The Software Failure and Reliability Assessment Tool (SFRAT): A Guide for Contributors

[1]Richard Muri, [1]Vidhyashree Nagaraju, [1]Venkteswaran Shekar, [1]Lance Fiondella, and [2]Thierry Wandji

[1] *University of Massachusetts Dartmouth, MA, USA*
Email: {rmuri,vnagaraju,vshekar, lfiondella}@umassd.edu
Phone:

Abstract

This paper details contribution of a software reliability model to SFRAT, a free and open source application to promote the quantitative assessment of software reliability. SFRAT has been designed to serve as a single shared framework for collaborative research, experimentation, and use by practitioners. Unlike other tools, the SFRAT architecture enables the addition of existing software reliability models, promoting more systematic comparison of models than previously possible. Using the Musa-Okumoto model as an example, the process of adding a model is explained in depth. The tool is presently being used by federal government agencies, federally funded research and development centers, and private companies. Contributing to the tool will therefore enable software reliability researchers to reach high profile users of their results.

Keywords: Software reliability, software reliability growth model, open source tool, GitHub, R statistical programming language, Shiny

Acronyms

SFRAT	Software Failure and Reliability Assessment Tool
SRGM	Software reliability growth model
SMERFS	Statistical Modeling and Estimation of Reliability Functions for Software
CASRE	(Computer - Aided Software Reliability Estimation tool
SRATS	Software Reliability Assessment Tool on Spreadsheet
FT	Failure times
FN	Failure number
FC	Failure count
IF	Interfailure times
CFC	Cumulative failure count
ECM	Expectation conditional maximization
NM	Newton's method
MO	Musa-Okumoto SRGM
MLE	Maximum likelihood estimation
TBF	Times between failure
LL	Log-likelihood
MVF	Mean value function
MTTF	Mean Time to Failure
FI	Failure Intensity
OR	Optimal release
CC	Cost constrained
RC	Reliability constrained

Notation

$m(t)$	MVF of NHPP
λ	Instantaneous failure rate
a	Rate of reduction in the normalized failure intensity per failure
b	Initial failure intensity
$\mathbf{T}$	Vector of failure times
n	Length of failure data
t_n	Time to n^{th} failure
$\hat{}$	Maximum likelihood estimate of a parameter
R	Reliability
Δ	Mission length
T	Software lifecycle
$C(t)$	Total estimated cost at time t
c_1	Cost of faut removal during testing
c_2	Cost of fault removal after release
c_3	Cost of testing per unit time
t^*	Optimal release time considering cost
s^*	Optimal release time considering reliability

1. Introduction

Many critical aspects of modern society depend on reliable software systems to ensure convenience, safety, and security. Researchers have developed a large number of models to quantitatively assess the reliability of software (Lyu et al., 1996), which is commonly defined as the probability of failure free operation during a specified period of time in a specified environment. However, software engineers are often unfamiliar with the underlying mathematics which has limited the impact of software reliability research. To overcome this limitation, some researchers (Farr and Smith, 1993; Lyu and Nikora, 1992; Okamura and Dohi, 2013) have implemented tools to automatically apply software reliability growth models (SRGM) that fit software failure data to make useful predictions about the software system such as the number of remaining faults, reliability, failure intensity, mean time to failure as well as estimation of optimal release time. Despite these contributions over the past several decades, there is no single tool to which the international software reliability research community can contribute.

Some of the automated tools for software reliability engineering includes SMERFS (Statistical Modeling and Estimation of Reliability Functions for Software) (Farr and Smith, 1993) which was then incorporated into CASRE

(Computer - Aided Software Reliability Estimation tool) (Lyu and Nikora, 1992) and SRATS (Software Reliability Assessment Tool on Spreadsheet) (Okamura and Dohi, 2013). While these tools promote the accessibility of software reliability research to the user community, each contains a limited number of models and have not been designed in a manner that would easily allow other researchers to add additional models. Moreover, these tools are often dependent on a particular operating system and programming language which may require the user to pay a fee in order for them to be able to use the tool.

