Imperfect Debugging in Software Reliability

Tevfik Aktekin and Toros Caglar*

University of New Hampshire
Peter T. Paul College of Business and Economics
Department of Decision Sciences
and United Health Care*

July 9, 2013
Outline:

- What is Meant by Software Reliability and Debugging?
- Motivation and Uncertainties of Interest
- Very Brief Overview of Software Reliability Models
- The Proposed Model and Its Bayesian Inference
- Numerical Applications of the Proposed Model
- Investigating The Existence of Perfect vs. Imperfect Debugging in Software Data
**Definition**

**Software Reliability:** Can be defined as the probability of successful performance of software for a specified time interval under certain conditions. The fact that the software runs without any problems implies that the code is generating the intended output.

- A failure is said to occur on a piece of software, when a fault (or a so called bug) causes the software to fail.
- **Note:** If a failure does not occur for a period of time, this does not necessarily imply that the software is bug free.
The goal of software testing (i.e. debugging) is to detect/fix software faults (bugs) inherent in the software code and to decide when to release the software.

Software testing is expensive. Thus, there is a trade-off between releasing a reasonably good piece of software and keeping to debug.

There are many examples of buggy software being released and negatively affecting sales (first impression in the market in gaming software for instance).

The testing stage consists of several consecutive program executions. Whenever a failure occurs, the software engineer attempts to fix the problem.

As the faults reveal themselves and are eliminated by the software engineer, the reliability of the software tends to increase if no new faults are introduced to the software during the elimination stage.
Perfect vs. Imperfect Debugging

**Definition**

**Perfect Debugging:** If during the debugging stage the fault which caused the failure has been eliminated permanently and no new faults are introduced, then a perfect debugging is said to have occurred (software reliability gets better).

**Imperfect Debugging:** Loosely speaking, if during the debugging stage a fault is detected and is not eliminated permanently (ex: by introducing new faults), then an imperfect debugging is said to have occurred (software reliability stays the same or worsens).

Note that there are many possible definitions of imperfect debugging (dealing with multiple faults vs. a single fault or the type of software being dealt with as in gaming vs. word processing).
What is Meant by Software Reliability?

- $P(T|\theta)$ is usually referred to as a software reliability model, where $T$ represents the time to failure of a piece of software. It represents the probability that a particular piece of software will fail in a given interval.
- $P(T \geq t|\theta)$ for some $t \geq 0$ represents the reliability function of $T$.
- **Note** that here the notion of time is based on the mission time $t$, in other words the time during which the software is executed.
Carry out inference on the unknown parameters such as the number of faults inherent in the code and fault detection rate after each stage of testing.

Obtain the (predictive) reliability function after each debugging stage as a function of the parameters of interest.

\[ P(T_{i+1} \geq t | \theta, D). \]  

Making a decision on whether to stop the testing and release the software based on the reliability assessment of the software at time \( t \).
The following two can be considered to be the building blocks for most of the current software reliability models:

- **Jelinski and Moranda (JM) Model (1972)**
  - Each fault is permanently removed upon failure (perfect debugging).
  - Each fault contributes equally to the failure rate at any stage of the testing.

- **Littlewood and Verall (LV) Model (1973)**
  - Both assumptions from the JM Model have been relaxed by the LV Model; perfect debugging and equally likely contributions from faults at each stage of debugging.
Proposed Model

- Consider modeling of a multiplicative failure rate model whose components are evolving stochastically over testing stages (an NHPP type model).

- Proposed model is based on the Jelinski Moranda (JM) model as is the case for most subsequent work in the software reliability literature.

- Two of the main assumptions of the JM model is that every fault contributes equally to the failure rate at any stage of testing and that each fault is removed permanently upon failure.

