RESID: A Practical Stochastic Model for Software Reliability

Arnab Chakraborty

Applied Statistics Unit

Indian Statistical Institute

203, B T Road

Kolkata, India 700108

Phone: 91-33-2334-0337

arnabc@isical.ac.in

Abstract

A new approach called RESID is proposed in this paper for estimating reliability of a software allowing for imperfect debugging. Unlike earlier approaches based on counting number of bugs or modelling inter-failure time gaps, RESID focuses on the probability of "bugginess" of different parts of a program buggy. This perspective allows an easy way to incorporate the structure of the software under test, as well as imperfect debugging. One main design objective behind RESID is ease of implementation in practical scenarios.

Keywords: Software reliability, Debugging, Statistical modelling.

1 Introduction

With computer programs pervading all walks of modern life, software debugging has long been an area of active interest. There are three major aspects to this problem. Firstly, one needs better software development tools to avoid creating bugs. Secondly, one needs to be able to detect and correct bugs that have already crept in. The third goal is to estimate the reliability of a software program. Since no fool proof debugging method is known to exist, the third goal is no less important than the first two. Various approaches have been suggested in the literature to estimate the reliability of a given piece of software, ranging from simple profiling techniques (disregarding the stochastic nature of bugs) to elaborate stochastic models (that often overlooks the structure of the program). In this paper we propose a new technique called **Reliability Estimation for Software under Imperfect Debugging (RESID)** to make the twain meet: a statistical method based on maximum likelihood estimation that also takes the structure of the program into account. The model also allows the possibility of imperfect debugging, where a particular block of code is allowed to contain bugs (albeit with a reduced probability) even after multiple debugging sessions.

The paper is laid out as follows. In the next section we review various techniques proposed for the problem, with a brief discussion of their merits and demerits. Section 3 presents the new method from the theoretical viewpoint. Suggestions for practical implementation of the method are given in section 4. Section 5 presents some discussion about the performance of the technique based on simulation. Section 6 deals with some variations of **RESID** to suit specific needs. After a brief concluding section some probabilistic underpinnings of the method are outlined in the very last section.

2 Review of existing techniques

Many models and approaches have been suggested in the literature to assess software reliability. We shall briefly a review a selection of these techniques, without aspiring for comprehensiveness, which anyway is beyond the scope of this short paper. An extensive literature review is given in [11]. Software reliability is typically defined as the probability of failure-free operation of a computer programme in a specified environment for a specified period of time [7]. As [2] points out the statistical models for software reliability come in two distinct flavours, those that deal with time between successive failures, and those that deal with counting bugs in a program.

The former approach, pioneered by (Jelinski & Moranda 1972), (Moranda 1975) makes the (somewhat unrealistic) assumption that inter-failure times are exponential in nature, and are independent of one another. Methods of this genre also make the assumption that a bug is always rectified when it is detected. This unfortunately leaves no place for wrong or incomplete fixes, a phenomenon that ubiquitously plagues the software industry. A related approach is taken by (Singpurwalla & Soyer 1985), (Singpurwalla & Wilson 1999), where the authors try to fit time series models to the inter-failure times.

The second approach uses point processes to model occurrences of bugs (Goel & Okumoto 1979). The paper (Nayak 1988) is a typical example, where the author employs Poisson processes for this purpose. One notable feature of this approach is that the same bug is allowed to recur. Indeed, the author suggests that after detection a bug should either not be removed or at least have its place marked, so that every subsequent pass through that position may be counted. This idea has been extended in (Dewanji, Nayak & Sen 1995) where the presence of multiple bugs is modelled as multiple Poisson processes running independently in parallel.

The main interest there lies in estimating the number of processes.

However, as noted in (Xie 2000), the plethora of proposed, pedantic models contrasts sadly with the paucity of practically implementable ones. Due to resource constraints, debuggers and programmers often have to resort to *ad hoc* plans to yield quick results, often in response to the needs of some irritated customer demanding a fix for a particular bug. Most of the methods proposed in the literature are a bit too elaborate to cope with such real life scenarios. Some practical method based on a reasonable statistical model would be a useful addition to a software engineer's repertoire. In this paper we seek to propose such a method.

3 RESID

One of the trickiest aspect of quantifying software reliability is to come up with a quantitative definition of a bug. Often a single mistake in a software triggers multiple branches of a system to fail. Techniques based on only the occurrences of failures often count each failed branch as a separate bug, while from the viewpoint of software management these should be considered as a single bug. We circumvent this inherent ambiguity in the definition of a bug by looking instead at the concept of "bugginess" as follows.