To overcome these shortcomings, we have developed the Software Failure and Reliability Assessment Tool (SFRAT), which is both free and open source. The main objective of this project is to promote collaboration among members of the international software reliability research community as well as users from industry and government. The application has been implemented in the R programming language which is an open source environment for statistical computing. This paper explains the architecture of the tool and the steps required to add a new model. In order to use this guide, the reader must have a functional local copy of the SFRAT and the RStudio IDE. Installation instructions can be found in the user guide available at https://sasdlc.org/lab/projects/resources/SFRAT/SFRAT_user_guide.pdf.

The remainder of the paper is organized as follows: Section 2 contains a top-level overview of the tool architecture, Section 3 describes the files contributors must submit, Section 4 provides background information on the Musa-Okumoto model, Section 5 details required functions and provides example code, Section 6 describes the debugging and verification process, and Section 7 shows how to submit a completed model.

2. Tool Architecture

To contribute a model, no knowledge of the inner workings of SFRAT is required. However, a basic understanding of the architecture of the tool is useful. SFRAT is built using the Shiny framework for the R language. R is used to perform statistical analysis and provide data for graphing, and Shiny generates a webpage to display the results graphically. A top level abstract view of the architecture is shown in Figure 1.

The only required component to use the SFRAT is input data in the form of a Microsoft Excel file (.xlsx) or a comma-separated value file (.csv). One of the three types of input data format can be assessed by the SFRAT

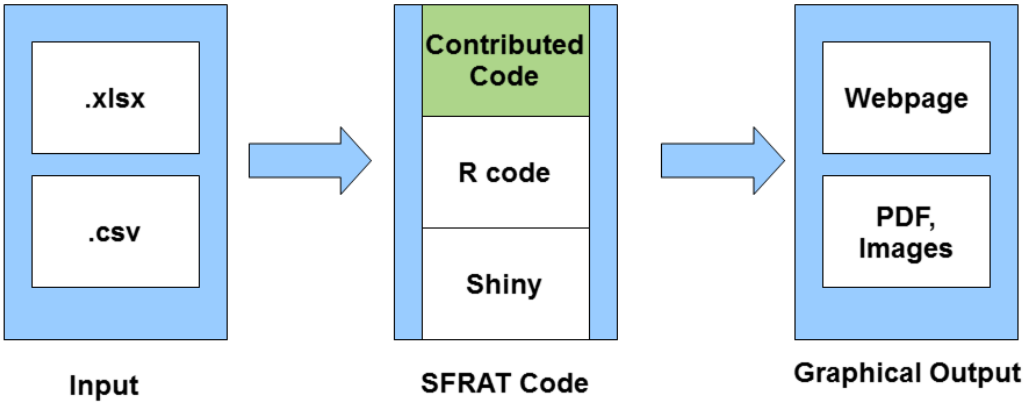

Figure 1: Basic architecture of SFRAT

including failure counts (FC) (the number of failures that have occured in a given interval), failure times (FT) (the time at which failures have occurred), or interfailure times (IF) (times between two successive failures) data. Failure counts can also be provided as cumulative failure counts (CFC). Input data must be formatted as follows: For failure times data, the first column is an index with column header 'FN' indicating failure number, while the second and third column is labelled 'FT' for failure times, or 'IF' for interfailure times. Similarly, for failure count data, the first column is labeled as 'T' for testing time, while the corresponding columns are labeled as 'FC' for failure counts, 'CFC' for cumulative failure counts,

Model contributors must provide two additional files: an R file containing function definitions for their model (see Section 3.1), and an R file containing information about the model type (see Section 3.2). Implementation of these files is discussed in detail in Section 3. Figure 2 shows how contributed code is integrated in the file hierarchy of SFRAT using the Musa-Okumoto model as an example. The first column shows the top level directory, the second column the directory containing model specific files, the third column those files, and the fourth column the model function definition names.

If any additional R packages are used, the install_script.R that appears in the tool's top level directory must be updated; add to the bottom of the file the following line of code, replacing packageName and repoName with the appropriate package name and repository name. Contributors are discouraged from using additional packages unless it is absolutely necessary.

