include probabilistic statements on the reliability of the software taking test results into account, guidance on optimal design of test suites, and the opportunity to support management decisions quantitatively at several stages of the project, including deciding on time and cost budgets at early planning stages.

The simplest case occurs when the tester judges all possible test results to be exchangeable, which implies that all such test results are judged to have the same probability of failing, and that observing each test result gives the same information about all other tests. Qualitatively, exchangeability is a simple judgement to make: either there are features of the set of possible inputs which cause the tester to treat some subsets of test outcomes differently from other subsets or, for him, the collection of outcomes is exchangeable. Some practical aspects, including elicitation of such judgements, are discussed later. Of course, testers may not be 'correct' in their judgements, but attempts to model the software in far greater detail, relying on many different sources, tend to be very time-consuming and are not part of common practice in many software testing situations. The approach discussed here starts with the judgements of the testers, and helps in selecting good test suites in line with those judgements. An important further aspect is the use of diagnostic methods and tests, aimed at discovering discrepancies between those judgements and the actual software performance. Later, aspects of such diagnostic testing are also discussed. In the example, suppose that the software is intended to cope with credit cards with both short (S) and long (L) numbers, and that the tester judges test success to be exchangeable for all short numbers and exchangeable for all long numbers. For example, it might be the case that dealing with long numbers is newly added functionality. In addition, suppose that the tester judges test success to be exchangeable for numbers starting with zero (Z), and exchangeable for numbers starting with a non-zero digit (X).

Figure 1 is a BGM, in reduced form, reflecting such judgements. Each node represents a random

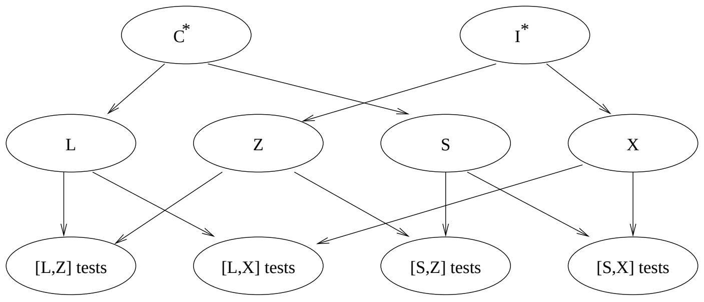

Figure 1: BGM (reduced form) for a software problem with two failure modes.

quantity, and the arcs represent conditional independence relations between the random quantities [4, 6, 7]. There are four subgroups of tests, resulting from the combinations of number length and starting digit given the exchangeability judgements. For example, node '$[L, Z]$ tests', with attached probability, represents the probability that inputs of the type $[L, Z]$ lead to an output error when tested. Such an error could be caused by three different problems, namely general problems for long numbers (node 'L'), general problems for numbers starting zero (node 'Z'), or problems specific for this combination. This last cause is not explicitly represented in this reduced BGM, as it would be a single parent node feeding into the node '$[L, Z]$ tests', but for the sake of simplicity all nodes with a single child and no parents have been deleted. Similarly, the node C^* represents general problems related to length of the numbers, so problems in 'L' can be either caused by problems in C^*, or in a second parent node of 'L' (not shown), for problems specifically arising for long numbers. Upon testing, the random quantities represented by these nodes are typically not directly observable. Their interpretation is as binary random quantities, indicating whether or not there are faults in the software, particularly leading to failures when a related input is tested. Finally, the actual test inputs are also not represented by nodes in this reduced BGM; they would be included in the form of a single node corresponding to each input tested, feeding off the corresponding node at the lowest level presented in Figure 1.

To build such a model, exchangeability assumptions are required over all relevant aspects, and many probabilities must be assigned, both on all nodes without parents, and for all further nodes in the form of conditional probability tables, for all combinations of the values of the parent nodes. For larger applications, which typically included several thousands of nodes, methods to reduce the burden of specification to manageable levels are described in [7]. For example, the tester was asked to rank nodes according to 'faultiness', and then to judge overall reliability levels to assign probabilities consistent with these judgements. See Wooff, *et al.* [7] for more details, where in particular further theoretical, modelling, and practical aspects are discussed. After the test results have been observed, all these probabilities are updated by the rules of BGMs [4, 6], which provides an explicit quantification of the effect of the test results on overall reliability of the software system, within the subjectively quantified reliability model reflecting the judgements of the testers. Therefore one can assess the value of a test suite from this perspective, because either it will indicate faults, which must be fixed and followed by retesting, or one observes no failures and then has the decision as to whether to release the software. Thus, one may value the test suite according to the probability of no faults remaining, if one enters successful test results for all inputs of the test suite into the model.

The main case study presented in [7], where this method was applied in a project similar to management of credit card databases, involved a major update of existing software, providing new functionality and with some earlier faults fixed. It could be called a 'medium scale' case study; there were 16 separate major testable areas, and initial structuring by the tester led to identification of 168 different domain nodes (in Figure 1, there are 4 domain nodes, namely those at the lowest level presented), contained in 54 independent BGMs. A single test typically consisted of a variety of inputs spread over several of these BGMs. For example, one test could be 'change of credit card', which would consist of a combination of delete an existing card from the database and add a new card to the database with appropriate credit limits. The practical problem for the testers is optimal choice of the test suite: assessment of the information that may be gained from a given test suite is very difficult unless a formal method is used. In this case study, the tester had originally identified a test suite consisting of 233 tests. Application of the BGM approach revealed that the best 11 tests of these reduced the prior probability of at least one fault remaining by about 95%, and that 66 tests could not add any further information, according to the tester's judgements, assuming that the other tests had not revealed any failures. The BGM approach was also used to design extra tests to cover gaps in the coverage provided by the existing test suites. One extra test was designed which had the effect of reducing the residual probability of failure by about 57%, mostly because one area had been left completely untested in the originally proposed test suite. The value of this test was agreed by the senior tester. Subsequently, the authors have developed a fully automated approach for test design, providing test suites without using a tester's test suite as input. This leads to even better designs, and allows the inputs to be tested in sequences according to a variety of possible optimisation criteria, where for example aspects of test-fix-test can be taken into account, e.g. minimising total expected test time including time to fix faults shown during testing and the required retesting, as is discussed later.

A second case study [7] involved software for renumbering all records in a database, required because the present number of digits was insufficient to meet an expansion in customer demand. Similar actions had been carried out in the past, but this software was newly created and to be used only once. The sole tester had no prior expertise in the reliability of this software and only vague notions about the software operations. This was a 'small scale' case, with three separate major testable areas represented by three independent BGMs, with a total of 40 domain nodes. It was assumed that, although the tester must choose an input from a possibly large range of allowable inputs, each such test was 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 BGM analysis showed that this original test suite, if no failure were revealed, would reduce the probability of at least one fault remaining from 0.8011 to 0.2091, so reducing the original probability by about 74%. Of these 20 original tests, at least 11 were shown to be completely redundant, under the elicited assumptions in the BGM approach, in the sense that their test coverage was fully covered by (some combinations of) the remaining nine tests. The BGM approach was used to automatically design an efficient test suite, given only the tester's basic specification of the operations to be tested and the BGM so constructed. This test suite contained only six tests, but would fully test the software according to the judgements provided by the tester. Consequently, the tester agreed that the automatically designed test suite was more efficient at testing the software, and that it tested an area he had missed. The tester had not originally considered that this area needed testing, but agreed that it was sensible to test it, which came to light only through the initial BGM structuring process. It should be emphasized that, although several of the 20 originally suggested tests were redundant, they are of value from the perspective of diagnostics for the elicited model and probabilities, as is discussed later.

The following sections focus on aspects of implementation of the BGM approach, and corresponding management issues, in addition to those discussed by Wooff et al [7]. These general discussions are based on the industrial collaboration and motivation is sometimes given in terms of database-related software functions, in line with the case studies mentioned above.

2 Modelling for the test-retest process

One of the most challenging aspects of the testing process is modelling and design in the sequence where software is tested, faults are, supposedly, fixed and the system is partially retested, to protect against new errors that might have been introduced in the fixing process.

Full modelling of the complete retest cycle for all possible failures that we might possibly observe may be very complex, due to the many different types of fault that we may encounter. Further, much of this detailed modelling will usually be wasted, as the great majority of tests will not fail. However, choices as to the amount of retesting that we must do will be key determinants of overall testing costs and thus of our ability to complete testing within budgetary constraints, usually against strict deadlines. Therefore, we cannot avoid some form of retest modelling, to guide the design and sequencing of the test suite through estimation of the cost and time to completion of the full testing process. We suggest a pragmatic simplification for retest modelling which should roughly correspond to the judgements of the tester, and give sensible guidelines both for choosing the test suite and assessing total budget implications, with the understanding that for the, hopefully small, number of faults which are actually found, we may choose to reconsider the implications of the individual failures.