We can consider a piece of code as a flowchart with branches and loops. Control may flow down different branches depending on the initial data. However, every program consists of some *blocks* which is defined as a sequence of consecutive instructions without any embedded branching in it. Thus, once control enters a block it must either flow through the block along a unique linear path, or must crash the program (possibly because of a bug in some earlier block.

We shall assume that each block has some probability p of being "buggy". More specifically, p is the chance that we encounter some bug in

that block while running the software with a random data. Thus, strictly speaking randomness enters not only due to the inadvertent errors of the programmers, but also due to the choice of the input data. We shall also assume that the event that one block contains a bug is independent of another block containing a bug. This is not as impractical as it may sound, as bugs are born of inadvertent mistakes on part of the programmers, and not as a result of the control structure relating the blocks. Our assumption of independence does not preclude the possibility that a bug in one block may wreak havoc in a subsequent block. Mathematically inclined readers not content with this explanation may see the last section for a more rigorous presentation of this idea.

To assess the reliability of a piece of software we start with the structure of the program in terms of the blocks. During the debugging phase the program is run multiple times, each time with independent initial data. For each run we record the following information.

1. whether the run has terminated correctly or not,
2. if the run indicates the presence of a bug, then which block contains the bug,
3. which blocks have been executed and how many times.

The first two pieces of information are available from any standard debugging session. In order to collect the frequency of execution of the blocks one has to embed a logging command in each block. Such an embedding can be easily achieved automatically using software tools. We assume that a bug is fixed (possibly imperfectly) once it is detected. We shall later point out two variants of the same approach to cope with situations where a bug cannot be removed, or where multiple bugs are identified in a single run.

Notice that even though we feed independent initial values in the program each time, the collected data are not iid in nature, since the under-

lying software changes with each debugging.

We stochastically model the software debugging mechanism as follows. Initially each block is believed to be buggy with probability p.

Every time a block is "debugged" we assume that the probability of its still remaining buggy is scaled down by a known factor $\alpha \in (0, 1)$, which measures the debugging inefficiency. Thus, after detection and correction of its first k bugs, a block has probability $p\alpha^k$ of remaining buggy. A value of α close to 0 implies efficient debugging, while a value close to 1 implies the opposite.

We shall consider p as a measure of unreliability (or, rather, the lack thereof) of the over all software. Each block gets its own unreliability score $p\alpha^k$. It is not difficult to come up with a color coding scheme to depict the unreliability scores of the different blocks diagrammatically, *e.g.*, using a UML diagram.

If p is sufficiently small we might consider the software as reliable enough. If, however, p is large, then we may like to focus our debugging efforts on the blocks with higher unreliability scores.

We shall employ maximum likelihood estimation to estimate p. Writing down the likelihood function, however, is a bit tricky here, as we have to take the structure of the software into account. The procedure is best explained with a simple example.

Consider a simple program to check for divisibility by 7:

```
main() {
  int n;

  printf("Please supply an integer: ");
  scanf("%d",&n);
  if((n % 7) == 0) {
    printf("The number is divisible by 7.\n");
```

```
  else
    printf("The number is not divisible by 7.\n");
}
```

with control flow as shown in Fig 1. It has 3 blocks each labelled with an arrow and number. The control starts in block 1, then comes to an `if`-class, and branches out into block 2 or 3.

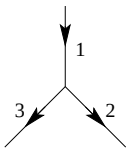

Fig 1: A 3-block program with an `if`-clause

The program is executed 5 times with the following results:

1. 1; bug in 1 (*i.e.*, the program crashed in block 1 due to a bug in that block)
2. 1,2; bug in 2
3. 1,3; no bug
4. 1, 2; bug in 1 (*i.e.*, the program continued up to block 2, but a bug was found in block 1)
5. 1,3; bug in 3

We assume that the buggy block is (imperfectly) debugged after each unsuccessful run. Also, the blocks visited after passing through a buggy block produce unreliable results, and so are to be ignored. For example, we shall truncate the record for the fourth run above to

1; bug in 1.