- Consider the case where each fault is removed permanently upon failure, along with the possibility of introducing new faults during debugging.
Proposed Model and Definitions

- $t_i$, for $i = 1, \ldots, N$: time until failure during (or prior to) the $i$th stage of testing.
- $\phi_i$, for $i = 1, \ldots, N$: fault detection rate per fault during (or prior to) the $i$th stage of testing.
- $\lambda_i$, for $i = 1, \ldots, N$: the number of faults present on the software code during (or prior to) the $i$th stage of testing.
- $\lambda_1$: the number of faults present on the software code during (or prior to) the first stage of testing (i.e. prior to testing).
  Note that $\lambda_i$s will be functions of $\lambda_1$ and $\phi_i$s.
- $\phi_i \lambda_i$: the failure rate of the software during (or prior to) the $i$th stage of testing.
Assume that the inter-failure times, $t_i$, are exponentially distributed. The likelihood would be

$$L(\Phi, \Lambda; D) = \prod_{i=1}^{N} \phi_i \lambda_i \exp\{-t_i \phi_i \lambda_i\}, \quad (2)$$

where $\Phi = \{\phi_1, \ldots, \phi_N\}$, $\Lambda = \{\lambda_1, \ldots, \lambda_N\}$ and $D = \{t_1, \ldots, t_N\}$. Therefore the joint posterior of $\Phi$ and $\Lambda$ would be given by

$$p(\Phi, \Lambda|D) \propto \prod_{i=1}^{N} \phi_i \lambda_i \exp\{-t_i \phi_i \lambda_i\} p(\Phi)p(\Lambda). \quad (3)$$
Modeling the fault detection rate per fault, $\phi_i$

$s$ be given by the following power law relationship:

$$
\phi_i = \phi^\beta_{i-1} \times \nu_i, \text{ for } i = 1, \ldots, N,
$$

where $\nu_i \sim LN(0, \sigma^2)$. We can obtain the following the linear model in logarithms:

$$
\log(\phi_i) = \beta \log(\phi_{i-1}) + \epsilon_i, \text{ for } i = 1, \ldots, N,
$$

where $\epsilon_i = \log(\nu_i)$. (5) is a first order autoregressive process of the latent fault detection rates per fault in the log scale. The conditional distributions of $\log(\phi_i)$s can be written as

$$
\log(\phi_i) | \log(\phi_{i-1}), \beta \sim N(\beta \log(\phi_{i-1}), \sigma^2), \text{ for } i = 1, \ldots, N,
$$

Note the scale of the inter-failure times (all above one in our numerical example). $\phi_i$s were all fractions and $\beta$ was found to be between zero and one. To keep the interpretation of $\beta$ consistent, it is always possible to rescale the data such that they are all above one.
Modeling the fault detection rate per fault, $\phi_i$

The relationship implied by (4) also dictates the type of debugging that occurs during the $i$th debugging stage.

- If $\phi_i < \phi_{i-1}$, then perfect debugging is said to have occurred.
- If $\phi_i \geq \phi_{i-1}$, then imperfect debugging is said to have occurred.

In other words, when a failure is detected at the $(i - 1)$th failure epoch, a fault has been detected and repaired, however a new fault was introduced during the same debugging stage. $\beta$ determines on average how the fault detection rate per fault is changing from stage to stage. For instance, when $0 < \phi_{i-1} < 1$ and $\beta > 1$ then perfect debugging tends to occur, conversely when $0 < \beta < 1$ then imperfect debugging tends to occur.
Modeling the total number of faults, $\lambda_i$

Conditional on whether perfect or imperfect debugging has occurred during the previous debugging stage, the total number of faults left in the software code, $\lambda_i$, is assumed to have the following structure

$$\lambda_i = \lambda_{i-1} - \gamma_i, \text{ for } i = 1, \ldots, N$$

(7)

where

$$\gamma_i = 1, \text{ with probability } p(\phi_i < \phi_{i-1})$$

$$= 0, \text{ with probability } p(\phi_i \geq \phi_{i-1}) \text{ for } i = 1, \ldots, N.$$  