2.1 Retest modelling

Suppose that we have already carried out test suite $T_{[r-1]} = T_1, ..., T_{r-1}$, all tests proving successful. We now carry out test T_r, and an error occurs on a test attached to domain node J. To simplify what follows, let us suppose that the first test we run on retest is the test T_r which previously failed and that this test is successful (as otherwise, we would return the software for further fixing and then repeat the procedure). It does seems reasonable to select T_r as our first test, and this avoids unnecessary complications in the following account. (Of course, in practice, we might need to run some of the other tests in order to carry out the test T_r but this does not affect the general principle of our approach.) What we need to decide is whether further errors have been introduced into the model during the software fix, and to update our probability model accordingly thus revealing which tests need to be rerun. A simple way to model new errors is to divide such errors into two types:

- *similar* errors to the observed error (for example, errors in nodes which are linked to the node J in the model);
- *side-effect* errors (i.e. errors that could occur anywhere in the model). These side-effect errors are errors specifically introduced by the fixing of another bug, and which in principle we can find by further appropriate testing.

A simple way of handling such errors is to return the probability description to the state that it was in before testing began.

- If there are *similar* errors then we cannot trust the results of the subset of previous tests with outcomes linked to the node with the error, and so these results must be ignored.
- If there are *side-effect* errors then we cannot trust any of the previous test results.

We might combine this approach with a more sophisticated analysis of the root probabilities. For example, if we find many errors, then we might feel that we should have raised the prior probability of errors in all root nodes by some fixed percentage. Doing so does not change the general idea of the remodelling.

2.2 Consequences of a test failure

We denote by T_S the subset of the previous test suite $T_{[r-1]}$ which tests nodes which are *similar* to the node J which failed on test T_r. We let T_G be the corresponding subset of remaining tests in $T_{[r-1]}$ which tests those nodes which are *dissimilar* to node J. We must take into account three possibilities.

R_1**:** The error detected by T_r is fixed and no new errors are introduced.

R_2**:** The error detected by T_r is fixed, but, possibly, *similar* errors have been introduced.

R_3**:** The error detected by T_r is fixed, but, possibly, *side-effect* errors have been introduced, possibly together with further *similar* errors.

Consideration of tests that result in failures in multiple unrelated nodes is left as a topic for future research. A simple way of proceeding is to postulate that these three scenarios have the following consequences.

R_1**:** Our beliefs about the model are just as though we had carried out on the original model test suite $T_{[r]} = T_1, ..., T_r$ and all tests had been successful.

R_2**:** Our beliefs about the model are just as though we had carried out test suite T_G and test T_r successfully, but we have not carried out any of the tests in T_S.

R_3**:** Our beliefs about the model are just as though we had carried out on the original model the single test T_r successfully, but we have not carried out any of the other tests.

These consequences are simplifications in the following sense. For scenario R_2, the subset of tests T_S will be testing many nodes which are *dissimilar* to the failure node, in addition to testing the failure node and nodes similar to it. For these *dissimilar* nodes, the beliefs which pertain to them at the time following the completion of test suite $T_{[r]}$ may be more appropriate than the corresponding prior beliefs. For scenario R_3, we might believe it appropriate to handle *similar* and *dissimilar* nodes distinctively; dealing with these refinements, which offer some benefits, is left as a topic for future research.

2.3 Further specification

In order to handle this problem, we need to understand the relative probability of each of these three possibilities. Therefore, to complete the specification we introduce the three numbers p_1, p_2, p_3 to represent the probability that we assign to the three possibilities R_1, R_2, R_3, given that test T_r succeeded. We suppose that we have already tested T_r successfully as otherwise we would need three more possibilities corresponding to T_r failing which would unnecessarily complicate the model, as we would not carry out further tests given failure of T_r, but would instead return the software for further correction.

$$p_i = P(R_i), \quad i = 1, 2, 3; \quad p_1 + p_2 + p_3 = 1.$$

2.4 Prior specification

It may be appropriate that these three values, and the division into *similar* and *side-effect* errors, can be assigned in advance for each possible node to be tested. For example, such a specification might be guided by knowledge of the abilities of the software engineer responsible for fixing the software for which the error occurred. If there are n nodes in the model being tested, this would require at most a specification of $3n$ probabilities and n divisions into error types, although there will usually be many simplifications to be exploited.

Otherwise, it may be appropriate simply to specify in advance three basic probabilities p_1, p_2, p_3 which we use for each node to be tested, along with a simple rule of thumb for distinguishing the test subsets T_S and T_G. For example, a simple rule is to assign those nodes which are connected to the observed failure node in the BGM as *similar*, and the remainder as *dissimilar.* Both the full specification and the simple specification allows an analysis of expected completion time for the test suite. Such prior specification is mostly useful to provide for order-of-magnitude appropriate calculations, and for simulation purposes.

2.5 Wait-and-see specification

Instead of specifying the probabilities and divisions into *similar*, and *side-effect* a priori, it is better to wait until we see a particular test failure and specify the probabilities p_1, p_2, p_3 and the separation into subsets T_s and T_G which are relevant for that particular test. This might be appropriate, for example, if the decision as to which software engineer will be responsible for fixing a bug is only taken when a bug is found. However, a wait-and-see specification will not allow an analysis of expected completion time for the test suite.

2.6 Informative failure

In practice, when the software is returned after an error is found, there will usually be some auxiliary information which will be helpful in determining which of the three possibilities R_1, R_2, R_3 has occurred. Such information is usually difficult to model in advance. Therefore, we shall concentrate in what follows on evaluating upper and lower bounds on such information. The upper bound corresponds to the situation where we obtain no additional information on failure, which we call *uninformative failure.* The lower bound comes from supposing that, having found an error and sent the software for retesting, then we will obtain sufficient additional information to allow us to determine precisely which of the three possibilities R_1, R_2, R_3 will apply in the retesting situation. We term this *informative failure.* In practice, we will usually obtain some, but not perfect, additional information about the retest status of the model on failure, so that the analysis of informative and uninformative versions of the retest model will provide bounds on the expected time for the retest cycle, and identify the importance of such additional information for the given testing problem.

2.7 Modifying the graphical model

We now describe how to incorporate the above probabilistic specification into a modified graphical model. We begin with the original graphical model. We add each test T_i that we carried out previously as a node which is linked to the corresponding domain nodes which determine the probability that the test is successful. We now introduce a further *retest* root node, R. The node R is linked to each test node T_i except node T_r. We now need to describe the probability distribution of each T_i given all parents, i.e. the original parents plus node R. Let T_i^* be the event that test T_i succeeds. Suppose that the original parents of T_i have possible combined values $A_1, ..., A_m$. In the original model, we therefore determined $P(T_i^*|A_j), j = 1, ..., m$, and used these probabilities to propagate the observation of the test successes across the model. In the new model, we must describe the probability for the event that T_i succeeds given each combination of A_j and R_k. We do this by assigning the probability for T_i to succeed as given by the original parents if either R_1 occurs or the test in question is not similar to T_r, but assigning the probability for T_i

success to be the prior probability $P(T_i^*)$ of success in all other cases, as in such cases the observed probability is not affected by the current states of the parent nodes. Formally, we have

$$\begin{aligned}
P(T_i^*|A_j, R_1) &= P(T_i^*|A_j) && \\
P(T_i^*|A_j, R_2) &= P(T_i^*|A_j), && T_i \in T_G \\
P(T_i^*|A_j, R_2) &= P(T_i^*), && T_i \in T_S \\
P(T_i^*|A_j, R_3) &= P(T_i^*) &&
\end{aligned} \tag{1}$$

Given the new model, we can now feed in the observations of success for each of the original tests $T_{[r-1]}$ and the new test T_r and this will give our updated probabilities for the new model. Each additional test that we carry out is added to the model in the usual way as a fresh node; for example a new test which repeats an earlier test is added as an additional node to the model whose success updates beliefs across the model in the usual way. Observe that while no further test nodes are attached to retest node R, each retest will update beliefs over R and thus update beliefs about the information contained in the original tests.

2.8 Expected duration of the retest cycle

In principle, we should choose a test suite in order to minimize the time to completion of the full retest sequence. Further, after each fault is found, we should choose a new test suite optimised against the revised model. As there will usually be a substantial period while the software is being fixed, then it will often be practical to rerun the test selection algorithm in that period to select a new test suite. However, to assess the expected time for completion of the cycle in this way would be very computer intensive for large test suites. Therefore, we find instead a simple upper bound on the test length which corresponds well to much of current practice, by considering the expected length of time to complete successfully any given fixed test suite. Our test procedure is as follows. We carry out each test in sequence. Either test j passes, in which case we go on to the test $j + 1$, or test j fails, in which case the software is fixed, and retested, where we repeat the tests on that subset of the first j tests which are required under the model (1). Either all these tests are successful, in which case we can go on to test $j + 1$, or one of the intermediate tests fails, in which case it must be fixed and retested, and so forth.