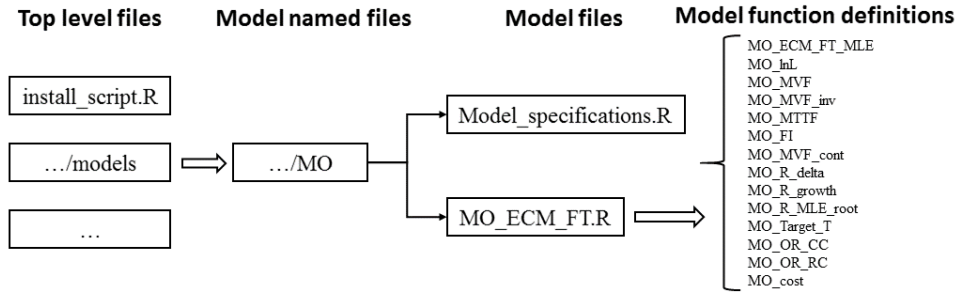

Figure 2: Model specific files for contributor's

```
if(!require(packageName)) {install.packages(''packageName'',
    repos=''repoName'');library(packageName)}
```

3. Required Files

Models are defined within the tool under the .../srt.core/models/<Model_name> directory. At least two files are required: Model function definitions and model specifications.R.

3.1. Model Function Definitions

One or more file must contain function definitions for the model. Additional files beyond the first can be used to implement alternative parameter estimation methods and alternative input data types. The files are named following the convention <Model_name>_<Method>_<Data_Type>.R. <Method> refers to the parameter estimation method used in the maximum likelihood estimation; it can be any string, but it is recommended that a two or three character abbreviation is used. <Data_type> refers to the input data type. It may take one of three values, "FT", "IF", or "FC".

For example, the Musa-Okumoto model definitions include the following file: .../srt.core/models/MO/MO_ECM_FT.R. This file contains function definitions for the Musa-Okumoto model using the expectation conditional maximization algorithm (Nagaraju et al., 2017) for parameter estimation with failure time input data. If one wanted to implement parameter estimation using Newton's method (NM) and interfailure times, the file would be named MO_NM_IF.R.

Table 1 provides detailed list of the fourteen functions to be implemented in the SFRAT tool.

Table 1: Functions implemented in the SFRAT

Function Name	Example Function	Function Summary
Maximum Likelihood Estimation	MO_ECM_FT_MLE	Estimate initial model parameters
Log-Likelihood	MO_lnL	Calculate log of the likelihood
Mean Value	MO_MVF	Calculates the mean value
Inverse Mean Value	MO_MVF_inv	Calculates inverse mean value
Mean Time to Failure	MO_MTTF	Calculate mean time to failure
Failure Intensity	MO_FI	Calculate failure intensity
Continuous Mean Value	MO_MVF_cont	Wrapper function for mean value at single point
Change in Reliability Over Time	MO_R_Delta	Calculate difference of mean values at different times
Reliability Growth	MO_R_growth	Wrapper function for change in reliability with time
Root Finding Equation	MO_R_MLE_root	Helper function for Time to Target Reliability
Time to Target Reliability	MO_Target_T	Calculate time to reach target reliability
Optimal Release Time Based on Cost	MO_OR_CC	Calculate release time as a function of cost
Cost at Software Lifecycle Time	MO_cost	Calculate total cost over software lifecycle
Optimal Release Time Based on Reliability	MO_OR_RC	Calculate release time as a function of reliability

3.2. Model_specifications

The file model_specifications.R defines the properties of the model being implemented. It contains seven fields, each preceded by <Model_name>_: input, methods, params, numfailsparam, fullname, plotcolor, and Finite.