The probability of bugginess of each block is listed below, along with the likelihood values for each run:

Stage	Block 1	Block 2	Block 3	Likelihood
0	p	p	p	
1	$p\alpha$	p	p	$P(1;\ \text{bug}) = p$
2	$p\alpha$	$p\alpha$	p	$P(1,2;\ \text{bug}) = (1-p\alpha)p$
3	$p\alpha$	$p\alpha$	p	$P(1,3;\ \text{no bug}) = (1-p\alpha)(1-p)$
4	$p\alpha^2$	$p\alpha$	p	$P(1;\ \text{bug}) = p\alpha$
5	$p\alpha^2$	$p\alpha$	$p\alpha$	$P(1,3;\ \text{bug}) = (1-p\alpha^2)p$

Taking the product of the last column we get the likelihood of the entire data set as

$$L(p) = \text{constant} \times p^4(1-p)(1-p\alpha)^2(1-p\alpha^2).$$

It is not hard to see that this scheme can be generalized to any branching structure. However, the situation becomes somewhat different in presence of loops. A bug inside a loop may not be triggered during the very first pass through the loop. In practice it is often difficult to keep track of the exact pass when a bug inside a loop is triggered for the first time. So if the program halts due to a buggy block inside a loop, all that we can be sure of is that the blocks leading to it have worked correctly at least once. We have no way of knowing if the bug has already been triggered before subsequent passes through the loop or not. We explain this idea with an example.

Consider the loop structure shown in Fig 2.

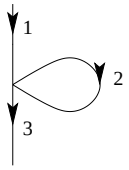

Fig 2: A 3-block prgroam with a loop

Suppose that a run through this program produces the following record:

1,2,2,2,3; bug in 2.

As should be obvious to anybody with even moderate debugging experience, it is hard to detect which pass(es) through block 2 has (have) triggered the bug. As a result all that we can be sure of is that block 1 has worked fine in this run, with at least one pass of block 2 failing. Once a bug is triggered the subsequent blocks cannot be reliably debugged. So we shall truncate the record to

1,2; bug in 2.

As can be seen easily the likelihood under this model is of the form

$$L(p) \propto p^m \prod_{i=0}^{k} (1 - p\alpha^i)^{n_i}.$$

Here k is the maximum number of debugging session for any block, m is the number of bugs detected and removed, and n_i (for $i = 0, 1, ...$) is the number of perfect runs for blocks with exactly i debugging attempts.

The log-likelihood is (up to an additive constant)

$$\ell(p) = m \log p + \sum_{i=0}^{k} n_i \log(1 - p\alpha^i).$$

The following fact is useful for numerically maximising this function for $p \in (0, 1)$.

Lemma: For any program and any debugging outcomes, the model has a strictly concave log-likelihood function. ///

Proof: The functions $\log p$ and $\log(1 - \frac{p}{a})$ are strictly concave for any $a > 0$, and a linear combination of concave functions with positive coefficients is again strictly concave. ///

This fact implies the uniqueness of MLE $\hat{p}$, if it exists. Unfortunately, MLE may not always exist. But it does exist under the following fairly mild condition.

Lemma: If $m, n_0 > 0$ then $\ell(p)$ has unique maximum over $(0,1)$.///

Proof: This is because

$$\begin{aligned} \lim_{p\to 0+} \frac{1}{p} &= \infty \\ \lim_{p\to 0+} \frac{1}{1-p\alpha^i} &\in \mathbb{R} \text{ for } i = 0, 1, ... \end{aligned}$$

Also

$$\begin{aligned} \lim_{p\to 1-} \frac{1}{p} &\in \mathbb{R} \\ \lim_{p\to 1-} \frac{1}{1-p} &> \infty \\ \lim_{p\to 1-} \frac{1}{1-p\alpha} &\in \mathbb{R} \text{ for } i = 1, 2, ... \end{aligned}$$

So if $m, n_0 > 0$, we have

$$\ell'(0+) > 0 \text{ and } \ell'(1-) < 0.$$

Strict concavity from the last lemma now clinches the argument. ///

The condition of this lemma has a simple interpretation: $m > 0$ means at least one bug is encountered somewhere during some run of the program. The condition $n_0 > 0$ means at lest one block has worked properly in the very first attempt. It is easy to see that the probability of both these happening goes to 1 as the number of blocks go to infinity.

Incidentally, it may be shown without much additional effort that the condition $m > 0$ is necessary for the existence of MLE. However, the condition $n_0 > 0$ is only sufficient. In fact, this is one of a general class of sufficient conditions of the form $n_i > \alpha^{-i} - 1$.

One may now easily apply numerical methods like Newton-Raphson to solve

$$\ell'(p) = 0,$$

or

$$\frac{m}{p} - \sum_{i=0}^{k} \frac{n_i \alpha^i}{1 - p\alpha^i} = 0.$$

However, here we can avoid computing the second derivative needed for Newton-Raphson iteration by using bisection method. By virtue of the last lemma we can perform bisection method over the interval $[\epsilon, 1-\epsilon]$ for some suitably small $\epsilon > 0$.

