Risks in software development with imperfect testing

Tapan P. Bagchi

Vinod Gupta School of Management and
Industrial Management and Engineering,
Indian Institute of Technology Kharagpur,
Kharagpur 721302, India
E-mail: bagchi@vgsom.iitkgp.ernet.in

Abstract: This study analytically assesses the risk of releasing defective software that cannot be exhaustively tested and of needlessly testing defect-free software. Specifically, it quantifies the probability of committing a Type II error ($\beta$) in software development when one may release software that still is faulty while the test methods themselves may not be perfect. The study uses truncated Poisson and geometric distributed path lengths and Bernoulli-type inspection errors to link $\beta$ to software design features, the development philosophies employed and certain aspects that include code quality, cyclomatic complexity and the average length of basis paths. For risk reduction, this study finds quantitative justification for raising test coverage, perfecting the test methods, the adoption of recent innovations and programming methods such as component-based design, SOA and XP as ways to raise the likelihood that the product developed will be fault-free. Results are relatively robust with respect to the probability distributions assumed.

Keywords: software engineering; SW testing; defect modelling; Type I and Type II errors; imperfect inspection.


Biographical notes: Tapan P. Bagchi is a Professor at IIT Kharagpur, India. He received his BTech in Mechanical Engineering from IIT Kanpur (1967) and PhD in Operations Research from the University of Toronto (1971). He is also a Fellow of the Institution of Engineers (India). He is a registered Professional Engineer in Ontario. In 1972 onwards, he served EXXON-Mobil in manufacturing management, process engineering, facilities debottlenecking, product engineering, quality engineering, automation, technology transfer and corporate planning. He is the author of six texts in computer science, quality engineering and applied mathematics. Since 1987, he has served IIT Kanpur, NITIE India and SP Jain Institute variously as Professor, Department Chair, Dean and Director.

1 The contemporary software development scenario

A leading European bank (the client) recently contracted with a major software (SW) developer (the provider) for a product that would help manage all of the bank’s debit card accounts. Out of curiosity this author asked one of the provider’s senior systems
specialists about how they would go about doing this assignment, specifically, how would they ensure superior quality of the product to be developed? The process seemed typical: The client would develop the project concept considering its business needs and do preliminary cost-benefit analysis, following which it would ball park its budget and float the request for proposal. Interested providers would respond and a selection would be made. Following this, the selected provider would execute a key QA step – they, jointly with the client’s designee, would do requirements analysis leading up to GUI specs, use cases and a prototype – all to be signed off by the client.

The client’s systems architect and the provider’s architect would then jointly determine systems specs. This might include identification of databases and some key design features of the system to be developed. Such close involvement of the user/client in SW development is now a well-established QA practice, reflecting ownership, agreement on the scope and the client to experience minimal surprise with what is delivered (Pressman, 2005). Following this step development would proceed with coding, unit testing, integration testing and finally user acceptance testing. Tests would use test cases to the extent possible within the agreed upon budget and delivery date. The systems specialist admitted that while absolute assurance of faulty-free quality could not be given, the provider would utilise the most sensible design and test methods to ensure that the final product would perform at a high level of integrity. Subsequent to product release, the provider would extend limited production run support, minor enhancements and scope further work using enhancement requests. However, the systems specialist took particular care to observe that the provider had recently decided to switch to a SOA-type architecture – specifically, building the system as an ‘assembly of small modules’ each covering only a specific function – in all its significant SW development projects.

This new stance was provoked by the provider’s cumulative experience of SW QA gathered over a decade’s trouble-shooting, testing and debugging over 25 large systems developed and sold to clients. In the new approach the provider would encourage coding the system as an assembly of many small modules, each handling one distinct function, coded and tested separately. For instance, in handling normalised data tables, the coder was encouraged to stay away from complex SQL-type constructs in coding, using instead a function coded to take care of a distinct operation to be performed with each table. The final product was built by assembling these small modules into a total system to meet the user’s needs. This author wondered if such a move was not just trendy, following the media hype of 2006/2007 about service oriented architecture (SOA) being the ‘cure-all’ (and in part to save money by outsourcing some of a corporation’s IT tasks to web-enabled off-site services) to all malice attributed to frizzy-haired creators of SW.

This author’s reaction was an analytical study of two QA issues of the SW development –

\[ \text{a} \] not releasing a fault-free system that a flawed test case ‘fails’ and more seriously

\[ \text{b} \] delivering a faulty SW to the client, since many large systems by their very size and complexity often negate exhaustive testing of every aspect of the creation.

The latter situation is similar to decision making by the method of statistical hypothesis testing when the ‘population’ is just too big, preventing the decision maker to unequivocally establish the true state of nature (Rice, 2007). An additional reason for the present attempt to model SW testing errors analytically was to address the desire
expressed by many that SW testing must become practical rather than impossible, random or hit-and-miss (Mallory, 2002). This paper reports the inferences drawn from this study.

2 SW defects discovery – some empirical evidence

2.1 SW bugs

A ‘SW bug’ causes a program to produce invalid output or response or to crash or lock up. The problem is usually erroneous or insufficient logic. Thus, a program may crash if not enough validity checks are performed in the input or on the calculations being performed. Bad logic may direct execution to a place in the program where the predicate expected does not exist. Such a program would also crash with the consequence of penalties to developers, loss of essential services to users and even threats developed to life and/or property. Pressman (2005) provides the description of contemporary practices that directionally reduce bugs and defects in SW. A key strategy is testing the SW, similar to inspection done in hardware manufacturing to ensure quality. For ‘inspecting’ the SW, various approaches – unit testing, black box, white box, integration testing, validation testing, system testing and regression testing, etc. – have been devised and incorporated in practice.

But testing is reactive – it looks out for the discovery of defects in the SW after it has been produced. In inspection-based quality assurance there is no attempt to control the process of SW development so as to minimise the occurrence of bugs. So in a way the ‘testing’ approach to SW quality control parallels inspection-based quality control after the product has been produced practiced by yesterday’s manufacturers of hardware, an approach that is increasingly getting replaced by proactive off-line quality assurance methods (Juran and Gryna, 1993). In fact, hardware producers now attempt to design around the causes of failure with the help of advanced techniques that include design of experiments and several other techniques. Such methods empirically track down the root causes or factors that cause failure or malfunction, often some aspect of the product’s design. Not attended to, such factors would negatively affect the final product’s performance in the hands of users. SW development is not at such stage of control yet; statistically designed experiments to help identify sources of SW errors have only recently begun to be used (Banks et al., 1998; Jayaswal and Patton, 2007). However, if certain controllable structural aspects of a SW that correlate with the faults present in the SW could be identified, it might then be possible to control these factors proactively. This would reduce the conventional dependency on testing the SW as the sole method to assure its quality.