To assess the effect of retesting, we must consider, at the minimum, the amount of time that it is likely to take to fix a particular fault, and the amount of retesting that is required when a fault is found and supposedly fixed. Therefore, let us suppose that with every test T_j, we associate an amount of time c_j to carry out the test and a random quantity $X(T_j)$, which is the length of time required to fix a fault uncovered by that test. We could similarly attach costs for carrying out the tests and fixing any faults. We elicit the prior expectation and variance for each $X(T_j)$. We may, if appropriate, consider the time for fixing more carefully, by treating $X(T_j)$ to be a vector of times taken for error fixing, depending on the nature and severity of the error, or we may instead simplify the model by choosing a single random quantity, X representing the time to fixing given any fault. Different levels of detail in the description will affect the effort in eliciting the inputs into the model, but will not affect the overall analysis.

Suppose that we have chosen and sequenced a test suite $S = (T_1, ..., T_r)$. Suppose that tests $T_1, ..., T_{j-1}$ are successful, and then test T_j fails. Suppose then that the software is fixed. Suppose now that test T_j is repeated and is successful, so that the current model is as described by (1). Now, some of the tests $T_1, ..., T_{j-1}$ will need to be repeated, while some may not be necessary. We can check whether any of the first $j - 1$ tests may be eliminated by running a step-wise deletion algorithm on the subset of tests $T_1, ...T_{j-1}$ by finding at each step the minimum value of the posterior probability for the criterion function on which the original design was constructed when we attempt to delete one of the first $j-1$ tests, where all other tests in S are successful. We denote by $R(j)$ the *retest set* given failure of T_j, namely the sub-collection of $T_1, ...T_{j-1}$ which must be retested when we have deleted all of the tests that we can by stepwise deletion under the retest model. For the uninformative failure model the retest sets required for (3) are extracted by direct calculation on the model (1). For the informative failure model, if T_j fails, then with probabilities p_{1j}, p_{2j}, p_{3j}, there will occur possibilities (R_1, R_2, R_3) of subsection 2.1, and after the software has been fixed we will obtain sufficient additional information to allow us to determine precisely which of the three possibilities (R_1, R_2, R_3) will apply in the retesting situation. Under (R_1), no retests are required, under (R_2), the set T_{Sj} of similar tests to T_j are repeated, and under (R_3) all tests are repeated.

We denote by $\mathrm{E}_+(T_j)$ the expected length of time between starting to test T_j for the first time and starting to test T_{j+1} for the first time, so that the expected time to complete testing for the suite $S = (T_1, ..., T_r)$ is

$$\mathrm{E}_+(S) = \sum_{j=1}^{r} \mathrm{E}_+(T_j) \tag{2}$$

To evaluate each $\mathrm{E}_+(T_j)$ fully is an extremely computationally intensive task. An approximate calculation which will give roughly the right order of magnitude for test time is as follows. When we first run test T_j, there are two possibilities. With probability $(1-q_j)$, T_j is immediately successful, and, with probability q_j, T_j fails, in which case the software must be fixed and the subset $R(j)$ of tests must be repeated, along with T_j, where q_j is the probability that test j fails given that tests $1, ..., j-1$ were successful. The full analysis of the length of retest time under the retest model is usually too complicated to be used directly as a design criterion. A simple approximation, on the assumption that the retest will definitely fix the error that was found, that will usually give roughly the same result is to define recursively

$$\mathrm{E}_+(T_j) \approx c_j + q_j(c_j + \mathrm{E}(X(T_j)) + (\sum_{i:T_i \in R(j)} \mathrm{E}_+(T_i)). \tag{3}$$

If each retest set is obtained from the uninformative failure model, then (3) gives an upper bound for expected length of the test cycle. Under the informative failure model, we have approximately that

$$\mathrm{E}_+(T_j) \approx c_j + q_j(c_j + \mathrm{E}(X(T_j)) + p_{2j}(\sum_{i:T_i \in T_{Sj}} \mathrm{E}_+(T_i)) + p_{3j} \sum_{i:i<j} \mathrm{E}_+(T_i)). \tag{4}$$

which we use to establish our lower bound for the expected time to complete the test sequence.

Now suppose that we have already selected and sequenced a test suite $S = (T_1, ..., T_r)$, using the BGM approach described by Wooff et al [7]. We would prefer to run as early as possible in the sequence those tests which, if they fail, will require a large number of retests. A practical approximation to an optimal sequencing for the full retest cycle for the selected test suite is to choose a stepwise search algorithm that proceeds by switching adjacent pairs of tests in S in ways that we expect to shorten the overall testing time. In particular, if $T_{i-1} \in R(i)$, then, to a good approximation, the only effect of switching the order of T_{i-1} and T_i is to reduce testing time if, having switched the order of the two tests, then we would not need to include T_i in the retest set for T_{i-1}. We may choose to do this under either the informative or the uninformative form of the model. Therefore, we continue to make pairwise transpositions, until we can no longer remove an item from any retest suite by such transpositions.

Finally, when we have constructed our test suite, we may choose to carry out a probabilistic simulation to assess the full distribution of the time to completion of the test procedure. We may choose to incorporate a more realistic description of the test schedule into the simulation, for example allowing the category of errors which we would choose to fix to depend on the length of time remaining before the proposed release date of the software. Simulation under informative failure is straightforward. Simulation under uninformative failure is more complicated as the retest model increases in complexity with multiple failures, but in principle the simulation is again straightforward.

2.9 Remarks

There are many further complexities to take into account, for example: the role of provocative testing and the way in which batching of tests is handled. Additionally, the elicitation of the specifications $X(T_j)$ needs further consideration. For example, the sensitivity to simplifying choices (such as choosing $X(T_j)$ to be the same for all j) needs careful examination. One possibility here, for example, might be to choose lower and upper choices X_L and X_U to correspond to the lower and upper bound arguments.

3 Multiple failure modes

Treating all faults as equally important is appropriate for many applications. However, for some applications, there may be many kinds of possible faults at the domain node level, ranging from errors which are sufficiently severe that they would certainly need to be fixed before the software could be released to errors that certainly would not be worth the cost of fixing, where the only purpose of finding the errors is to be able to alert users in the documentation. Many errors are of intermediate severity, such that they would be fixed if found sufficiently early in the test cycle, but would not be fixed if uncovered too near the projected release date. Thus, we now consider how to allow for a range of consequences of error, how to allow for different costs for different types of tests, and the implications for test design.

3.1 Nodes for multiple failures

To handle the range of different errors, we need to describe in more detail the consequences of failure. In [7] we partly addressed these features, in the sense that we have allowed different domain nodes to have different consequences of failure. While there are many ways to classify the different types of fault, one approach is to rate each fault on a four point scale, as follows:

4 (show-stopper) corresponds to a fault which must be fixed before release;

3 (major) to a fault which we would very much want to fix if allowed by time constraints;

2 (minor) to a less serious fault which we might fix given time;

1 (trivial) to an essentially cosmetic fault which we would usually not bother to fix.

Generally, the level of granularity in the scale adopted should be chosen to reflect the amount of detail that is available and considered relevant by the expert testers.

There are various ways to incorporate different levels of faults into the graphical model. At present, each domain node within a BGM is linked to a single failure node which carries, inter alia, the probability that the entire BGM contains at least one bug. One way to handle multiple failures is to replace the single failure node by one node for each failure level. Thus, if the testers have specified four failure levels (show-stopper, major, minor, cosmetic) then we replace the BGM's single failure node by four failure nodes, F_1, F_2, F_3, F_4 and add a directed arc from each domain node to each failure level node for which it is possible that a test in the domain might uncover an error at the corresponding level. In summary, this is a mechanism for decomposing each BGM according to the different failure modes contained therein.

3.2 Specifications and test implications

If each test of a domain node can only reveal errors at one particular level, then we do not need to carry out any further modelling. If it is possible that a test in a domain node might uncover errors of different degrees of severity, then we must model not just the probability of error in a given node, but the probability of each different level of error in that node. We model the different fault levels in the explanatory nodes, and propagate the different errors to the domain and test nodes. While it may require more effort to construct the corresponding conditional probability tables, introducing the different types of error that we can observe may actually simplify the resulting elicitation process as the knowledge of the expert tester may be based on previous experiences of finding the different kinds of error.

In order to modify the BGM to take account of multiple failure modes, we need to construct each failure node F_i and specify $\mathrm{P}(F_i|N_j)$ for every domain node N_j in the BGM. $\mathrm{P}(F_i|N_j)$ is the probability, given that node N_j fails a test, that the failure is of degree i. Note that $\sum_i \mathrm{P}(F_i|N_j) = 1$ for each N_j. Given these specifications, we can calculate the prior probability of each kind of failure. We write S_0 to be the null test set, and write $\mathrm{P}(F_i|S_0)$ to be the prior probability of failure of each kind before testing.

Each successful test modifies the probabilities at the four failure nodes. We denote by $\mathrm{P}(F|S+)$ the vector of probabilities $(\mathrm{P}(F_1|S+), \mathrm{P}(F_2|S+), \mathrm{P}(F_3|S+), \mathrm{P}(F_4|S+))$, where $\mathrm{P}(F_i|S+)$ is the posterior probability that there is at least one failure in the failure node at level i given that all members of the test suite S are successful.