4 Implementation

The main aim of this paper is to propose a systematic debugging technique that can be easily integrated with existing methods. Our exposition so far has been primarily theoretical. This section outlines how our approach fits into a typical debugging session, where input data may come from a customized design or user feedback or simply generated randomly.

The first step is to identify the blocks in the software. This can be achieved easily by simple lexical analyzers and parsers (like those generated by **flex** and **bison**(Appel 2004)). Such preprocessing steps are common in many program validating scenarios. In our case the preprocessing step creates a data base of the blocks, associating each block with its file name and the first and last line numbers. It also embeds a data logging command at the start of the each block, such that the block number is recorded in a log file the moment control enters that block.

After this simple preprocessing is over the actual debugging starts, which consists of repeated runs each time with fresh initial data. After each run the programmer checks if the output is OK or not. If it is, then this fact is recorded. If something has gone wrong then the programmer debugs the program, and identifies the line(s) that required correction. Thus, for each run we get the logged record of visited blocks, as well as the location of the bug, if any.

The resulting data set is now ready to be analyzed with our approach.

First, we identify the block containing the buggy line(s). We shall discuss later the scenario where multiple blocks are corrected in a single run. Next we identify the first occurrence of this block in the logged record, and discard everything after this occurrence. This is necessary because once a buggy block is visited, all subsequent steps are unreliable.

Next, we extract the statistics m, k and n_i's from the accumulated records, and use these to compute and maximize $\ell(p)$ for $p \in (0, 1)$.

It should be noted that all the steps can be easily automated, and hardly cause any disruption to the usual work flow of the programmer in charge of debugging. This seamless integration with the existing habits of programmers is a major strong point of **RESID**.

5 Results

The proper evaluation of **RESID** may only be done in an industrial set up where a large, complex software is actually being debugged. In this paper we present the results using a simulated toy example. We start with a C program, and simulate a bug in each block with probability $p\alpha^r$ where p is some chosen value of the parameter of interest, $\alpha = 0.9$ (chosen arbitrarily) measures debugging inefficiency, and r is the number of times this block has already been debugged.

The first example is a simple program consisting of just 4 blocks as shown in the flowchart in Fig 3. Each rhombus represents an `if`-clause, and the rounded rectangle represents a loop that extends up to the circular connector. In the simulated runs we take each branch in an `if`-clause with equal probability. Also the loop is run for a random number of steps generated uniformly from $\{1, ..., 100\}$.

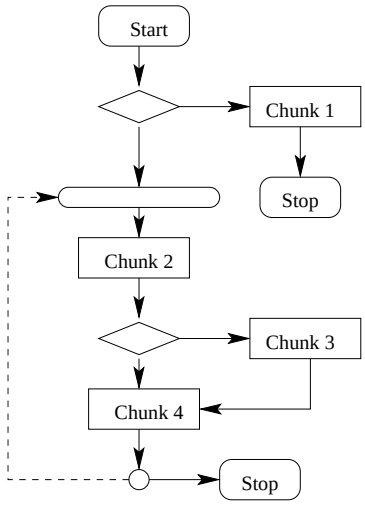

Fig 3: A simple flowchart

The debugging session is run 100 times each with the values $p = 0.2, 0.4, 0.6$ and 0.8. The log-likelihood functions are shown in Fig 4.