Use of quantitative methods in SW engineering is relatively recent. The CMMI philosophy covertly nudges one towards these. For object-oriented systems the Succi-Stefanovic-Pedryez model (Succi et al., 2001) quantitatively determines the dependence of defects on two design aspects, namely, inheritance and communication between classes. It employs statistical models based on Poisson regression and negative binomial regression to determine the dependency of faults on these two design factors. Using empirical validation, this approach can identify the most critical classes, thereby assisting the developer in deciding on the rigor of tests he would plan and run. Analytical models have also been built to predict the reliability of SW (Gokhale et al., 1998).
2.2 Cyclomatic complexity

The intricacy of a program module – a measure of its complexity for QA purposes as well as for determining the number of tests that ought to be run on it – influences its fault-free performance on release (Mall, 2004). Developed by McCabe and Watson (1994), a complexity measure known as cyclomatic complexity (CCN) counts the exact number of linearly independent paths through a program’s source code. This measure is an indicator of the expected soundness of the SW when released and is calculated by creating a control flow graph of the source code in order to determine all the execution pathways of the program. To fully test a SW module, ideally all possible execution paths in it should be exercised. Thus, a module with high CCN would require more testing than one with lower complexity. Also, a module with high complexity would tend to have low cohesion. CCN is independent of language and language format and studies indicate a correlation between a program’s CCN and its error frequency. Additionally, lower complexity also contributes to a program’s comprehensibility and maintainability and its testability under various usage conditions. Thus, expressed numerically as an integer, a CCN of 1–10 would be regarded as a simple program entailing low risk of being faulty when released, 11–20 would entail moderate risk and 21–50 being quite complex and difficult to exhaustively test using test cases would implicate a high level of risk of being faulty on release.

NASA (Rosenberg and Hammer, 1998) identified a few metrics that help in assessing the reliability of a SW. Such metrics would assist in the evaluation of a SW’s ability to perform its required functions under stated conditions for a stated period of time. NASA also articulated the different terms used when one talked about SW quality. An ‘error’ is a programmer action or omission that results in a ‘fault’. A fault is a SW defect that causes a ‘failure’, an unacceptable departure of a program operation from program requirements. NASA asserted the size of modules to be a quality indicator of SW and suggested that error-proneness of SW is a combination of its size and its complexity.

About errors, an empirical study by Shooman and Bolsky (1975) found that many errors in a program comprising 4,000 machine instructions could be found cheaply by hand processing, without running a computer. A study by Ostrand and Weyuker (2002) related the size of code components to the number and density of faults in them. This study, which tracked 13 releases of a large inventory tracking SW, found no evidence that the larger modules had higher fault densities than smaller ones. Actually, they found that the reverse was true: files fewer than 100 lines had between ten and 75 faults/KLOC, whereas those with more than 1,000 lines had two to three faults/KLOC. A widely cited empirical study by Basili and Perricone (1984) also found the surprising result that module size does not seem to raise error proneness. In fact, they found that the larger the module, the less error prone it was, even when the larger modules were more complex.

In the last three decades, a number of major SW failures have occurred setting off a great deal of introspection and also modification of industry practices. The outcome is innovations in SW development, testing approaches and tools that assist in delivering reliable SW. However, most of this development is empirically-driven – by experience with various practices and rarely by quantitative analysis. Thus, many basic questions are yet to be answered.

CCN was created to help objectively evaluate code quality. It tells definitively that a complex code is risky as it is likely to be faulty. Based on graph theory, this metric is able to locate the knotty areas of the code that need careful testing and special effort to
Risks in software development with imperfect testing

maintain them when changes to it are made. SEI recommends risk assessment benchmarks based on CCN. Nevertheless, from coding methodology standpoint, some methods do indeed reduce complexity. These include test-driven development (TDD) (Newkirk and Voronstsov, 2004) and the SOA (Bell, 2008). SOA avoids monolithic spaghetti codes by creating a collection well developed ‘service’ modules that individually do specialised jobs, ready to be utilised when an executor calls upon them on an as needed basis to realise overall information processing. SOA services are relatively small and perform dedicated functions. Held in a library, services – chunks of well-built codes – are continually re-used.

Thus, by splitting complex jobs into smaller services SOA avoids the extent of complex logic in codes. TDD, on the other hand, uses short iterations of intense code-test-code-test cycles to develop SW. For each incremental functionality TDD tests are written before the functionality itself is coded. New codes must pass testing before the next functionality is coded. It is observed that high CCN rarely appears in a system coded in this manner (Beck, 1999). Both SOA and TDD improve the maintainability of SW as well.

The natural extension of TDD is ‘extreme programming’ (XP) (Pressman, 2005; Beck, 1999), which aims at reducing the cost of change. XP is a style of development that starts with the simplest solution. Extra functionalities are incrementally added later. XP maintains very close contact with the user and thus readily accepts user feedback for performance and the next requirement to be coded. XP uses automated unit tests to ensure what is coded is what one wants. Subsequently, XP uses acceptance tests to ensure what the user conveyed when he stated his requirements was actually what he meant to convey. Also, XP organises the logic in the system such that changing one part of the system would not affect other parts. There is no big design up front with XP.

But exhortations (Nagpal, 2008) such as ‘lower your CCN’, ‘refactor when CCN gets too high’, ‘code in pairs – do ‘code-test-code-test’ as your routine’, etc. do not themselves provide clear-cut directions for creating bug-free SW, nor are they inherently compelling. Should one strive to keep individual module sizes small? What would one gain? Would close-knit quality assurance as promoted by the XP methodology lead to fewer errors at delivery? Should one use SOA in all developments? This paper probes such questions quantitatively.

3 Risks in passing judgment on SW based on limited testing

In statistical hypothesis, testing it is assumed that sometimes it is not possible to completely remove uncertainties surrounding a decision situation (Rice, 2007). This leads to working with limited information about the ‘state of nature’, which in turn can lead to four distinct possibilities in the decision situation confronting a decision maker. First, if the true state of nature is that a hypothesis is correct and the test accepts that hypothesis as true, no error is committed. The hypothesis in the present context is ‘The SW being tested is fault-free’. However, the second possibility is that one rejects a correct hypothesis based on the data gathered from a test conducted. A ‘Type I error’ is committed here. The third possibility is that the hypothesis is false but one accepts it based on the limited information revealed by the test. Here a ‘Type II error’ occurs. Lastly, one commits no error if one rejects a false hypothesis. If not all basis paths are exhaustively tested, even if no defects have been found up to the time tests are stopped, it
is probable that some defects would escape and the tester would erroneously release defective SW (Type II error in Table 1).

Table 1 Type I and Type II errors in hypothesis testing

<table>
<thead>
<tr>
<th>Your decision based on the tests conducted on the software</th>
<th>True state of nature</th>
<th>Software is truly defect-free</th>
<th>Software has defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare that ‘Software is defect-free’</td>
<td>Decision is correct</td>
<td>Type II error: you have erroneously accepted a defective SW as good</td>
<td></td>
</tr>
<tr>
<td>Declare that ‘Software is defective’</td>
<td>Type I error: you have incorrectly rejected a good SW</td>
<td>Decision is correct</td>
<td></td>
</tr>
</tbody>
</table>

In this paper’s view, the risk in not testing all basis paths and terminating the test is the expected loss from the ‘probability’ of committing a Type II error. The reverse (committing a Type I error in SW testing) is possible if the testing methods are not perfect. This is when the test cases used themselves are defective and are unable to detect a coding or logical defect that is actually present. In setting up statistical hypothesis testing one attempts to design the tests so as to minimise the two risks – by minimising $\alpha$, the probability of committing a Type I error, as well as $\beta$, the probability of committing a Type II error.

In ‘white box testing’ a SW, not all basis [i.e., linearly independent (Pressman, 2005)] paths may be exercised by the test cases applied, either due to limited budget or because the SW itself is too complex or too large (Pressman, 2005; Mall, 2004). A version of the MS Word® apparently consisted of > 4000 basis paths, negating any realistic effort to test all such paths exhaustively. Even if one is able to determine the CCN of a code, it may not be feasible to design test cases that would exhaustively exercise all the basis paths. This may lead to releasing the SW as ‘accepted’ when not all tests on it have been completed, ensuing the consequence of releasing defective SW. This situation – decision-making based on limited knowledge about the true quality of a SW – parallels the domain of statistical hypothesis testing. SW test planners have not yet adopted such an approach to SW quality assurance or SW design. In this paper, we attempt to quantify the risks of releasing defective SW or rejecting defect-free SW using non-exhaustive tests. We utilise the principles of statistical hypothesis testing. We also spot structural features of SW that may tell upon the quality of a SW created.