In (7), $\gamma_i$ is a Bernoulli process whose probability of success is the probability of perfect debugging, $p(\phi_i < \phi_{i-1})$. When perfect debugging occurs, $\lambda_i$ goes down by one unit, since the fault that has caused the failure has been found and fixed. When imperfect debugging occurs, $\lambda_i$ stays the same, since the fault that has caused the failure has been found and fixed, however a new fault has been introduced while fixing the previous one.
Other Model Priors

For the model on $\phi_i$s,

$$\sigma^2 \sim \text{Gamma}(a, b)$$  \hspace{1cm} (8)

and

$$\beta \sim \text{U}(c, d)$$ \hspace{1cm} (9)

and for the initial fault detection rate, $\phi_1$, we assume the following

$$\phi_1 \sim \text{LN}(e, f)$$ \hspace{1cm} (10)

For the initial number of inherent faults, $\lambda_1$, we assume the following

$$\lambda_1 \sim \text{Poisson}(\theta)$$ \hspace{1cm} (11)

with

$$\theta \sim \text{Gamma}(g, h).$$ \hspace{1cm} (12)
Goal: To generate samples from

\[
p(\Phi, \lambda_1, \theta, \beta, \sigma^2 | D) \propto \prod_{i=1}^{N} \phi_i X_i \exp\{-t_i \phi_i \lambda_i\} p(\Phi | \beta, \sigma^2) p(\lambda_1 | \theta) p(\beta) p(\sigma^2) p(\theta). \quad (13)
\]

In (13), the conditional joint prior distribution for \( \phi_i \)'s using the chain rule and dropping terms that are independent can be obtained as

\[
p(\Phi | \beta, \sigma^2) = p(\phi_N | \phi_{N-1}, \beta, \sigma^2) \ldots p(\phi_2 | \phi_1, \beta, \sigma^2) p(\phi_1). \quad (14)
\]
To generate the full conditionals

- $p(\phi_i|\ldots,D)$ for $i = 1,\ldots,N$: Use Metropolis-Hastings.
- $p(\lambda_1|\ldots,D)$: Discrete. In addition, one can compute $\lambda_j$ as $\lambda_{j-1} - 1(\phi_j < \phi_{j-1})$ for $j = 2,\ldots,N$ once we have the required samples.
- $p(\theta|\ldots,D)$: Gamma
- $p(\beta|\ldots,D)$: Normal
- $p(\sigma^2|\ldots,D)$: Use Metropolis-Hastings.
To generate samples from $p(\Phi, \lambda_1, \theta, \beta, \sigma^2|D_N)$

1. Assume the starting points $(\lambda_1^{(0)}, \phi_2^{(0)}, \theta^{(0)}, (\sigma^2)^{(0)}, \beta^{(0)})$ and set $l=1$.
2. Generate $\phi_1^{(l)}$ using $\lambda_1^{(l-1)}, \phi_2^{(l)}, \beta^{(l-1)}$ and $(\sigma^2)^{(l-1)}$ from $(\Phi|\ldots, D)$.
3. Sequentially generate $\phi_i^{(l)}$ for $i = 2, \ldots, N$ using $\lambda_1^{(l-1)}, \beta^{(l-1)}, (\sigma^2)^{(l-1)}$ and $\phi_{i-1}^{(l)}$ from $(\Phi_i|\ldots, D)$.
4. Generate $\beta^{(l)}$ using $(\sigma^2)^{(l-1)}$ and $\phi_i^{(l)}$ for $i = 1, \ldots, N$ from $(\beta|\ldots, D)$.
5. Generate $(\sigma^2)^{(l)}$ using $\beta^{(l)}$ and $\phi_i^{(l)}$ for $i = 1, \ldots, N$ from $(\sigma^2|\ldots, D)$.
6. Generate $\lambda_1^{(l)}$ using $\phi_i^{(l)}$ for $i = 1, \ldots, N$ and $\theta^{(l-1)}$ from $(X_1|\ldots, D)$.
7. Sequentially compute $\lambda_i^{(l)}$ for $i = 2, \ldots, N$ using $\lambda_{i-1}^{(l)}$ and $\phi_i^{(l)}$ for $i = 1, \ldots, N$ via $\lambda_i = \lambda_{i-1} - 1(\phi_i < \phi_{i-1})$.
8. Generate $\theta^{(l)}$ using $\lambda_1^{(l)}$ from $(\theta|\ldots, D_N)$.
9. Set $l=l+1$ and go back to step 1.
Numerical Example