3.3 Considerations for design

There are two complementary approaches to testing that we may take. Firstly, we may take a decision-theoretic view, in which each error that is not found incurs a cost, on a utility scale, to the company. Therefore, we extract from the graphical model at any stage the expected utility for releasing the software given the current state of knowledge about potential errors.

For many problems, however, there will not be an agreed utility structure, partly because there will be many stake-holders each of whom have a different view of the costs and partly because the test team may not have access to the detailed judgements required in order to place precise utility values. In such cases, we may instead identify the different risks involved in the software release, and extract from the graphical model the probability associated with each particular risk. In this view, the model provides the precise quantitative information which is required in order to implement the risk management procedures related to the software release.

3.4 Test design criteria for multiple failure modes

Our intention, in high reliability testing, is to reduce the posterior probability of even one serious fault to an acceptably small level. Suppose that for each level of fault, i, we specify the utility cost for releasing the software with at least one fault of this type to be U_i. A simple way to design for the extended fault classification is to construct the *release cost*:

$$C(S+) = \sum_i U_i \mathrm{P}(F_i|S+),$$

which is the expected utility cost of release given that all tests in suite S are successful. If we accept this test criterion, then it is straightforward to design a corresponding test suite, using identical methods to the approach given in [7], but replacing the criterion of minimizing $\mathrm{P}(F|S+)$ with the criterion of minimizing $C(S+)$.

If we do not want to place costs on the various fault nodes, then we may instead place targets on the individual elements of $\mathrm{P}(F|S+)$. We may choose designs to meet these targets using similar step-wise search algorithms. Although we are designing on several criteria, we have classified the errors in order of importance, so that a simple approach is firstly to design the test suite to control the posterior probability for level 4 errors, then to add tests to control the posterior probability for level 3 errors and so forth. Regular step-wise deletion stages are used to weed out superfluous tests. Alternately, we may seek to optimise some similar criterion to the decision-theoretic criterion described above, and monitor the individual failure probabilities to ensure that the test design controls the individual error probabilities to the required levels.

3.5 Implications of resource limitations

If, due to constraints of money and time, it becomes apparent that we cannot sufficiently reduce the overall error probabilities, then we could further subdivide the collection of possible faults, by introducing a fuller collection of fault nodes so that we can track errors in different core functions of the software. This will give more information on which to assess the suitability of any test suite to identify the various areas in which substantial levels of risk remain, so that a reasonable compromise suite of tests can be made. Incorporating the corresponding utility costs, we design for the modified version of $C(S+)$ as above.

The implications of resource limitations may become apparent prior to testing, through using the analyses we have described here and elsewhere. It is also frequently the case that resource constraints become apparent during the testing process. For example, the testing process may reveal bugs which take unexpectedly long to fix and so delay further testing. When this is the case, it may similarly be advisable to further subdivide the collection of possible faults and so enable any further testing to concentrate on whatever are, by that stage, the most crucial areas in the software still to be tested.

3.6 Remarks

In the above description, we have treated each test as though it has the same cost. If test costs differ, then the algorithm that we follow is as above, but at each step-wise addition or deletion, we measure the benefit that can be gained for a fixed level of cost.

All of the test procedures that we have described above are reliant on the underlying graphical model. In certain circumstances, we may reserve a portion of our budget for diagnostic testing of the judgements in the model, which essentially corresponds to placing rather more tests on very important areas of the software functionality than appear to be necessary given the form of the model. This is addressed later.

We described in [7] how we carry out sensitivity analyses to check the sensitivity of the conclusions we reach to variations in the prior specification. It is straightforward here to carry out similar sensitivity analyses to test the effect of varying the prior specification on the four probabilities for each kind of failure. This is useful particularly when testers have a good feel for the number of the most serious kinds of bugs thought to be in the software, prior to release. Further, historic information from earlier testing of similar software, or of software sourced from the same supplier, can be used to help calibrate the models for the software being tested.

We have addressed test design for multiple failure modes in the sense of targeting design at the most probable locations of serious failures. This leaves open the question of how to handle the test-retest cycle where a test fails and where we allow the possibility of determining the level of failure at that time.

A related issue is that, even if the test process is uncovering no faults, the importance of certain kinds of error can be upgraded or downgraded as time proceeds. For example, it might be decided late on in the testing process that certain functionality will not, after all, be released. In this case, what might have been very serious possible errors are substantially downgraded. One possibility is simply to re-optimise what remains of the testing process at this stage. An alternative is to have a mechanism which takes account of the downgrading or upgrading of certain kinds of error without going to full re-optimisation of the test process.

Finally, we have not considered how we may take advantage of observing the kind of error that occurs, and propagating this information over the network. For example, if we see a show-stopper, one possibility would be to propagate this information over the models and thereby raise the probability of there being a show-stopper in untested nodes. This would require some extra modelling, in particular because the number of states in each domain node, at present essentially the two states [good, bad], would have to be decomposed further: for example, into the states [good, show-stopper, major fault, minor fault, cosmetic fault]. Beyond such more complicated modelling, there are few extra difficulties associated with this extension.

4 Diagnostics for the BGM approach

The main purpose of diagnostic assessments is to compare actual performance (either through testing or by use of software in the field) to the performance predicted by the models. In the BGM approach, the BGMs are representations of the tester's knowledge, so that the value of good diagnostic procedures is (a) to provide feedback on the quality of the representation for an individual tester for a specific software testing problem, and more generally (b) to help calibrate the general BGM approach through continued feedback on the success of the modelling and test design processes.

With regard to the quality of representation for an individual tester (or team of testers) for a specific problem, our concerns fall into two areas: whether or not the structuring process gives rise to adequately structured models; and whether or not the probabilistic specification appears to be in line with actual testing and field performance. These two issues are affected by the diagnostics that we attach to (1) test performance; and (2) field performance, which we describe separately.

4.1 Diagnostics during testing

4.1.1 Test performance

Typically, there are three scenarios we might consider. For these, we suppose that the testing process does not include purely diagnostic tests (which we describe below), and that a test suite has been designed to achieve desired levels of reliability. The testing process should be represented by one of the following scenarios.

- Performance of the software is about as expected. That is, testing reveals about the number of faults which the models predicted, and in expected places. The implication is that the structuring process delivered about the right structures, and the specification process delivered about the right specification.

- Performance of the software is better than expected. That is, testing reveals fewer faults than the models predicted. The implication is that the software was actually more reliable than specified, and thus possibly that resource was expended unnecessarily on too much testing or structuring to too fine a level of detail.
- Performance of the software is worse than expected. That is, testing reveals more faults than the models predicted, and may also have found faults in unexpected places. The implication is that the specification process over-stated the reliability of the software, or that the structuring process provided models which were inadequate.

4.1.2 Test diagnostics

Two kinds of diagnostic are available from observing tests from a test suite. First, there is a global measure available at the end of testing which compares the number of faults found with the number of faults expected at the start of testing. Such information is mostly useful for calibrating the general BGM approach (for example, testers will obtain an improved 'feel' for the specification process) and for updating or evolving the BGM for further use.

Secondly, diagnostics can be calculated as each successive test in a test suite passes or fails. Each test has a certain additional chance of revealing a fault. Assuming the tests are scheduled to test the software areas with highest probability of failure first, we compare the results of successive tests with the marginal reduction in probability of failure (or expected number of faults remaining) that each such test provides. For each test (and in particular each sequence of tests) we expect a roughly matching number of faults found and reduction in expected faults. Too high a proportion of faults found provides an early indication that the prior probability of failure is too low, and vice-versa. Later in the testing cycle, it may be that a test with very low probability of revealing an error fails. This may be an indication of deficiency in the structure of the model.

4.1.3 Sensitivity analyses and diagnostic tests

Information from tests is propagated over the various BGMs. Depending on the testers' judgements, the results of tests will have direct implications for some nodes (that is, the nodes directly observed), and indirect implications for some other nodes (that is, the nodes linked to those directly observed). For many nodes, depending on the strength of link between observed nodes and the efficiency of the test design, the model will suggest (according to the testers' judgements) that the nodes will have been fully tested. It will be useful to measure how important it is to have observed such nodes directly. This measure will provide us with a diagnostic as to which further diagnostic tests we might carry out if we do not entirely trust the links between nodes. One possibility is to construct a measure within each node which partitions the evidence in a node into direct and indirect evidence. This would probably be straightforward in a Bayes linear paradigm [5], but would require new research in this setting.