First, we find the probability of committing Type II error in testing a SW when we do not exhaustively exercise ‘all’ distinct ‘basis paths’ present in a code. Subsequently in Section 4, we quantitatively estimate the probability of committing a Type II error in randomly testing a ‘single’ basis path.

3.1 The probability of committing Type I error

As noted in Table 1, a Type I error is committed when based on limited or faulty testing one erroneously rejects SW that is truly defect-free. This can happen if the test cases are improperly designed or if the tests are improperly run and therefore yield incorrect results. The consequence of committing a Type I error is a false signal and the waste of resources on pointless ‘debugging’ exercises. Type I errors may delay the release of a SW that really has no bugs.
Type I error can be minimised by properly designing the test cases and by thoroughly training the tester on the procedures and protocols of conducting the tests. We shall presently assume that the organisation has the means to minimise Type I errors in SW testing. Type II errors, on the other hand, are dependent on certain structural features and the inherent quality of the code, as we show below.

3.2 The probability of committing Type II error

In contrast to Type I error, committing a Type II error is quite serious as it may lead to the release of a defective SW in the marketplace. A defective SW may lead to rework and returns, labourious maintenance and it may even harm the client’s operations financially, cause unsafe actions or delay operationalising critical functions. Therefore, one must make every effort to prevent or minimise the probability of committing a Type II error, i.e., releasing a SW with unsatisfactory or unacceptable performance. We address its estimation in this section.

Let the $CCN$, i.e., the number of basis paths present in the code, be $CCN$. Assume that the code has been partially tested and the number of basis paths successfully tested when the test is stopped is $T$, $T$ simulating coverage. Hence, the number of basis paths yet to be tested is $CCN - T$.

An untested basis path may have one or more defective nodes in it. By the acceptability protocol in force, even one defective node present in the untested basis paths would render a SW defective. Let $\eta$ be the code quality or the probability that a given node is defective. Further, let $\eta$ be the average length (measured by the count of nodes on it) of a randomly selected and untested basis path. Therefore, average number of nodes yet to be tested when testing is prematurely stopped $= \eta (CCN - T)$.

Hence,

\[
P[\text{one or more defective nodes in } \eta (CCN - T) \text{ nodes}] = \sum_{r=1}^{\eta (CCN - T)} P[r \text{ defectives in } \eta (CCN - T) \text{ nodes}] \tag{1}
\]

Therefore, RHS in above becomes $1 - P[r = 0]$ or

\[
1 - \varepsilon^0 (1 - \varepsilon)^{\eta (CCN - T)}
\]

\[
1 - (1 - \varepsilon)^{\eta (CCN - T)} \tag{2}
\]

hence $\beta$ or the $P[\text{Committing a Type II error in non-exhaustive testing}]$ equals $1 - (1 - \varepsilon)^{\eta (CCN - T)}$. Based on this the following special situations may be easily evaluated. It is trivial to see that $\beta$ reduces as $\varepsilon$ is reduced (i.e., code quality is improved) or if $CCN$ is lowered.

If the exhaustive testing is successfully completed, i.e., all $CCN$ basis paths have been successfully tested, $T = CCN$, hence

\[
P[\text{Committing a Type II error}] = 0 \text{. If no tests have been run yet,}
\]

\[
P[\text{Committing a Type II error}] = 1 - (1 - \varepsilon)^{\eta CCN}
\]
And if $\varepsilon = 0$,

$$P[\text{Committing a Type II error}] = 1 - (1)^{\eta(\text{CCN} - T)} = 0.$$ 

Each result above is consistent with our expectations. Figure 1 displays the typical sensitivity of the probability of committing a Type II error to $\varepsilon$ and the fraction of total basis paths tested when testing is prematurely stopped without testing each of the $\text{CCN}$ basis paths. Figure 1 also indicates that the factor effects on Type II error are not additives; they interact (Rice, 2007). For the data displayed, $\text{CCN}$ was 50 and average basis path length $\eta$ was set at 10. The values set for code quality $\varepsilon$ — the probability of a node being defective — were based on Table 9 of Basili and Perricone (1984). Figure 2 shows the sensitivity of $P[\text{Type II Error}]$ to $\text{CCN}$ and $\eta$.

**Figure 1** Probability of declaring a defective SW to be good when not all $\text{CCN}$ basis paths have been tested and passed (see online version for colours)

**Figure 2** Probability of declaring a defective SW to be good as function of $\text{CCN}$ and avg. length ($\lambda$) of basis paths (see online version for colours)
Before we conclude this section, we remark that the ongoing debate about the relationship between the CCN of a code and its error-proneness is often a comparison of apples and oranges. This would include the phenomenon tagged ‘Goldilock’s Conjecture’ (Koru et al., 2007). No studies, to the best of the author’s knowledge, have documented the level of code quality $\varepsilon$ (the probability of a single node being defective) in these comparisons. The larger and more complex systems may have been more carefully coded (i.e., inherently have lower $\varepsilon$) (Banks et al., 1998). In the absence of such information, comments on size or complexity not apparently being related to a code’s error proneness are likely to be misleading. See Binstock (2008) for an instance.

4 A model for SW defects and testing strategies based on size

4.1 Testing a path

We investigate a different issue in this section – the probability of missing defects in a single randomly run test. In such a test, only one basis path is executed.

As stated above, a SW cannot be treated to be properly tested until all of the code (really all of its basis paths) has been executed. Often a SW comprises segments with paths running between these segments, not merely through the segments themselves. In an unstructured program, therefore, exhaustive testing is nearly impossible. This implies that an unstructured program is very likely to deny attempts to test it exhaustively.

Thus, from testing standpoint, the single entry, single-exit construct provides a valuable abet to quality assurance by testing (Mallory, 2002). We assume therefore that such control is being exercised when a SW is being created as a collection of processing entities (nodes) and interconnecting paths. The initiator is the ‘starting node’ where the initial input is given. Subsequently, based on the SW’s response/behaviour/design, the user interacts and provides further inputs. In normal operation, through state transitions, the SW moves from node to node along paths and produces the desired output (Figure 3).

We use here the definitions given in Section 4.3.1 of Jorgensen (2002) of nodes and edges. Figure 3 is an adaptation of various program graphs provided in Jorgensen (2002).

**Figure 3** A simplified ‘node-path’ representation of a SW (see online version for colours)
Some interesting and yet critical conditions may occur in testing a SW (Mallory, 2002). We recognise that a SW cannot be said to have been properly tested until all the code has been executed. However, one appreciates that a single successful execution of a code segment does not prove that the segment is free of defects. Testing therefore should include checking all the paths between segments, not only code segments themselves. Indeed if all paths between code segments can be executed successfully, then the SW can be certified as fully tested. In an unstructured program, this approach of testing is impossible to apply. In a structured program, this difficulty reduces. Ideally, of course, the single entry, single exit construct is noted to provide a valuable test environment. A directed graph with nodes and paths is an easy way to illustrate the segments and the accessibility of the different segments when test cases are being planned. Such a graph can also display the program’s internal logic structure.