3.2.1. Musa-Okumoto Model Specifications

Shown below is the full model specifications file for the Musa-Okumoto (MO) model (Musa and Okumoto, 1984) provided with SFRAT. It can be found in the .../srt.core/models/MO directory. Each field is described in greater detail below.

```
# Musa-Okumoto model
MO_input <- c("FT")
MO_methods <- c("ECM")
MO_params <- c("aMLE", "bMLE")
MO_numfailsparm <- c(1)
MO_fullname <- c("Musa-Okumoto")
MO_plotcolor <- c("green")
MO_Finite <- TRUE
```

3.2.2. Input

Input is a string vector describing the input data type(s) a model requires. If multiple implementations of the model are created, input may take multiple values. Possible values are “FC”, “FT”, and “IF”. If a function definition of a model uses a certain input type, that type must be included. For instance, the Musa-Okumoto model has function definitions for FT input data, as shown by the file name MO_ECM_FT.R. The input variable in the model specifications is therefore set as shown below.

```
MO_input <- c("FT")
```

3.2.3. Methods

Methods takes a string that describes the optimization method used to estimate the parameters used by the model. If multiple alternative methods are provided, the field must contain a value for each. For instance, the Musa-Okumoto model uses the expectation conditional maximization (ECM) algorithm, so the MO_methods variable contains one string ”ECM”. If an

alternative implementation were created using Newton's Method, it would contain two strings, "ECM" and "NM".

```
MO_methods <- c("ECM")
```

3.2.4. Params

Params describes the curve fitting parameters used by the model. A value must be assigned to each parameter. Any string is acceptable, however the recommended naming convention is "aMLE", "bMLE", "cMLE" etc. for as many parameters as exist. For example, the Musa-Okumoto model uses two parameters, and assigns the params variable as shown below.

```
MO_params <- c("aMLE", "bMLE")
```

3.2.5. Numfailsparms

Numfailsparam indicates the integer index of the parameter associated with number of failures. For the Musa-Okumoto model, the a parameter ("aMLE" in params) is the parameter related to failure number; because it is the first parameter listed, the index of 1 is appropriate as shown below.

```
MO_numfailsparm <- c(1)
```

3.2.6. Fullname

Fullname is a string that indicates the unabbreviated name for the model.

```
MO_fullname <- c("Musa-Okumoto")
```

3.2.7. Plotcolor

Plot color is an optional parameter that indicates a color preference for plotting the model. It takes a string color name. Acceptable values may be found in the R documentation.

```
MO_plotcolor <- c("green")
```

3.2.8. Finite

The finite variable is boolean, describing whether a model assumes an infinite number of failures, or if it will converge to a maximum value. The Musa-Okumoto model assumes eventually all failures will be fixed, thus it is a finite model. Acceptable values are TRUE or FALSE.

```
MO_finite <- TRUE
```

4. Musa-Okumoto Logarithmic Poisson Model

This section describes the mathematical equations associated with adding the Musa-Okumoto model to the SFRAT.

The mean value function of the Musa-Okumoto (Musa and Okumoto, 1984) model is

$$m(t) = \frac{1}{a}\log(abt + 1) \tag{1}$$

where a is the rate of reduction in the normalized failure intensity per failure and b is the initial failure intensity.

The failure intensity is

$$\lambda(t) = \frac{\partial(m(t))}{\partial t} = \frac{b}{abt + 1} \tag{2}$$

The times between failures or interfailure time is

$$TBF = \frac{1}{\lambda(t)} \tag{3}$$

Let $\mathbf{T} = \langle t_1, t_2, \ldots, t_n \rangle$ be the vector of failure times, then the log-likelihood function is

$$LL = -m(t_n) + \sum_{i=1}^{n} \log\left(\lambda(t_i)\right) \tag{4}$$

Therefore, the log-likelihood function of the MO model is

$$LL = -\frac{1}{a}\log(abt_n + 1) + \sum_{i=1}^{n} \log\left(\frac{b}{abt_i + 1}\right) \tag{5}$$

The maximum likelihood estimates of parameters a and b are obtained by taking partial derivatives of Equation (5) with respect to each model

parameters and then solving by equating it to zero. The MLEs of a and b are

$$\hat{a} := -\frac{-bt_n}{a(abt_n+1)} + \frac{\log(abt_n+1)}{a^2} - \sum_{i=1}^{n} \frac{bt_i}{abt_i+1} = 0 \tag{6}$$

and,

$$\hat{b} := -\frac{t_n}{abt_n+1} + \sum_{i=1}^{n} \frac{1}{b(abt_i+1)} = 0 \tag{7}$$

The partial derivatives in Equations (6) and (7) are solved by applying the expectation conditional maximization (Nagaraju et al., 2017) algorithm.