In addition to diagnostic tests which arise from identifying areas of the model which have been only indirectly tested, there is a straightforward way to identify further diagnostic tests. At some point, the main testing process ends, either because the resource available for testing has been used up, or because testing indicates that the software has achieved desired levels of reliability. In the former case, there is presumably no resource available for further diagnostic tests, so we need only consider the situation where the software is ready for release according to specified levels of reliability, and where there may be some resource available for further diagnostic tests. We now run as a first diagnostic test the test which we expect to resolve most uncertainty in the model (in the same way that we identify useful tests for the test design process). This might require no extra effort because the test design process will typically result in an ordered sequence of tests that we wish to run. Testing will have stopped part way through this sequence, and so the first diagnostic test is simply the first test not to have been run in the original ordered sequence. (Alternatively, we might simply design a new suite of diagnostic tests, or design a single diagnostic test at every iteration.) We now check whether there is sufficient resource to run an additional diagnostic test, and proceed in this way until the resource has been spent, or the model has been fully tested.

The advantage of this procedure is that the diagnostic tests continue (in some optimal sense) to test the software to higher and higher levels of reliability, whilst also helping to confirm the validity of the model's structure through testing of areas which should have no, or a very low, probability of failure.

Whether or not diagnostic tests can be run depends on whether or not there is resource available, after normal testing has been completed. Such decisions need to be addressed taking into account the viability issues discussed later. For example, if there is resource available within the budget for testing a particular piece of software after the normal testing stage has been completed, it may be that this resource is better diverted to other needier projects.

4.2 Diagnostics after software release

4.2.1 Field performance

We suppose that the software is released for field use following testing to an adequate level of reliability, and in particular that the models suggest that there are no (or there is only a small probability of) serious faults remaining in the software. The possible scenarios arising from field use are as follows.

- Performance of the software is about as expected. That is, no serious extra faults were revealed. The implication is that the structuring process delivered about the right structures, and the specification process delivered about the right specification, that the chosen test design was effective, and that the test-bed provided an adequate representation of field conditions.
- Performance of the software is better than expected. That is, no serious extra faults were revealed and there were fewer other faults than expected (assuming the software to have been released with some probability of remaining faults). The implication is that the structuring process delivered about the right structures, and the specification process may be understating the reliability of the software, that the chosen test design was effective, and that the test-bed provided an adequate representation of field conditions.
- Performance of the software is worse than expected. That is, serious faults were revealed in field use, or more than expected less serious faults occurred. The implication is that the specification process over-stated the reliability of the software, or that the structuring process delivered structures which were not sufficiently finely detailed, or that the chosen test design was ineffective, or that the test-bed did not provide an adequate representation of field conditions.

4.2.2 Diagnostic feedback from field use

Diagnostic feedback from field use can be used in a number of different ways. Field faults can be classified as follows.

- Faults which could have been found by the testing process, implying that the testing process was not sufficiently effective.

- Faults which could not have been found by the testing process, implying that the test conditions were inadequate to represent field conditions.

We concentrate on the former kinds of fault. We will assume that the nature of the fault can be precisely identified, and that the mechanism by which that fault could have been found by testing can be precisely identified. For example, we assume that we can identify a specific test or test type which could have found the fault. Such faults can be further classified as follows.

- A fault is category A if the fault occurs in a location which the model suggests has been fully tested.
- A fault is category B if the software was released with a relatively low probability of such a fault occurring, and in particular that the test that could have tested for this fault, but was not carried out.
- A fault is category C if the software was released with a relatively high probability of at least one fault remaining, and the test which could have identified the fault is just one of a number of tests which might have been carried out, but were not.

Category A faults are the most serious as they imply that the tester has omitted to consider parts of the software to be tested, or that the BGMs are not to a sufficiently fine level of detail. An example would be that the tester judges that one test of software intended to process 8 to 16 digit numbers is appropriate fully to test the software, and that a single test (say, using an 8 digit number) passes. Later, we discover that there is a fault in processing 9 digit numbers. The conclusion is (a) that the tester was wrong to consider that one test tests all, and (b) that the tester might consider partitioning the input domain further, say into 8, 9-14, 15, and 16 digit numbers. Category A faults thus have an implication for the underlying BGMs.

Category B faults have particular implications for the probabilistic specification, as an error has occurred that the model suggests as having been unlikely. This provides information to update the current model, perhaps by down-scaling the probabilities that the software is fault-free, or perhaps by directing the tester to reconsider the judgements which were used to quantify the model.

Category C faults are, in some sense, expected. Thus, they do not have special implications for the model or the approach.

In general, in addition to the implications already considered, the number of field failures might also be used for general calibration of the BGM approach. This includes the following: choices of tuning parameters for those areas of the BGMs concerned with sporadic error propagation, as considered in section 6; choices of probabilities for the probability of introducing side-effect errors during the test-retest cycle, as considered in section 2.

4.3 Fault severity

We have addressed the implications of faults from test-bed and field use for diagnostics. It is also useful to take into account the severity of such faults. At present we have simply described these as serious or non-serious, but it is clearly desirable to formalize them, and to link the severity of fault with our recommendations on multiple failure modes, as described in section 3.

5 Model maintenance and evolution

The BGM encodes the knowledge of the tester: (a) for further use of the tester, and (b) for the use of other testers. To enable changes to the BGM in the light of new information or changed functionality, it is important to distinguish between the actual uncertainty judgements made by the tester and the reasons for making these judgements. Particularly in circumstances where testers involved in the creation of the BGM are not involved in any future revision (for example restructuring) of the BGM, it is important that the information on which previous judgements were based is still available.

When a BGM is developed to assist software testers, the structuring of the software actions according to the tester's view of its functionality is not essentially a Bayesian activity. Instead, such structuring of software functionality extracts from testers a core framework for the testing process which is of considerable practical use whatever the testing strategy adopted. One advantage is that the corpus of structures elicited provides an explicit representation of the tester's reasoning concerning the functionality of the software to be tested. A second advantage, of course, is that we can construct from them BGMs to provide the full panoply of the BGM approach. These are the twin themes of what we must take into account with regard to model maintenance:

- how the structures, which carry the tester's judgements as to how the software functions, can be modified or evolved to take account of new or modified functionality;
- how the probabilistic information specified over the corresponding BGMs can be modified or evolved.

For the remainder of this section, we suppose that we have constructed a Bayesian model for the previous release of some software, and wish to update to a new release of the software. We describe the sources of information about software reliability which may help to construct the new model, and the actions which should be taken to use such information.

5.1 Documenting the construction

If BGMs for a software system are to be maintained in the sense outlined above, it is essential first to add an extra layer to the structuring process. This consists of (a) careful documentation of the grounds for structuring the software functionality in the first place, and of (b) careful documentation of the reasoning underlying the probability specification process. The former is naturally recommended whether or not the testing strategy employs the BGM approach.

Such careful documentation might be of use at several stages. First, when creating the initial BGM, it is useful to have a clear idea of the information available, to ensure that all relevant information is taken into account correctly. Secondly, if at a later stage the need occurs to re-examine the BGM, then well-recorded information will be of great value. It should be emphasised that such information may originate from several sources, ranging from actual reliability data to tester's judgements based mostly on experience. The benefits of recording such information include the following.

- The availability of rigorous documentation should simplify and improve the process of making prior probability judgements.
- Experiences arising throughout the testing process can be referred back to the underlying groundwork, thus providing for sanity checks and assessment of the testing methodology (whether or not via the BGM approach).
- Proper documentation as to why certain judgements were made is available for auditing purposes. When software turns out to have been badly tested, this presumably leads to large costs of some kind. Therefore it is useful to have audit procedures for identifying the reasons why various decisions were made. It is likewise possible to identify when software has been

very competently tested, and it is useful then to be able to use any documentation to identify good practice.

- When the models need to evolve, there is a careful record as to how the models were constructed initially, so that no extra uncertainty as to original software functionality need be introduced at the later stage of model evolution, and any further evolution can (if desired) take place along the lines chosen for the initial structuring.

5.2 Evolving the structures

5.2.1 New functionality

The evolution of structures is necessary when the functionality of the software is extended or added. Note that the careful documentation of earlier structuring for testing is vital in deciding what is new and what is extended, and by how much it is extended. We define new functionality to be extensions to the software for actual new functionality, so that testing such functionality is not informative to remaining functionality. We handle such new functionality just as we would handle any other software to be tested for the first time, as described in [7]; for example we can judge which functions are intrinsically difficult to program and so identify areas with high failure probabilities. Note, in addition, that we also have the advantage of being able to take into account the general level of reliability of the remaining pre-existing software being tested, which will help the specification process.

5.2.2 Extended functionality

Our approach differs according to whether the extended functionality is minor or major. By minor extended functionality we mean that (according to the tester's judgement or to explicit knowledge of any required software rewriting) the software is modified in a minor way to accommodate minor changes in functionality. A possible example is that software dealing with 8-digit numbers might be revised so that numbers ending with a zero are treated slightly differently. Such extended functionality should be able to be handled at the stage where we partition a software action for its inputs. This will involve adding to the current partition structure, which is straightforward, and adding new nodes to existing BGMs. Within a tool, this should be simple to accomplish. Finally, we will need to make a probability specification over the new and old parts of the structure.