However, certain conditions can make testing challenging and reduce it to random or hit and miss, raising the risk of faults going undetected. A simple loop can create virtual paths that cannot be tested. Further, missing controls cannot be tested since by definition they are not present. But their absence can create SW failures. A SW with incorrect or inadequate branch condition specification may exist. Here all paths may be individually exercised and tested, but some of the paths will be taken under wrong conditions (Mallory, 2002). One may even have paths that are impossible to reach. Size alone, however, may prevent one from doing exhaustive testing exposing the user or the developer to the associated risk. This last point makes one intrigued about a relationship that might exist between the defects left in a SW created and the size of the code. In this section, we attempt to uncover such dependency with the hope to evolve helpful hints for developers.

In testing the acceptability of a well structured SW, the following protocol is generally followed. Every time the SW is tested, it is initiated with an input. It then traverses a path through a random string of nodes. The test may be repeated as many times as permitted by the testing budget. If all planned test runs lead to a ‘pass’, the SW is declared OK. We note that the particular path traversed during such tests depends on

a the input

b user interaction

c the dynamics as it evolves based on the logic as coded in the SW.

We consider the case when $CCN$ of the SW – is large. Here not all $CCN$ test cases can be run. The (particular) path traversed in a test run, generally speaking, is random. Even if user interactions are ordained, widely different paths may be generated and executed by the different test cases. This results from varying inputs in the different runs and also variations in the user’s interactions as dictated by the design of the test cases. Thus, if the number of debugging test cases run were less than $CCN$, not all paths would be tested in a large SW.

A SW would be regarded as ‘failed’ if any node in it is discovered to be defective. On the other hand, a good node would process data and information always as intended or designed. A SW is regarded to be defective if it has at least one defective node.

When the entire basis paths existing within a SW lead to desired and correct outputs, the SW is said to be free of defects and is declared to have ‘passed’.
If one or more node(s) on a particular path are defective (shaded dark in Figure 3), the path will produce an unsatisfactory response. The SW may then abort or generate incorrect and unintended response.

The goal in SW testing is to trace and test ‘as many paths as possible’ from input to output – to help assert that the SW system will always behave as expected and produce responses that are correct in each case. This would not be so if some nodes were defective. Note that if \( CCN \) is large, during the white box test less than \( CCN \) basis paths may be tested.

Ideally, therefore, to declare a SW to be free of defects, one should exhaustively discover and test each basis path existing within the SW that connects the input node to output/response. (In other words, if budget permits, \( CCN \) would be the upper bound of the number of basis paths that must all be tested.)

But this may be impractical or impossible to accomplish. So we must limit our testing effort to what the resources provided (budget) would allow. As the consequence, we should make the effort to quantify the risks inherent in such non-exhaustive testing.

As noted earlier, risks in SW testing are two-fold:

1. we may reject a defect-free SW by erroneously calling it ‘defective’ (~ Type I error in statistical hypothesis testing)
2. we may ‘pass’ a defective SW (one that has some defective nodes left in it) (Type II error in hypothesis testing).

The goal is to develop SW testing methods that minimise both above risks.

Type I error occurs if the test misclassifies a good node as a defective one or a correct response as unacceptable. One can minimise and even eliminate this by using correctly designed instruments (test cases) and training the tester thoroughly on tools and test protocols. As in Section 3, we assume that this has been done and therefore Type I errors would not be committed. Thus,

\[
\alpha = P[\text{Type I error}] = 0.
\]

Type II errors may occur in limited testing a SW when we do not test each and every node present or trace and traverse every basis path from input to output and diligently remove all defects and bugs identified. This, however, is impossible unless we do exhaustive testing. Still, good SW design and test strategies can help minimise \( \beta \), the probability of Type II error.

In this section, we develop a quantitative model for the probability of committing a Type II error in one random execution pass that goes from input to output. We then evaluate the sensitivity of Type II errors to certain parameters that get decided during the design of the SW. To proceed we make some assumptions, part to reflect the realities of the test environment and part to retain mathematical tractability.

### 4.1.1 Assumptions

1. The basis paths traverse every possible execution paths connecting input to output through the maze of nodes existing in the SW system.
2. A SW is defective if any node in it is defective (shown as \( \bullet \) in Figure 3). This implies that if some path is found to have a defective node on it, it would make little sense to proceed further without rectifying that defective node.
3 A SW is acceptable if ‘every’ basis path contains only good nodes (☐ in Figure 3).

In limited budget testing or in testing a large SW in which all possible basis paths may not be tested, the exercise can be a ‘lucky’ one in that the few paths tested do not contain any defective node(s). Such a ‘lucky’ test will falsely declare the faulty SW to be good (Type II error in testing).

4.2 Derivation of \( P[\text{Type II Error}] \) for a single randomly run test

As shown in Figure 3, we assume that defective nodes in a SW to be subjected to white box testing are randomly distributed within the coded bulk. Code quality determines the likelihood (\( \varepsilon \)) of a node being defective, which again for simplicity is assumed to be identical for all nodes. Further, we stipulate that a faulty SW will contain at least one defective node in it. Thus, if a basis path goes through \( n \) nodes, the probability that this path will be free of defective nodes is \((1 – \varepsilon)^n\) where \( n \) is the length of the basis path measured as the count of nodes on it.

The number of nodes \( N \) on a basis path will in general have a discrete distribution, \( P_n \), \( n = 1, 2, 3, \ldots \) In order to illustrate the derivation of \( P[\text{Type II Error}] \) for a single randomly run test that traverses a basis path from input to output we shall assume that \( N \) will have a truncated Poisson distribution (Rider, 1953; Plackett, 1953). Other discrete distributions with probability mass spread over non-zero integers could also be used if the situation would justify. The truncated Poisson distribution is obtained from a standard Poisson distribution with parameter \( \lambda \) as follows. If the parent distribution were Poisson with parameter \( \lambda \), we would have

\[
P_n = \frac{\lambda^n}{n!} e^{-\lambda}, \quad n = 0, 1, 2, \ldots
\]

But the length of basis path (i.e., the number of nodes it traverses in going from input to output) cannot be zero. Hence by eliminating the pathological case of \( n = 0 \) one may model path length by the truncated Poisson distribution (Rider, 1953; Plackett, 1953),

\[
P_n = \frac{\lambda^n e^{-\lambda}}{1 - P_0}, \quad n = 0, 1, 2, \ldots
\]

Here \( P_0 = e^{-\lambda} \) is used to normalise \( P_n \) (Rider, 1953) such that

\[
\sum_{n=0}^{\infty} P_n = 1
\]

The distribution given by equation (4) has the mean \( \eta \), which equals \( \lambda/(1 – e^{-\lambda}) \).

Therefore, \( P[\text{SW with code quality } \varepsilon \text{ passes a single test tracing a random path}] \)

\[
= \sum P[\text{path length } = n] \times P[\text{path is free of defects/path length } = n]
\]

\[
= \sum P_n \times P[\text{path is free of defects/path length } = n]
\]
Now we use the identity
\[ 1 = \sum_{n=0}^{\infty} \frac{\lambda^n}{n!} e^{-\lambda} \]
Hence
\[ 1 = \sum_{n=0}^{\infty} \frac{[\lambda(1-\epsilon)]^n}{n!} e^{-\lambda(1-\epsilon)} \]
\[ = e^{-\lambda(1-\epsilon)} + \sum_{n=1}^{\infty} \frac{[\lambda(1-\epsilon)]^n}{n!} e^{-\lambda(1-\epsilon)} \]

On rearranging the terms we get
\[ P[\text{a SW with quality } \epsilon \text{ passes a single random test}] = \frac{e^{-\lambda \epsilon}}{1 - P_0} \left[ 1 - e^{-\lambda(1-\epsilon)} \right] \]
where \( \epsilon = \text{Prob[ a node is bad]} \) and \( P_0 = e^{-\lambda} \).