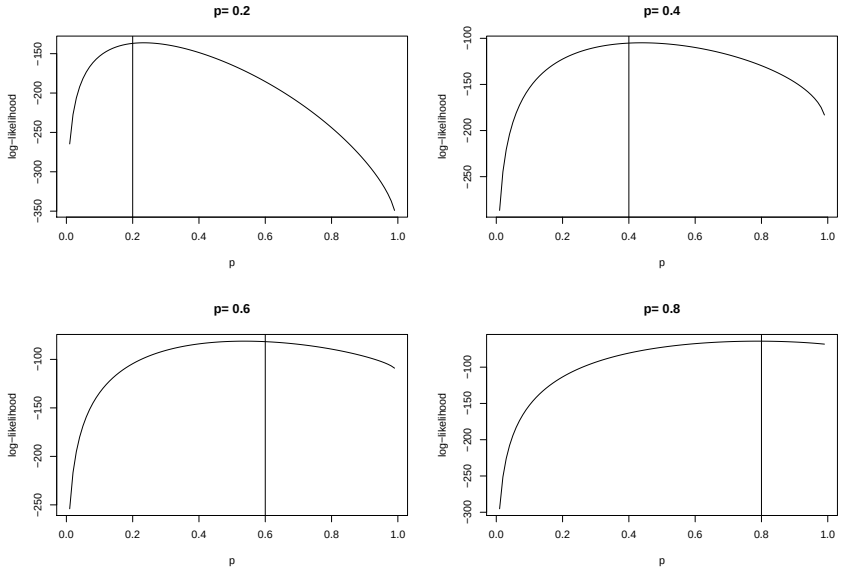

Fig 4: Simulated log-likelihood functions

Each time the peak is very near the actual value of p (shown with a vertical line).

In order to estimate the variances and also to judge the effect of α we apply 50-run simulations 100 times for three different values $p = 0.3, 0.6, 0.9$ and three different values of $\alpha = 0.3, 0.6, 0.9$. The resulting mean MLE's are as follows.

		$\alpha = 0.3$	$\alpha = 0.3$	$\alpha = 0.3$
$p = 0.3$	0.3269	0.3161	0.3057	
$p = 0.6$	0.6096	0.6080	0.6155	
$p = 0.9$	0.9178	0.9083	0.9006	

The corresponding variances are

		$\alpha = 0.3$	$\alpha = 0.3$	$\alpha = 0.3$
$p = 0.3$	0.005	0.003	0.002	
$p = 0.6$	0.015	0.010	0.006	
$p = 0.9$	0.009	0.008	0.006	

6 Variants

The main aim of this paper is to present practically applicable software debugging procedure. In view of practical implementation the main idea presented in the last section may need to be changed to some extent. We discuss some such variations in this section.

6.1 Block-specific bug probabilities

The assumption that each block has the same *a priori* chance of being buggy may be stretching imagination a bit too far. After all, a block consisting of just a few lines of initialization is lot less likely to spring a bug than a block containing many lines of complex code. Yet allowing each block to have its own p will cause an explosion in the number of parameters. A reasonable balance may be obtained by positing that that the chance of bugginess of a block is related to the number of lines in it

as follows:

$$P(\text{a block is buggy}) = 1 - (1-p)^K, \qquad (*)$$

where K is the number of lines in the block, and $p \in (0,1)$. This formula may be motivated by considering each line to have probability p of springing a bug, and assuming that the event that one line is buggy is independent of another line being buggy. Then (*) gives the chance of the block containing at least a single bug.

In this case the likelihood function is of the form

$$L(p) \propto p^m \prod_{i=0}^{k} \prod_{j=1}^{n_i} (1 - (1 - (1-p)^{K_{ij}})\alpha^i).$$

As before this is just the form. Trying to interpret the quantities like K_{ij} would not lead to any easy way to compute it for a given data set. One must rely on automated tools for its evaluation. Complicated as it is, the log-likelihood function still obeys the lemmas given earlier.

6.2 Classification of blocks

The inefficiency of debugging may depend on the type of code a block contains. A block involving numerical analysis is typical harder to debug than one consisting of some initialization commands. Accordingly it may be possible to classify all the blocks into a small number of broad categories , and assign different debugging inefficiency factors to these.

6.3 Multi-block debugging

After a program terminates in an undesirable fashion (*e.g.*, crashes or gives wrong output), a programmer has to carefully go through the code to identify the corresponding bug. In some rare situations the bug may not be confined in a single block, as we have assumed so far. But our approach can be easily adapted to such a scenario. The programmer is to

just record the lines that needed correction. All the corresponding blocks are then marked as buggy, and the logged record for the run is truncated up to and including the first occurrence of *any* of these blocks. Also the chance of these blocks still remaining buggy is updated by scaling down the current probabilities with a factor α.

The corresponding change to the log-likelihood is notationally cumbersome, but easily effected in a computer.

6.4 Bug detected, but not removed

Since software debugging is often done under resource constraints, not all bugs whose presence can be detected can be removed. This is especially true about bugs that are deemed more "esoteric" in nature. In such a case we can still apply our approach, but we do not scale down the probability for the block.

7 Conclusion

In this paper we have proposed a new software reliability technique. The technique seeks to integrate *ad hoc* practical debugging methods with statistical modelling. We have mentioned how a classical debugging session can be easily cast into the new approach with the use of the software tools.

The author likes to thank Prof Anup Dewanji for introducing him to this area of research.