The probability specification over the old parts of the structure can be handled as described below where we address evolving the probability specification. In particular, it will normally be advisable to take into account that the software extension may have introduced new faults into the pre-existing (and previously tested) areas. The probability specification over the new parts of the structure is guided by the specification made for the old. Typically, the tester would be guided to probabilities of the same order of magnitude as for the old parts of the structure, but generally larger to reflect the novel nature of the new functionality. Sensitivity analyses can then be run to assess the implications for test design and so forth, and calibration analyses can be run to help ensure that the BGMs are in line with the tester's judgements.

For major extended functionality, we have the choice of dealing with the software as though it is entirely novel or of repartitioning and evolving existing BGMs, as for a minor extension of functionality. An example is provided by extending old software capable of dealing with 8-digit numbers to software dealing with up to 16-digit numbers. If we choose to regard this as extended functionality, new nodes are added to the existing BGMs, with corresponding scaling-up of root probabilities to express the possibility of new errors being introduced in old functionality. Alternatively, we can choose to regard the software changes as so wide-ranging as to merit ab initio modelling, but where we can take advantage (a) of the existing BGMs for the software being extended, to give order-of-magnitude guides as to the likely reliability outputs from the new model; (b) from the careful documentation process outlined above, which details how the existing BGMs were formed, and whether or not there are diagnostic features from live testing which need taking into account.

5.2.3 Knowledge at code level

Depending on how much information is available, we may have direct access to code information or be able to request such information from the code producers. The types of information that we might find informative are knowledge of the formal call structure or the observed call structure or information as to which aspects of the code had been substantially modified since the previous release. If this is the case, any information about the code that is newly available is now used to check whether the modelling has overlooked any important features of the new release. Information about the code for earlier releases of the software should already be reflected in the previous generation of graphical model.

5.3 Evolving the probability specification

In this section we assume that there is no substantial new functionality, and that we are dealing with a revision of existing software with the same basic functionality as before. (It is straightforward to allow for minor extensions of functionality.) The probability evolution needs to take into account: information from previous testing of previous versions of the software; information from field usage of previous versions of the software; and the scale of software revision.

5.3.1 Exploiting the test history

Test records for previous releases of the software contain information as to which tests passed and which failed, both initially and on retest. These records are directly relevant for criticizing current modelling strategies and for constructing the new models. Therefore, diagnostic comparisons between the previous model and the corresponding test results are used to identify where the tester was over-confident or under-confident for the software reliability in the previous release, and by how much. The diagram for the new release of the software might start by recreating the diagram for the preceding release, as a simple scaling, at the level of the root probabilities, of the original prior probabilities using the above heuristic to express confidence in the original model.

5.3.2 Exploiting previous field experience

Commonly, the previous release of the software will have been tested or used live. From such use, new faults may have been identified. Failure reports from the field identify: which tests were overlooked; which *provocative* tests should have been carried out; and, by implication, areas with no reported failures increase confidence in the testing process and the true software reliability for the previous release. Therefore, one approach is to make an overall assessment of software reliability for the previous software release, taking into account the evidence from its field usage. This can take the form of a confidence assessment in the previous test model and also a level of confidence in the performance of the software immediately prior to the latest rewrite. This can be used to rescale the root probabilities derived above (after considering the previous test history) and possibly suggests further nodes representing provocative tests.

5.3.3 Knowledge about the nature of the software rewrite

If the new release is a minor upgrade to the old release, then the most important consideration is whether the field errors have been fixed, while if there has been a major rewrite, what matters is largely the new errors that may have been introduced. The outputs from the two previous sections, taking into account both the testing history and field experience, will have produced revised models for the initial state of uncertainty and the posterior state of uncertainty for the original model. These are now combined in light of knowledge about the nature of the software rewrite. For example, we might choose a linear weighting of the prior and posterior root uncertainties at each node (near prior for major rewrite, near posterior for minor rewrite), possibly choosing different weightings for different regions of the model.

6 End-to-end testing

By end-to-end testing, we mean testing of a complete software system, including all its subsystems and links between them. The end-to-end testing problem is broadly similar to the testing problem we have already covered, and much of the methodology we have provided already can be applied straightforwardly to the end-to-end testing problem. It is particularly characterized by the requirement to test many subsystems, and the links between them, simultaneously, and typically requires more testing in the way of testing flow of software processing, and this has an implication for the test design process, as there are far more possibilities to consider.

6.1 Modelling for integration of subsystems

We view a software system as a collection of subsystems linked together. The full testing problem can be separated into two stages. First, we apply the BGM approach, as far as practicable, to each of the subsystems, which we treat as independent modules. We consider dependent subsystems below. Therefore, whatever the present level of testing, we consider that all the subsystems have been structured for testing according to the BGM approach. We deal with the case where this is not so, together with viability issues, later. It should be noted that if we have applied the BGM approach to all the subsystems in the system, then each such subsystem provides a full probabilistic summary of the current state of testing within that subsystem, whether or not any testing has taken place within that subsystem. In principle, there is no difficulty in designing either end-to-end tests or within-subsystem tests which will have an impact on the current probabilistic description of reliability within a subsystem. However, it may be more natural (and easier to manage) to undertake testing within subsystems as modules, especially if teams are given special responsibility for particular subsystems. Note that end-to-end tests will continue to provide fresh information about the state of reliability within each subsystem.

Consequently, we focus here on testing the integration of subsystems. The structuring required for the integration problem is similar to the structuring involved for the subsystem testing problem. The team responsible for end-to-end testing identifies the software actions which link the subsystems (and also identifies any reliability dependencies between subsystems: see below), and provide judgements as to their reliability. This team must also specify the inputs which form the domain of each software action and partition them in the usual way. Further, the team must complete the mapping of tests to observables, and must specify, for example, any constraints on the ranges of inputs provided for tests. The testing of the integration of subsystems needs to address two issues.

- The links between subsystems need testing. This is relatively straightforward to achieve, as all the links should be representable as simple software actions dealing with transmission and reception of data. The case study addressed in [7] contained several examples of such links.

- The integration of subsystems. The kinds of errors we want to detect here are the kinds of error which would not ordinarily occur if the testers are correct in their judgements about how the software is structured. Such errors may occur intermittently or be difficult to replicate. Other such errors may occur because there were unforeseen (and therefore unmodelled) relationships between subsystems.

Of these two issues, the second is much harder to address. We propose to deal with it by allowing a general model for sporadic errors of this kind. We begin by specifying a high probability for sporadic error across the system. Each test pass reduces this probability. However, the observable for each such test is that each database must be examined for side effects before and after each test, and examined to ensure that the only differences are those required by the instructions implied by the test. Such checks may, or may not, be an intolerable burden, but do genuinely test the integration. Handling of such sporadic errors is considered briefly in section 2, in which we also consider side-effect errors introduced by the test-fix-retest process.

If the burden of checking databases for side effects is too costly, simpler approaches may be employed. However, these would provide less information. One such approach is to consider a hierarchy of sporadic errors for databases within a BGM framework. For example, we might represent each database used in the system by a node in a BGM, each connected to a common parent node which carries the general level of sporadic error across the system. This general level of sporadic error will need to be specified at the outset to reflect the testers judgements, but would be a tunable parameter which could be subjected to sensitivity and calibration analyses. Similarly, each of the nodes for the single databases carries the general level of sporadic error for that database. The nodes on the BGM are connected by arcs to represent possible flow of information, to whatever level of sophistication is deemed appropriate to meet the testers judgements. For example, a pair of databases might be judged as having a higher pairwise degree of unforeseen side-effect, and this can be straightforwardly modelled via a BGM.

6.2 Input arrival testing

Suppose that testing has been carried out to some satisfactory level. Typically, the tests we design for end-to-end testing will pass through a series of subsystems. For each of these subsystems, the inputs will arrive as intended and of the correct type. That is, if a subsystem expects an eight-digit number to process, part of the testing process focuses on delivering that eight-digit number correctly. Furthermore, each subsystem is typically tested for the range of inputs it is intended to process.

End-to-end testing should also test what happens if one subsystem passes on inputs which are correct for that subsystem, but inappropriate for a subsequent subsystem. This might happen for two different reasons. First, it may be that the systems design has overlooked an incompatibility between one subsystem and a subsequent subsystem. This will be handled by at the stage at which tests are mapped to the observables, as the input constraints for the various subsystems are considered at that point. Secondly, it may be that the inputs arriving for a subsystem are inappropriate because of an error occurring in the previous subsystem, or because of a transmission error. However, both of these are modelled via software actions in the usual way, and so require no separate treatment.

6.3 Clusters of subsystems

It may be necessary to handle situations where the software in one subsystem is related, in terms of reliability, to software in another subsystem. This might occur, for example, because the software engineers copy code from one application and re-use it for another. An example is where encryption and decryption algorithms exist as identical duplicates in different subsystems. This then implies that there may be situations where nodes within BGMs for one subsystem may be connected to BGMs within another subsystem. In principle this can be handled in the way we describe in earlier documents, which is to connect such BGMs via a clustering factor. However, this has the drawback of being more difficult to manage, particularly if the subsystems are being tested by different teams.