From equation (5), it is possible to verify that if the SW is fault-free, i.e., if \( \epsilon = 0 \), then the probability that the SW passes any test is 1.0.

Equation (5) is an illustrative relationship that would help us relate the probability of committing a Type II error in a SW test initiated by an input and executing the SW with conditions involving the three factors denoted here by \( \epsilon, \eta \) and \( P_0 \). These are...
1 the probability that a node is bad ($\varepsilon$)
2 the average length ($\eta$) of the basis path traversed within the SW as the initial input and the various user interactions are processed
3 $P_o$, the probability that the initial input that triggers the start of the test itself is erroneous.

Recall again that equation (5) gives the probability of ‘passing’ the faulty SW as OK in a single application of the testing protocol.

Each test run would attempt to detect one or more defective nodes in the SW. If a defect is found, it is rectified and the fault is removed before proceeding with another test. Let $F_i = \text{Prob}[\text{the } i^{th} \text{ test run on the SW misses a defect}]$ [equation (5) evaluates this quantity]. If we run $k$ independent tests, then

$$P[\text{SW is fault-free}] = 1 - F_1 F_2 F_3 \ldots F_k$$

Now if $k = CCN$, then the tests run would have exhausted testing all nodes of SW, including any and all defective nodes. Hence at this stage,

$$F_1 F_2 F_3 \ldots F_{CCN} = 0$$

Because taken together, the $CCN$ tests would have corrected all defective nodes. At this stage,

$$P[\text{Sw is fault-free}] = 1 - F_1 F_2 F_3 \ldots F_{CCN} = 1 - 0 = 1$$

Figure 4  Sensitivity of probability of Type II error to $\varepsilon$ and $\lambda$

The sensitivity of the probability that a SW would pass a single random test may be numerically explored. We show the typical behaviour of the probability of committing Type II in Figure 4. $\eta$ here is the average number of nodes on a typical basis path; hence, it is indicative of the size of the code, while $\varepsilon$ represents its quality. Note that the effects of $\varepsilon$ and $\eta$ are not ‘additive’ – they ‘interact’ (Rice, 2007). We recall that we assumed the distribution of path lengths (i.e., the number of nodes on a basis path) to be truncated
Poisson and the quality of a node to be a Bernoulli distribution with parameter $\varepsilon$. It is theoretically possible to derive expressions resembling equation (5) if some other distributions govern these random dispositions of the code.

The above observation produces an important upshot. Table 2 indicates the zones (in the $\varepsilon - \eta$ space) where the classical unit test-oriented SW development at one end of practice and XP/XV or SOA style of development that would fall at the other end would reside. Therefore, it seems now that the classical unit test, integration test, etc. treat and test the modularised SW as something consisting of some number of modules interacting with each other. So, here the ‘nodes’ are themselves large in size and could have high $\varepsilon$ (lower code quality) but the SW has low $\eta$ (average basis path length).

On the other hand, in XP (Beck, 1999) (utilising methods such as XV (Herzinger et al., 2003) and in SOA, the ‘nodes’ (processing units) are numerous but each cover a small functionality, hence remain small in size and thus get tested well. So here, the nodes have low $\varepsilon$ while the SW may have very large $\eta$.

What this is indicating is that going the XP-XV (Herzinger et al., 2003) way or even going the way of SOA and designing by small modules is a good way to reduce Type II errors. In other words, the likelihood of releasing a defective SW could be low (very low?) with XP-XV or even with SOA – when modules are kept small, each addressing only a specific function that can be tested well resulting in high code quality (i.e., low $\varepsilon$), regardless of the average length of basis paths.

### Table 2

Typical sensitivity of $\beta$ (the probability of Type II error) to $\varepsilon$ and $\eta$ (see online version for colours)

<table>
<thead>
<tr>
<th>$\eta = 10.00045$</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.952</td>
<td>0.862</td>
<td>0.780</td>
<td>0.706</td>
<td>0.638</td>
<td>0.578</td>
<td>0.523</td>
<td>0.473</td>
<td>0.428</td>
</tr>
<tr>
<td>0.02</td>
<td>0.862</td>
<td>0.706</td>
<td>0.578</td>
<td>0.473</td>
<td>0.387</td>
<td>0.317</td>
<td>0.260</td>
<td>0.213</td>
<td>0.174</td>
</tr>
<tr>
<td>0.03</td>
<td>0.780</td>
<td>0.578</td>
<td>0.428</td>
<td>0.317</td>
<td>0.235</td>
<td>0.174</td>
<td>0.129</td>
<td>0.095</td>
<td>0.071</td>
</tr>
<tr>
<td>0.04</td>
<td>0.706</td>
<td>0.473</td>
<td>0.317</td>
<td>0.213</td>
<td>0.142</td>
<td>0.095</td>
<td>0.064</td>
<td>0.043</td>
<td>0.029</td>
</tr>
<tr>
<td>0.05</td>
<td>0.638</td>
<td>0.387</td>
<td>0.235</td>
<td>0.142</td>
<td>0.086</td>
<td>0.052</td>
<td>0.032</td>
<td>0.019</td>
<td>0.012</td>
</tr>
<tr>
<td>0.06</td>
<td>0.578</td>
<td>0.317</td>
<td>0.174</td>
<td>0.095</td>
<td>0.052</td>
<td>0.029</td>
<td>0.016</td>
<td>0.009</td>
<td>0.005</td>
</tr>
<tr>
<td>0.07</td>
<td>0.523</td>
<td>0.260</td>
<td>0.129</td>
<td>0.064</td>
<td>0.032</td>
<td>0.016</td>
<td>0.008</td>
<td>0.004</td>
<td>0.002</td>
</tr>
<tr>
<td>0.08</td>
<td>0.473</td>
<td>0.213</td>
<td>0.095</td>
<td>0.043</td>
<td>0.019</td>
<td>0.009</td>
<td>0.004</td>
<td>0.002</td>
<td>0.001</td>
</tr>
<tr>
<td>0.09</td>
<td>0.428</td>
<td>0.174</td>
<td>0.071</td>
<td>0.029</td>
<td>0.012</td>
<td>0.005</td>
<td>0.002</td>
<td>0.001</td>
<td>0.000</td>
</tr>
<tr>
<td>0.10</td>
<td>0.387</td>
<td>0.142</td>
<td>0.052</td>
<td>0.019</td>
<td>0.007</td>
<td>0.003</td>
<td>0.001</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
4.3 Robustness of the results

We assumed path lengths counted as the number of nodes on the basis path to be truncated Poisson distributed in Section 4.2. How sensitive would be the Type II error probability $\beta$ be if path lengths had some other distribution? To check this we assume the distribution to be geometric with parameter $\gamma$ and re-derive below the expression equivalent to equation (5).

If path lengths were distributed geometrically with parameter $g$, then

$$P_n = P[\text{path length} = n] = \gamma (1 - \gamma)^{n-1}, \quad n = 1, 2, 3, \ldots$$

Therefore,

$$\sum_{n=1}^{\infty} P_n (1 - \epsilon)^n = \sum_{n=1}^{\infty} \gamma (1 - \gamma)^{n-1} (1 - \epsilon)^n$$

$$= \frac{\gamma}{1 - \gamma} \sum_{n=1}^{\infty} [(1 - \gamma)(1 - \epsilon)]^n$$

$$= \frac{\gamma (1 - \epsilon)}{\gamma + \epsilon - \gamma \epsilon}$$

(6)

It may be verified that if the SW is defect-free, that is if code quality is perfect, indicated by $\epsilon = 0$, then $P[\text{the SW passes the single test tracing a random basis path}] = 1$.