- Dataset 1: The numerical application of our model is carried out on the well known dataset first reported in JM 1972. The dataset consists of 31 software inter-failure times, 26 of which were obtained during the production stage of debugging and the remaining 5 during the rest of the testing stage. In our example, all 31 inter-failure times were used (most inference is based on this one).

- Dataset 2: The military systems application data (data 17) of John Musa of Bell Telephone Laboratories with 38 inter-failure times. (only used for comparison purposes).

- Model Comparison: Use the harmonic mean estimator of the marginal likelihood and the DIC.

- Models to Compare Against: MS88-M1, MS88-M2, KY96-GOS-W, KY96-RVS-P and a simple perfect debugging (PD) model.

1http://www.thedacs.com/databases/sled/swrel.php
### Numerical Example

**Table: log\{p(D)\} and DIC for the JM dataset**

<table>
<thead>
<tr>
<th>ID</th>
<th>PD</th>
<th>MS88-M1</th>
<th>MS88-M2</th>
<th>KY96-GOS-W</th>
<th>KY96-RVS-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>log{p(D)}</td>
<td></td>
<td>-108.55</td>
<td>-116.51</td>
<td>-113.81</td>
<td>-114.01</td>
</tr>
<tr>
<td>DIC</td>
<td></td>
<td>215.49</td>
<td>227.76</td>
<td>226.24</td>
<td>223.26</td>
</tr>
</tbody>
</table>

**Table: log\{p(D)\} and DIC for the MUSA dataset**

<table>
<thead>
<tr>
<th>ID</th>
<th>PD</th>
<th>MS88-M1</th>
<th>MS88-M2</th>
<th>KY96-GOS-W</th>
<th>KY96-RVS-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>log{p(D)}</td>
<td></td>
<td>-50.41</td>
<td>-57.87</td>
<td>-53.98.81</td>
<td>-53.36</td>
</tr>
<tr>
<td>DIC</td>
<td></td>
<td>101.95</td>
<td>114.52</td>
<td>107.89</td>
<td>105.45</td>
</tr>
</tbody>
</table>
Figure: Boxplots of $\phi_i$ for $i = 1, \ldots, 31$ (left) and the probability of perfect debugging vs. debugging stages (right)
Summary of Findings

Figure: Posterior distribution plots of $\beta$, $\phi_1$, $\theta$ and $\lambda_1$
Figure: Boxplots of $\lambda_i$ for $i = 1, \ldots, 31$
Summary of Findings

Figure: Boxplot of the failure rates, $\phi_i \lambda_i$, for $i = 1, \ldots, 31$
The predictive reliability function during the $i$th testing stage (given that the software has gone through $(i - 1)$th stages of testing) can be computed via

$$R(t_i|D^{(i-1)}) = 1 - \frac{1}{S} \sum_{j=1}^{S} F(t_i|\phi^{(j)}_i, \lambda^{(j)}_i, D^{(i-1)}) . \tag{15}$$

Once (15) is estimated it can easily be used as part of a software reliability optimal release scheme.

**Figure:** Predictive reliability functions for $i = 29, \ldots, 31$
Introduce a Markov chain type of structure on the number of bugs repaired or introduced during the debugging stage instead of the current Bernoulli setup.

Investigate the possibility of state space evolution of the $\beta$ coefficient in the power law relationship between the inter-failure times.

**Note:** Both extensions would be challenging from an MCMC estimation point of view.