An alternative is to continue to test independently within subsystems. However, when we come to use any reliability summaries from such modularized testing, the end-to-end model may understate the degree of reliability in each cluster of subsystems. In this respect, modularized testing offers a conservative picture. If the test budget allocated to such testing is sufficiently high, we may choose to undertake more detailed modelling in order to detail dependencies more carefully. Such a choice can be guided by viability considerations as discussed below.

We should, of course, take advantage of any such software relationships. Therefore, one of our recommendations is that, if practicable, the tester responsible for the full system should coordinate the BGM approach to structuring the software actions in the two subsystems to minimize the effort of duplication of structuring and documentation; and that test failures directly attributable to those software actions in one subsystem be notified to the testers responsible for testing the related subsystem.

6.4 Viability

As we describe in section 7, it may not be worthwhile to construct BGMs for every subsystem in the system being tested. The guidelines contained therein can be used to decide whether or not fully to model each given subsystem. For example, if one of the subsystems is known with high confidence to be highly reliable, and if it would also be very expensive to construct the BGMs for it and then test them, we might decide not to construct them. We would, of course, continue to test links between this and the other subsystems.

In order to obtain numerical summaries over the full system, it will be necessary to specify directly appropriate summaries for subsystems which we do not test by full BGM modelling. If we can observe whether or not the outputs from such subsystems are correct, we might further introduce some crude mechanism for updating such numerical summaries. For example, we might introduce a beta prior distribution to represent the probability that the full subsystem works correctly, and update for each successful test observed. Otherwise, it might be best to treat such a probability of failure for such subsystems as irreducible without further detailed attention. Clearly, as elsewhere, the cost of paying more attention to modelling is offset by pay-off in knowledge gained.

6.5 Implications for design

End-to-end testing can subsume within it the testing within individual subsystems (for example every end-to-end test will help to update the BGMs for separate subsystems), but is probably best tailored according to these principles.

- Testing proceeds first within subsystems, to agreed levels of reliability.
- Testing proceeds until the level of sporadic error is deemed sufficiently low. This can be guided by whatever benefit and resource constraints are involved, as described in other documents for the general BGM approach.
- Testing proceeds until the links between subsystems have been adequately tested. There are BGMs representing the software actions involved in the links and interfaces between subsystems, so the methodology to do this has already been described in other documents.

- Testing proceeds until the various combinations of flow of control between subsystems has been adequately tested. This extra remark is made because there are typically many more such combinations than for processing within subsystems, and although each such control has typically a very high level of prior reliability, there may be a considerable cost for failure.

It is worth noting that it may be more efficient to regard every problem as an end-to-end testing problem, with no modular testing of subsystems, so that the problem of test design is how to test end-to-end tests that test the whole system and all the subsystems satisfactorily. Alternatively, subsystems might be tested modularly to some specified level of reliability, but not to a releasable state of reliability, and then further end-to-end tests can be carried out with the partial aim of completing the subsystem testing *en passant*. Whilst this would be efficient (and possibly practicable for small systems), it might be much more difficult to tackle the testing problem in this way.

7 Viability of the BGM approach for individual applications

We now discuss how the viability assessment process can be achieved for individual testing situations within a company or institution, this was also presented by Coolen et al [2]. The first part of the process is to link the BGM approach with the company's strategy for making decisions and assessing risks. This depends strongly on context and so it is more meaningful to explain how to match the test effort required in using the BGM approach to the requirements of the problem, so that, if all goes well, managers will always have a procedure that suits its needs. To do so, we begin with a qualitative assessment by the company of the costs and benefits of their existing approaches, and then contrast these with the costs and benefits of the BGM approach.

7.1 Assessment of an existing testing approach

We focus on an imaginary forthcoming project at a company, and attempt to see what the assessment must cover. There are two aspects to consider:

General: features which will be relevant to the assessment whichever testing approach is applied.

Specific: features which depend on the approach that is used, one such approach being the BGM approach.

Each of these two aspects can be cross classified according to

Cost: the cost of overlooking errors for the project (general) and of carrying out testing (specific)

Error level: the level of error in the software to be tested before testing (a *general* feature) and after testing (a feature *specific* to the approach used).

7.2 General assessments

7.2.1 Project cost

There are many relevant considerations for the cost of overlooking errors in the software before release. This is intimately linked to the general importance of the project for the company, and underpins all the effort that it is worth taking to do a good job on the testing. Informally, we may classify the importance that the project performs appropriately as follows.

High Importance. A particularly important project at the heart of company profitability for which it would be extremely costly (in time, effort, prestige) if the project were to develop major failures in the field, and where it is very important that software is good.

Medium Importance. A fairly typical project, for which poor performance will be 'typically' costly, and where it is important that software is good.

Low Importance. The kind of project where it does not really matter whether the software is good. For example, the project might still be in a trial phase, or concern areas with little consequence, or the software may be a simple stopgap, until a better solution is found; code quality is not a major concern.

The effect of this classification is to determine one, or several, multipliers on the consequences of different types of failure after release. Of course, it is really a crude discretisation of (several) continuous quantities. If we were intending to carry out a full decision analysis, then such a quantification would doubtless be important. However, we are really trying to illustrate the qualitative structure of the assessment. Further, as soon as we try to elicit precise quantitative effects, we will tend to face many practical, psychological, and political types of issue, whereas it is often quite quick and easy to get agreement as to a broad categorisation as above.

Therefore, for each factor, here and below, we suggest three levels of importance: high, medium and low. By *high importance* we mean that improved testing is always a major issue. By *low importance* we mean that testing is not a major concern, according to the particular aspect under consideration. If we eventually implement the initial recommendations presented in this document, then we may reword or reclassify or add new levels to fit better into the company's particular milieu. Although it is not difficult to make more detailed distinctions, the relevance of the different levels should be fairly clear.

If we do wish to refine the judgement, then we might introduce a cost vector, representing, say, the expected cost of each minor, major and showstopper error found after release.

7.2.2 Initial error level of software

The importance of testing will obviously depend on how many errors the software contains initially. Clearly, this is unknown, but an assessment can be made into the following categories.

High error level. The software is likely to contain important faults. Testing is very important.

Medium error level. Software quality is uncertain. Some testing is required.

Low error level. Testing is mainly a formality.

The effect of this classification is to scale the probability of faults existing a priori in the software, and so likely to be remaining in the software after testing. We could further break down our information about the software according to the following considerations.

- Is the software substantially new, or largely a routine rewrite of existing, good quality software?
- Is the area of application of the software relatively straightforward (and so more likely to be error free), or novel and difficult to program reliably (and so more likely to contain errors)?
- Is the software long and elaborate, or short and simple?
- Is the software is produced by a reliable team, or not?

We could build up a simple assessment procedure based on this type of consideration to give probabilities for each type of level of software deficiency. Again, we might introduce a probability vector for the different types of error for each category.

7.3 Specific assessments

7.3.1 Test cost

We suppose that company assigns a testing budget, B, using their current methods of assessment. B has two components: money, M, and time, T. Each of these is a prior expectation, which may be modified, but they are both set fairly early in the process, to some order of magnitude. For simplicity, we suppose that each element M and T may be classified as follows.

High expense. Sufficiently expensive that reducing test cost is a major concern.

Medium expense. There may be some scope for cost reduction, but this is not a driving consideration.

Low expense. Cost reduction is not an issue.

Of course, these classifications are relative to the constraints of the problem. For example, a testing budget is expensive in time if the expected testing time gets very close to the promised release date of the software. Time enters the costings indirectly as a constraint. For example, if time cost is high and overall test quality is poor then we will seek to use time more efficiently to improve testing.

7.3.2 Error level of testing

Within their current approach, suppose that the company now assesses how effective they believe their testing regime is likely to be. Presumably most testing is not planned to be poor, though time constraints and so forth will mean that full testing (whatever that might be) is not always possible. Again, to keep things simple for purpose of illustration, suppose that the company considers three categories of test deficiency. Note that we are concerned here with errors which could be caught by the testing process. Any further errors which can only be detected after release are beyond the scope of this analysis.

High deficiency. There is large uncertainty as to whether testing will find all high priority faults. For example, some important areas of functionality might only be superficially tested. Improved testing is a priority.

Medium deficiency. There is reasonable confidence that all high priority faults will be found, but less confidence that all major errors will be found. There is scope for improved testing.

Low deficiency. There is confidence that all highest priority and other major faults will found, in sufficient time to fix before release. Improving test effectiveness is not considered important.

This classification is made by comparison with similar testing problems addressed by the company. In practice, this is essentially how budgets are assigned, in that expert testers make assessments of the resources required for careful testing, by analogy with previous testing experiences, and then by negotiation settle on a time frame and resource budget with which either they are happy (presumably as they feel this will lead to low test deficiency) or which they accept as a compromise involving some degree of test deficiency. As with each of the other inputs, we could refine the categories by discussion with test managers if we wanted to implement this approach.