Figure 5 displays the sensitivity of the probability of committing a Type II error (i.e., releasing a faulty SW) based on testing a single basis path to code quality $\epsilon$ and geometric distribution parameter $\gamma$ (note that average path length would now be $1/\gamma$). The noticeable similarity between Figures 4 and 5 suggest that the probability of committing a Type II error or releasing a faulty SW based on testing a single basis path is relatively robust with respect to path length distribution probabilities.

Figure 5 Sensitivity of probability of Type II error to $\epsilon$ and $\gamma$
5 Imperfect inspection

5.1 Testing may be defective!

Software quality assurance (SQA) remains for most part still at the very early stages of quality assurance when one observes where hardware quality assurance or process control methods are today. Six Sigma approach and methods are now well-set benchmarks in QM practices in the world of manufacturers and product developers while SQA leans primarily on inspection and testing as the sole bases for appraising SW. SW failures are no less serious now even in air or space travel than a defective O-ring or protective tile (Whittaker, 2000). It would behove one to set up methods and techniques to move the development of SW close to how hard artifacts are designed, prototyped, field tested and launched in the marketplace today.

A critical step in SW development continues to be ‘testing’, generally executed through pre-designed test protocols and ‘cases’. Since more often than otherwise the SW to be appraised is far too big to be exhaustively tested, the test cases normally test only a portion of the total bulk – all of the interlinks, logic, paths, variables, etc. of the SW, before one decides to release the SW to the user.

Besides and importantly, not only writing a defect-free SW is a challenge, even the test cases themselves may be defective or imperfect, leading to investigators coining the term ‘quality of the inspection process’ (see citations 2, 4, 6, 7, 10 14 and 15 in Wu et al., 2005). The evolving approach invokes Bayesian Networks (BNs), but it still produces mostly qualitative inferences, being based on subjective inputs regarding complex-to-express and quantify joint and/or conditional probabilities (Jensen, 2001). By contrast, the approach adopted in this paper is quantitative and analytic.

5.2 A model for defective testing

Defective testing in the hardware world is checked by gage repeatability and reproducibility studies (Gryna et al., 2007). A key aspect is calibration, the other being installing the required precision level of the measurement instruments – used where the quality characteristic can be measured. When such measurement is not possible but the artefact has a defect and can only be judged to be OK (acceptable) or defective (unacceptable), one resort to appraisal by attributes (Gryna et al., 2007).

Clearly, quality assurance by go/no-go appraisal does not reveal much about the quality of the ‘process’ producing those artifacts. Still, this level of inspection remains a part of practice when the quantity to be appraised is large (Gryna et al., 2007). SW appraisal at the unit test or at the ‘node’ level falls in this category, the ‘instruments’ being the test cases and the testing protocol (Pressman, 2005). Since this appraisal process is not prefect (SW error references), to study it we set up a simple Bernoulli-type (Rice, 2007) model for imperfect inspection using ‘misclassification’ probabilities $\theta$ and $\phi$ as follows:

$$\theta = \text{Probability[misclassify a good node as defective]}$$

Hence, the probability of correctly classifying a good node as good $= 1 - \theta$. Similarly, we define
\[ \phi = \text{Probability}\left[\text{misclassify a defective node as good}\right] \]

Hence, the probability of correctly appraising a defective node as defective = \(1 - \phi\).

These definitions are shown in Figure 6.

**Figure 6** Misclassification probabilities

---

**5.3 Risks in appraising SW based on limited and/or imperfect testing**

A Type I error (rejecting a perfectly good SW) would be committed even if one good node in the defect-free SW is misclassified as defective. The corresponding probability is \(\alpha\). As defined in Section 3.1, the average total number of nodes in the SW is \(\eta \cdot CCN\). If code quality (the probability that a node is defective) is \(\varepsilon\), then

\[
P[\text{there exist at least 1 defective node in the SW}]
= \sum_{r=1}^{\eta \cdot CCN} P[r \text{ defective nodes in SW}]
= 1 - P[r = 0] = 1 - \varepsilon^0 (1 - \varepsilon)^{\eta \cdot CCN}
\]

There are two ways in which a defect-free SW may be rejected. The first is due to imperfect inspection. This is the probability that at least one good node is misclassified as bad when only \(T\) nodes out of \(CCN\) basis paths have been tested (imperfectly). Thus

\[ \alpha_1 = P[\text{rejecting a defect-free SW when it has been tested up to } T \text{ basis paths and at least one good node has been misclassified}] \]
It is easy to check that if \( \theta = 0 \) (i.e., no good nodes are misclassified as defective), \( \alpha_1 = 0 \). Also, if the SW has no defective nodes in it, i.e., if \( \varepsilon = 0 \) and \( 0 \leq \theta \leq 1 \), then \( \alpha_1 = 1 - (1 - \theta)\eta^T \).

There is yet another (second) way by which a Type I error may be committed when only \( T \) basis paths out of the total \( CCN \) basis paths have been tested and found defect-free, still the SW is rejected on speculation about the \( (CCN - T) \) remaining paths. The probability of this is \( \alpha_2 \). Since there is no logical basis for rejecting such a SW at this ‘test incomplete’ stage, we assume that the testing protocol in effect will prevent the rejection of a SW when only part of it has been tested and found OK. Such stoppage may arise due to testing budget limitations or when perhaps the SW is just too large for further testing. However, one may either resume testing at a later time or just release the SW to the user and not reject the SW.

A Type II error (accepting or releasing a SW with at least one defective node in it) may also be committed as the result of imperfect testing. A defective node may be misclassified as good. The probability of committing a Type II error is \( \beta \), which may occur even if a single defective node survives in the SW being released. In general, \( \beta \) will arise out of two independent events. The first is when even one defective node is misclassified as good and escapes into the SW released. The second is when testing is non-exhaustive and only \( T \) of the total \( CCN \) basis paths in the SW have been tested and defects might remain in the nodes on the untested \( (CCN - T) \) basis paths, yet the SW is released.

\( \beta_1 \) may be evaluated as follows. If code quality (the probability that a node is defective) is \( \varepsilon \) and if \( T \) basis paths have been tested so far, then

\[
\beta_1 = P[\text{erroneously passing } \eta T \text{ defective nodes as 'good']} \\
= (\varepsilon \phi)\eta^T
\]

Note that if \( T = 0 \), then no testing has been done yet. Therefore, the decision to release involves risks that cannot be quantified. On the other hand, if the SW is defect-free, then \( \varepsilon = 0 \) and consequently \( \beta_1 \) is zero, indicating that there is no risk in releasing a perfect SW without any testing.

The other manner in which a Type II error may be committed is when one or more defects may remain in the untested (i.e., \( \eta (CCN - T) \) nodes) portion of the SW. The probability of this event is found as

\[
\beta_2 = P[\text{one or more defective nodes remain in the untested part of the SW, i.e., in } \eta (CCN - T) \text{ nodes}] \\
= 1 - (1 - \varepsilon)^\eta (CCN - T)
\]

Hence the probability of committing a Type II error with imperfect testing is
The special situations may be quickly checked. If no testing has been done on a SW with code quality $\varepsilon$, average basis path length $\eta$ and $CCN$, then $T = 0$, hence

$$\beta = \beta_1 + \beta_2 = (\varepsilon \phi)^{CT} + 1 - (1 - \varepsilon)^{CCN - T}$$

(8)

If all $CCN$ basis paths have been exhaustively tested, but test effectiveness or the probability of misclassifying a defective node as OK is $\phi$, then

$$\beta = (\varepsilon \phi)^{CCN}$$

And if code quality is perfect, i.e., $\varepsilon = 0$, then $\beta = 0$.