8 Some theoretical underpinning

Throughout the paper we have made the assumption that the existence of bug(s) in a block/line is independent of the existence of a bug in another block/line. We shall try to justify this heuristic statement here in terms of rigorous probability argument. Notice that when we say that "a block

is found buggy" we actually mean two things:

1. that the block contains a bug,
2. that this bug is triggered by the data we are using.

So there are two sources of randomness. The former is introduced by the programmer, the latter by the user. We shall accordingly employ two probability spaces $(\mathcal{X}, \mathcal{F}_x, P_x)$ and $(\mathcal{Y}, \mathcal{F}_y, P_y)$ for the programmer and user, respectively. We may think of $\mathcal{X}$ as all ways of creating bugs available to the programmer. Similarly, $\mathcal{Y}$ denotes the set of all possible user inputs. We make the following assumption.

Assumptions:

1. The user and the programmer behave independently.
2. Let $A_1, A_2 \subseteq \mathcal{X}$ be the events that there are bugs in two (disjoint) blocks. Then A_1, A_2 are mutually independent.
3. Let $B_1, B_2 \subseteq \mathcal{Y}$ be the events that the user chooses an input that triggers any two distinct bugs. Then B_1, B_2 are independent.

Thanks to the first assumption we are working in the product space $(\mathcal{X} \otimes \mathcal{Y}, \mathcal{F}_x \otimes \mathcal{F}_y, P_{xy} \equiv P_x \otimes P_y)$.

If a user encounters a bug in a particular block, this event is

$$A \times B \subseteq \mathcal{X} \otimes \mathcal{Y},$$

where $A \subseteq \mathcal{X}$ is the event that the programmer has left a bug in that block, and $B \subseteq \mathcal{Y}$ denotes the event that the user happens to have chosen an input to trigger it.

Thus when we say that the chance of finding a given block to be buggy is p we actually mean $P_{xy}(A \times B) = p$, and *not* $P_x(A) = p$.

Now let us fix any two distinct blocks and accordingly define $A_i \in \mathcal{F}_x$ as the event of block i containing a bug. Also let $B_i \in \mathcal{F}_y$ be the event

that the user chooses an input value that detects a bug (if any) in block i.

Let $C_i \in \mathcal{F}_{xy}$ be the event that the user actually encounters a bug in block i. Then C_1, C_2 are independent in the product space, because

$$\begin{aligned} P_{xy}(C_1 \cap C_2) &= P_{xy}((A_1 \times B_1) \cap (A_2 \times B_2)) \\ &= P_{xy}((A_1 \cap A_2) \times (B_1 \cap B_2)) \\ &= P_x(A_1 \cap A_2) P_y(B_1 \cap B_2) \\ &= P_x(A_1) P_x(A_2) P_y(B_1) P_y(B_2) \\ &= P_{xy}(A_1 \times B_1) P_{xy}(A_2 \times B_2) \\ &= P_{xy}(C_1) P_{xy}(C_2). \end{aligned}$$

References

[1] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, New York, NY, USA, 2004.

[2] P. Boland. Challenges in software reliability and testing. In *Third International Conference on Mathematical Methods in Reliability*, June 17–20 2002.

[3] A. Dewanji, T.K. Nayak, and P.K. Sen. Estimating the number of components of a system of superimposed renewal processes. *Sankhya, Series A*, 57(3):486–499, 1995.

[4] A.L. Goel and K. Okumoto. Time dependent error detection rate model for software reliability and other performance measures. *IEEE Transactions in Reliability*, R-28:206–211, 1979.

[5] Z. Jelinski and P. Moranda. Software reliability research. In W. Freiberger, editor, *Statistical Computer Performance Evaluation*, pages 465–84. New York: Academic, 1972.

[6] P.B. Moranda. Prediction of software reliability software during debugging. In *Proceedings on the 1975 Annual Reliability and Maintainability Symposium*, pages 327–32, 1975.

[7] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability, Measurement, Prediction, Application*. New York: Wiley, 1987.

[8] T.K. Nayak. Estimating population size by recapture sampling. *Biometrika*, 75(1):113–20, 1988.

[9] N.D. Singpurwalla and R. Soyer. Assessing (software) reliability growth using a random coefficient autoregressive process and its ramifications. *IEEE Transactions on Software Engineering*, 11:1456–1464, 1985.

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

[11] M. Xie. Software reliability models - past, present and future. In N. Limnios and M. Nikulin, editors, *Recent Advances in Reliability Theory: Methodology, Practice and Inference*, pages 323–340. Birkhauser, 2000.