Note that test error level refers to lack of confidence in the *test procedures*, not in the quality of the software after testing. For example, if we did no testing at all, then we would have no confidence in the testing (obviously) but if we had high confidence that the software was of good quality then we would still be happy to release the software. Effectively, we are dividing the belief specification problem into two parts. First, the chance of errors in the software (initial error level), and secondly, the probability of finding such errors given that they are in the software (testing error level). This decomposition is made to make it easier for the company to specify their judgements, and because software deficiency will be a common feature to the problem irrespective of the testing approach, but test deficiency strictly applies to what the company can achieve without the BGM approach. In the same way, in financial considerations, project importance is common to all approaches but current test budget only relates to the company's choice without the BGM approach. Again, we may specify a probability vector for the ability to find each type of error.

7.4 Use of assessments

We now consider the potential for the BGM approach to be worth considering for a new project. The financial considerations are summarised by the cost assessments, whilst the confidence in releasing good quality software is summarised in the initial and test error levels. The advantage of starting with simple classifications is that for many cases the conclusion will be fairly obvious without detailed calculation, for example if all costs and error likelihoods are assessed as high then we would expect the BGM approach to be very helpful, while if all are assessed as low then presumably we would expect much less benefit from more sophisticated approaches. In practice, we suspect that for many problems this cross-classification will quickly reveal whether the BGM approach should be explored, as the potential for improvement can be gauged from the number and nature of high values in the specification. If we do seek a quantification, the overall expected cost to the company is of the form

$$[\text{Project Cost}] * [\text{Initial Code Error}] * [\text{Test Error}] + [\text{Budget Cost}],$$

suitably scaled, and possibly based on vectors of costs and probabilities. However, it is more fruitful to use the above assessments to direct the amount of effort we put into the BGM approach analysis, as follows.

7.5 Assessment of the BGM approach

Set against the various advantages of the BGM approach (some of these advantages, such as the availability to managers of detailed justifiable assessments of software reliability before and after testing, are not easily tangible in money terms), are the costs of implementing the approach. Essentially, the extra cost that is incurred is the additional effort required to construct the Bayesian models for the testing problem. Of course, how easy or difficult it is to construct the model depends strongly on the effectiveness of the tools that are put in place by the company to implement the approach. Further, much of the work required to construct the BGM approach model is required of any approach to construct the test suite, and, given good tools, the cost, in complicated testing problems, may actually be less than for the alternative approaches, as the BGM approach may make it straightforward to generate efficient test designs. Finally, the BGM approach may be carried out either in full, or in cost-saving approximations, depending on the amount of time and thought that goes into the structuring and specification.

The BGM approach exploits the relationships between different test outcomes to transfer information from the outcome of one test to beliefs about potential failures elsewhere in the software. The more complex the diagram, the more the potential to exploit such transfers of information, but also the higher the cost in producing the diagram. Rather than making a somewhat arbitrary guess at the potential savings that a general BGM approach might yield, it seems more sensible to suggest how to match the amount of effort that should go into the BGM approach modelling with the requirements of the problem at hand, so that the BGM approach should always give good value.

In any case, we start with a description of the possible tests that can be run. This is essentially cost neutral, as we must make such an assessment whichever approach we take. We would imagine, however, that the general features of the problem (project cost, software quality) will influence the level of detail at which it is thought worth describing the observation space.

Now we describe various choices for the level of detail for the BGMs that we shall construct. We identify three aspects to this, and for each aspect we describe three levels: *full, partial*, and *simple.*

Outcome space description. The level of detail at which we *describe* the test outcome.

> **Full:** the finest level of detail that we might consider;
>
> **Partial:** an intermediate level of detail for the outcome space, aggregating outcomes which are almost equivalent;
>
> **Simple:** makes as few distinctions as possible.
>
> As a general principle, the more important it is not to overlook errors (that is, the higher is project cost), then the more detailed we will want the outcome space description to be, as it will be important not to overlook error types.

Model linkage. The level of detail of the *links* between different parts of the model.

> **Full:** makes all of the links between outcome nodes, through higher order structuring, that we can identify;
>
> **Partial:** keeps the main links, but drops link nodes which appear to have little influence (as they are very unlikely, or are likely to have small arc values).
>
> **Simple:** only adds unavoidable links to the diagram, minimizes the number of parent nodes.
>
> As a general principle, the more important it is to reduce the testing cost, i.e. the higher is the test budget, then the more detailed we will want the linkage to be, as this allows us to reduce probabilities in parts of the diagram based on tests in other parts of the diagram, and so reduce the number of tests.

Probability elicitation. The care with which we *elicit* probabilities for the model.

> **Full:** belief specification is made carefully for each node;
>
> **Partial:** some use is made of of exchangeability (similarity) to simplify specifications which are almost exchangeable;
>
> **Simple:** belief specification is based on a minimal number of exchangeable judgements and simple scaling arguments across parent nodes.
>
> As a general principle, the more we suspect that we will not be able to carry out full testing (that is, the higher is the test error level), then the more careful we will need to be in our prior elicitation, as we will not be able to drive all error probabilities to near zero, so some of our assessments may be strongly influenced by prior values.

The higher the initial error level of the software, the higher the level we prefer for each of the above three criteria.

7.6 Choosing an appropriate Bayesian approach

We now describe how to put the various ingredients together. There are three essentially different ways that the BGM approach may be valuable.

1. For the same test budget as for the existing company approach, it may provide a more effective test suite and thus reduce errors in software release.
2. It may achieve the same effectiveness as an existing company test suite constructed according to current practice, but using using less test effort and so reducing test budget.
3. The company may judge that there is an intrinsic "good practice" value in following an approach which offers a more rigorous and structured approach to testing, which constructs models for the test process which can be maintained over a series of software releases, which provides probabiistic assessments of software reliability which can be fed directly into any risk assessment that must be made about software release, and which can be used to monitor and forecast time until completion of the testing process and so give advance warning of problems in meeting test deadlines.

Our main objective is to reduce error when the project cost is high and to reduce budget when the budget cost is high. We suggest the following choices for setting the levels for the three aspects of the Bayesian specification.

1. Start with medium levels for outcome space description, model linkage and elicitation.
2. If the initial software error level is high (low), then raise (lower) each of the three levels.
3. If the project cost is high (low), then raise (lower) the level of outcome space description.
4. If budget is high (low), then raise (lower) the level of model linkage.
5. If test error level is high (low), then raise (lower) the level of detail of prior elicitation.
6. Consider raising the various levels to realise more fully the "good practice" value of the testing.

Obviously, we may suggest further rules, and add further criteria. However, this is intended to be illustrative of our intention to produce an approach which is appropriate to each problem, rather than a use or do not use rule with some arbitrary cutoff.

8 Summary

This chapter has presented a brief review of the Bayesian Graphical Models (BGM) approach to software testing, which the authors developed in close collaboration with industrial software testers. It is a method for the logical structuring of the software testing problem, where focus was on high reliability final stage integration testing. The chapter presents discussion of a range of topics for practical implementation of the BGM approach, including modelling for test-retest scenarios, the expected duration of the retest cycle, incorporation of multiple failure modes and diagnostic methods. Furthermore, model maintenance and evolution is addressed, including consideration of novel system functionality. Finally, end-to-end testing of complex systems is discussed and methods are preseented to assess the viability of the BGM approach for individual applications. These are all important aspects of high complexity testing which software testers have to deal with in practice, and for which Bayesian statistical methods can provide useful tools. This chapter presents the basic approaches to these important issues. These should enable software testers, with support from statisticians, to develop implementations for specific test scenarios.

Acknowledgement

The authors gratefully acknowledge the provision of funding for much of this work from British Telecommunications plc and input on its industrial direction.

References

[1] Bedford, T. and Cooke, R. (2001). *Probabilistic Risk Analysis: Foundations and Methods.* Cambridge University Press.

[2] Coolen, F.P.A., Goldstein, M. and Wooff, D.A. (2003). Project viability assessment for support of software testing via Bayesian graphical modelling. In: *Safety and Reliability*, Bedford and van Gelder (eds), Swets & Zeitlinger, Lisse (Proceedings ESREL'03, Maastricht), pp. 417-422.

[3] Coolen, F.P.A., Goldstein, M. and Wooff, D.A. (2007). Using Bayesian statistics to support testing of software systems. *Journal of Risk and Reliability* **221**, 85-93.

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

[5] Goldstein, M. and Wooff, D.A. (2007). *Bayes Linear Statistics: Theory and Methods.* Wiley, Chichester.

[6] Jensen, F.V. (2001). *Bayesian Networks in Decision Graphs.* Springer, New York.

[7] Wooff, D.A., Goldstein, M. and Coolen, F.P.A. (2002). Bayesian graphical models for software testing. *IEEE Transactions on Software Engineering*, **28**, 510-525.