Releasing SW with untested portions of codes in it or with ‘imperfect testing methods’ is a most serious matter today. A single SW disaster caused the loss of the European Ariane 5 launcher – a direct loss of $500$ million (Whittaker, 2000). Others have shut down airports for months or brought down airplanes killing scores. However, except in special cases in today’s world it is no more possible to test SW exhaustively.

The sensitivities of $P[Type I]$ to code quality $\varepsilon$ and misclassification calling good nodes defective and $P[Type II]$ to $\varepsilon$ and misclassification probability $\phi$ are shown in Figures 7 and 8.

Figure 7 $P[Type I Error]$ as function of code quality and misclassification calling good nodes defective (see online version for colours)

Notice that the causal effects are not additive in either case. Such precise results do not yet appear in SW engineering literature that adopt the BN basis to display dependencies and relationships among inspection effectiveness, product complexity, inspection process quality, product size, etc. The present approach is analytic and more general and the results are consistent with those projected in Wu et al. (2005) which used BN.
6 Type I error risk in a single randomly run test with imperfect inspection

This section presents the derivation of expressions for Type I error probability \( \alpha \) when inspection is not perfect and a good node may be misclassified as defective or a defective node is passed as acceptable (Figure 6). Thus, this is a generalisation of Section 4.

A path with \( n \) nodes will be free of defects with the probability \((1 - \varepsilon)^n\). If any one of these nodes is defective but the test is ‘passed’, a Type I error is committed. If inspection is perfect, it will always catch the defective nodes and declare the SW to be faulty. Thus the probability of declaring SW to be faulty given that some defective nodes are present in it, when inspection is perfect

\[
P[\text{Type I error committed in testing}] = 0
\]

However, if testing is not perfect, a good node may be declared to be faulty with probability \( \theta \), which reflects testing quality. Now if \( n \) nodes with code quality \( \varepsilon \) have been tested and testing quality is \( \theta \), then

\[
P[\text{Type I error}] = P[\text{at least one truly good node among } n \text{ nodes tested will be misclassified as defective}]
\]

\[
= \sum_{i=1}^{n} \binom{n}{i} (1 - \varepsilon)^i \theta^i
\]

\[
= [1 + (1 - \varepsilon) \theta]^n - 1
\]

It may be verified that when \( \theta = 0 \), \( P[\text{Type I error}] = 0 \) and when \( \varepsilon = 0 \) and \( \theta \geq 0 \), then \( P[\text{Type I error}] = (1 + \theta)^n - 1 \), implying that even a defect-free SW may be erroneously rejected when testing is imperfect.
Now, \( P[\text{SW is rejected in a single imperfect test tracing a random basis path through nodes}] \)

\[
\sum_{n=1}^{\infty} P[\text{path length} = n] \times P[\text{SW is rejected due to imperfect inspection} \\
/ \text{path length} = n] \\
= \sum_{n=1}^{\infty} P_n [(1+(1-\varepsilon)\theta)n-1]
\]

Here \( P_n \) is the probability distribution of path lengths.

When the basis path lengths are truncated Poisson distributed with parameter \( \lambda \) and when testing is imperfect with probability \( \theta \), it may be shown that

\[
P[\text{Type I error}] = [e^{\lambda(1+(1-\varepsilon)\theta)}-1]\frac{e^{-\lambda}}{1-e^{-\lambda}} - 1
\]  
(9)

It may be quickly checked that if \( \theta = 0 \), then \( P[\text{Type I error}] = 0 \), a result that is obvious.

When basis path lengths are geometrically distributed with parameter \( \gamma \), then

\[
P_n = \gamma(1-\gamma)^{n-1}, \ n = 1, 2, 3, ...
\]

and \( E[\text{path length}] = 1/\gamma \). In this case, the probability of committing a Type I error (i.e., erroneously rejecting a defect-free SW) may be evaluated as

\[
\sum_{n=1}^{\infty} P_n = \gamma(1-\gamma)^{\gamma-1} P[\text{SW rejected / path length} = n] \\
= \gamma \left[ \frac{1}{1-\gamma} \sum_{n=1}^{\infty} [(1-\gamma)(1+(1-\varepsilon)\theta)^n - \sum_{n=1}^{\infty} (1-\gamma)^n] \right]
\]

Therefore,

\[
P[\text{Type I error}] = \frac{\gamma(1+(1-\varepsilon)\theta)^n}{1-(1-\gamma)(1+(1-\varepsilon)\theta)} - 1
\]  
(10)

Observe the rapid rise in \( \alpha \), the probability of rejecting a good SW when inspection quality deteriorates and \( \theta \) becomes larger and larger – with both Poisson and geometric path length distribution cases. It would therefore behove us to perfect our instruments (the test cases used and the test protocol put into effect) whenever we are testing codes. Recall that Type I error is the probability of erroneously rejecting a SW that is defect-free. In the present case, this error is being amplified since the tester may judge a good node to be defective. Figure 9 displays the sensitivity of the probability of committing a Type I error to code quality \( \varepsilon \) and the imperfectness in inspection \( \theta \) when basis path lengths are Poisson-distributed. Figure 10 displays this picture when path lengths are geometrically distributed.
7 Type II error risk in a single randomly run test with imperfect inspection

Type II error is committed when one accepts (‘releases’ in the context of SW development) a SW with one or more defective nodes in it. In this section, we derive the probability of committing a Type II error in testing a single randomly chosen basis path. As before, here also we assume the definitions of imperfect inspection parameters given in Section 5.2.

The probability ($\beta$) of committing a Type II error, which occurs when even one defective node on the basis path being tested is declared ‘good’, is evaluated as follows.
Assume that basis path length is represented by \( N \). Then, assuming \( \phi \) to be the probability of declaring a defective node as error-free and code quality to be \( \varepsilon \),

\[
P[\text{Type II error in testing } N \text{ nodes}] = \\
\sum_{n=1}^{N} P[\text{at least 1 misclassification is committed / } n \text{ defective nodes present in } N] \\
\times P[n \text{ defectives present in } N \text{ nodes}] \\
\sum_{n=1}^{N} (1 - (1 - \phi)^n) \binom{N}{n} \varepsilon^n (1 - \varepsilon)^{N-n}
\]

One may quickly check that if \( \varepsilon = 0 \) (code quality is error-free) or if \( \phi = 0 \) (there are no misclassifications in testing defective nodes as good), the probability of committing a Type II error in testing \( N \) nodes that comprise the basis path is 0. Now we are ready to derive \( \beta \), the probability of committing a Type II error when a basis path is randomly tested.

\[
P[\text{Type II error in testing a single basis path}] \\
= \sum P[\text{Type II error / } N \text{ nodes tested}] \times P[\text{ } N \text{ nodes present in path}] \\
= \sum P_N \times P[\text{Type II error / } N \text{ nodes tested}]
\]

Path lengths (\( N \)) may have any discrete distribution – truncated Poisson, geometric, etc. – providing the expression for \( P_N \).

Let \( N \) be truncated-Poisson distributed as assumed in Section 4.2. Therefore using equation (4) we have

\[
= \frac{1}{1-P_0} \sum_{N=1}^{\infty} \frac{\lambda}{N!} e^{-\lambda} \left[1 - (1 - \varepsilon)^N\right] \\
= \frac{1}{1-P_0} \sum_{N=1}^{\infty} \frac{\lambda}{N!} e^{-\lambda} \sum_{n=1}^{N} \binom{N}{n} \varepsilon^n (1 - \varepsilon)^{N-n}
\]

Here \( P_0 = e^{-\lambda} \). On simplification and the use of well-established identities we get

\[
P[\text{Type II error in testing a single basis path}] \\
= \beta = 1 - \frac{1 - e^{-\lambda(1 - \phi)}}{e^{\lambda \phi}} (1 - e^{-\lambda})
\]

It may be verified that when the SW is defect-free, i.e., when \( \varepsilon = 0 \), the above probability \((\beta) = 0\). Also when inspection is perfect, i.e., when no defective nodes are misclassified as good or \( \phi = 0 \), then also \( \beta = 0 \).

If path lengths are geometrically distributed with parameter \( \gamma \) and

\[
P_N = \gamma (1 - \gamma)^{N-1}, \; N = 1, 2, 3, \ldots
\]

then it may be shown that
\[ \beta = 1 - \frac{(1 - \phi)\gamma}{\gamma + \phi - \gamma\phi\varepsilon} \]  

Figures 11 and 12 show the sensitivity of committing a Type II error to code quality \( \varepsilon \) and the probability of misclassifying a defective node as good (\( \phi \)) due to imperfect inspection. Clearly, the best situation is when the code is truly error-free and inspection is perfect. Note two things here. The sensitivities are relatively negligible when the path length probabilities change from truncated Poisson to geometric.

**Figure 11**  \( P[\text{Type II error}] \beta \) as function of code quality and misclassification calling good nodes defective; path lengths truncated Poisson distributed

**Figure 12**  \( P[\text{Type II error}] \beta \) as function of code quality and misclassification calling good nodes defective; path lengths geometrically distributed
Further, note that the impact of testing not being perfect is quite serious. The rise in $\beta$ is quite rapid when $\phi$ rises from 0 to 0.045 (almost considered ‘negligible’ by many). Note also the interaction present between code quality and level of imperfectness in inspection.

8 Discussions

Testing a modern complex SW system is hard (Whittaker, 2000). This study has explored the impact of code quality or node level defect rate ($\varepsilon$), $CCN$ and the average basis path length ($\eta$) on $\beta$, the probability of releasing a defective SW based on inexhaustible test runs and imperfect testing. Additionally, this study has extended analytical treatment of the likelihood of needlessly debugging a defect-free SW due to non-exhaustive and/or imperfect testing of the code. Thus, results provided through equations (5) to (12) materialise the quantitative assessment of such risks and one’s capability for enhancing SW development processes and designs. Such precise results do not exist in the literature. What little exists is based on subjective opinions or mere observations of cases [see Jensen (2001) and citations 2, 4, 6, 7, 10 and 14 in Wu et al. (2005)]. Such approaches are yet to precisely address the seriousness of potential SW disasters.

To this end simplifying but reasonable assumptions are made here to keep the analysis tractable as was done by queuing theorists in the last century. Exact mathematical models are built here to study the sensitivity of the probability of passing SW as defect-free at the outcome of limited and/or imperfect testing of codes. The present approach is precise and as it is illustrated here, it is capable of uncovering the sensitivity of the risk of releasing a defective SW or needlessly debugging a perfect SW. The examples are shown in Figures 4, 5 and 7 through 12. Other specific situations may be similarly studied – theoretically or numerically.

For analytical tractability, this study assumes all nodes to be equally likely to be defective ($\varepsilon$) whereas it assumes path lengths to be distributed as truncated Poisson. Subsequently the results are re-derived assuming path lengths to be geometrically distributed. These assumptions are not mandated, but they lead one to quickly derive illustrative relationships that help evaluate the sensitivity of $\alpha$ and $\beta$ – the likelihood of needless delays or releasing a faulty SW based on imperfect and/or limited testing. Other situations might be similarly modelled, albeit the results might have to be evaluated numerically rather than being abstracted into neat expressions such as equations (5) to (12). In the cases evaluated, the probability of releasing a defective SW based on testing a single basis path appears to be relatively robust to the path length distribution chosen (see Figures 5 and 6 for instance).

Committing a Type II error is quite serious as it may lead to the release of a defective SW in the marketplace. A defective SW may lead to rework and returns, labourious maintenance and it may even harm the client’s operations financially, cause unsafe actions or delay operationalising critical functions. Therefore, both SW design as well as testing should minimise the probability of releasing a SW with unacceptable performance. This study finds that when not all $CCN$ paths are tested, the likelihood of releasing a faulty SW is directly dependent on node level code quality $\varepsilon$ and average basis path length $\eta$, but inversely dependent on the fraction of all basis paths tested. These have significant implications for the SW architect, as well as for the tester. As expected, only when all basis paths have been successfully tested, $\beta$ becomes zero.
In another issue examined, we addressed the probability of committing a Type II error ($\beta$) in one random pass through a SW from input to output. It is found that the probability of a single node being defective ($\epsilon$) has relatively small impact on $\beta$ (the Type II error probability), but path length $\eta$ has a significantly higher impact on $\beta$. The longer is the path from the start (the first node) to the end (the last node), the lower is the probability of committing a Type II error. In other words, it is less likely a defective SW with many long paths through its maze of processing nodes will be declared acceptable. On the other hand, when rushed, patches or smaller SW with faults often slip by without detection. These conclusions parallel the results displayed in Table 9 of Basili and Perricone (1984).

9 Conclusions

We find strong support in this study for a SW design strategy that splits up the total task that a SW is expected to accomplish into a large number of small modules, each of which is coded carefully to complete a distinct task and thoroughly tested, somewhat similar to the SOA philosophy and that of the .net technology (DOT NET Technology). Both these approaches use a large number of pre-coded solutions that are subsequently called upon or assembled into an application program of interest (Bell, 2008). The final SW here would be a compilation of these smaller soundly developed subunits. The assembly is then tested through integration and user acceptance and later subjected to regression tests as needed. This work suggests that such a strategy would result in the minimum risk of releasing a SW with inadvertently left defects in it. In this light XP (code-test-code-test) also appears to be a sensible strategy to follow.

This study has explicitly modelled the effect of Bernoulli-type faulty testing on both $a$ (the needless debugging and the associated delays caused to a perfect SW) and $b$ (the release of a defective SW). The presently derived relationships are exact and analytic under the stated assumptions. This sharply contrasts with reported results based on the BN modelling approach.

So the three specific conclusions of this study are

1. Defective testing greatly raises the likelihood of releasing a defective SW or delaying a SW needlessly, often to unacceptable levels. Hence, minimise errors and mistakes in test case design and during testing.
2. Your protection rises proportional to the rise in coverage. Hence maximise the coverage in SW testing.
3. Build SW in small components and where possible using well-crafted reusable parts that are tested well (~ 100% coverage) before those are put in the library and then assemble them into the application product.

A future study will harness the power of the sequential probability ratio test (SPRT) in code verification, when subject to imperfect testing. This approach has the potential of reducing the total number of tests required to declare a large SW to be fault-free or defective (Bagchi, 1992; Gryna et al., 2007).
Acknowledgements

This study benefited greatly from the author’s discussions with Rupak Majumdar and Sujayendra Chakravarti. The author appreciates their generously offered reflections on SW quality.

References


DOT NET Technology available at http://en.wikipedia.org/wiki/Microsoft.NET#Microsoft.NET.


Risks in software development with imperfect testing