Reliability growth is

$$R(\Delta) = e^{-(m(t+\Delta)-m(t))} \tag{8}$$

where Δ is the mission length.

4.1. Optimal release

The cost equation is

$$C(t) = \frac{1}{a}\left((c_1 - c_2)\log(1+abt) + c_2\log(1+abT)\right) + ac_3t \tag{9}$$

where c_1 is the cost of removing a fault during testing, c_2 is the cost of removing a fault after release ($c_2 > c_1$), c_3 is the cost of testing per unit time, and T is the length of the software lifecycle.

The optimal release time considering cost is

$$t^* = \frac{b(c_2 - c_1) - c_3}{abc_3} \tag{10}$$

The optimal release time considering reliability is

$$s^* = -\frac{1}{b}\left(\log(\log\left(\frac{(1+abt)}{a}\right) - \log\left(\log(R)\right)\right) \tag{11}$$

5. Coding a Custom Model

SFRAT works by making function calls defined by the model creator. The model_specifications.R file provides all the information necessary to call the functions of a custom model. The model creator must now define 14 required functions listed in Table 1 in the <Model_name>_<Method>_<Data_Type>.R file to integrate fully their model with SFRAT. An in depth description of each required function follows.

5.1. Maximum Likelihood Estimation Function

The Maximum Likelihood Estimate (MLE) function computes the maximum likelihood estimates of the parameters of a model, given a particular data set as input. The technique refers to applying input data to estimate model parameters to maximize the likelihood that the model fits the input data. It requires the log-likelihood function described later in this section. The function is named according to the model initials, parameter estimation method, and input data type as shown in the following format: <Model name>_<method>_<Data type>_MLE. For example, MO_ECM_FT_MLE refers to the Musa-Okumoto model, using the Expectation Conditional Maximization algorithm on Failure Time data to calculate the maximum likelihood parameter estimates. Acceptable values for method and data type are taken from the file name containing the MLE function for the model.

The MLE function has one input: a vector of failure time or failure count data, as determined by the data type field in the name. Most models will use an optimization method to approximate the ideal parameter values for that particular data set. The parameters are returned as a data frame. The recommended naming scheme is <Model_name>_aMLE, <Model_name>_bMLE, <Model_name>_cMLE etc. This naming scheme must match the params variable of model_specifications.R. Shown below is an example template taken from the Musa-Okumoto model, using the expectation conditional maximization algorithm. This implementation is based on Equations (6) and (7) from Section 4. Other methods may be used; suggested methods include Newton's method or the bisection method.

```
MO_ECM_FT_MLE <- function(inputData){
  # inputData is a vector of failure times; if another data type is
      desired, the name of the MLE function should be changed to
      match the desired input type, such as MO_ECM_IF_MLE for
      interfailure times, or MO_ECM_FC_MLE for failure counts

  # Vector of failure times
  t <- as.numeric(inputData)
  n <- length(t) # size of input vector
  tn <- t[n] # final value of t

  # Partial derivative of log-likelihood function with respect to
      a. Used as objective function to optimize a parameter
  ahat = function(a, b, t, tn){
```

```
  leftTerm <- -(b * tn)/(a * (a * b * tn + 1)) + log(a * b * tn +
     1)/(a^2)

  rightTerm <- 0
  for(i in 1:n){
    rightTerm <- rightTerm + (b * t[i])/(a * b * t[i] + 1)
  }
  amle <- leftTerm - rightTerm
  return(amle)
}

#  Partial derivative of log-likelihood function with respect to
    b. Used as objective function to optimize b parameter
bhat = function(b, a, t, tn){
  leftTerm <- -tn/(a * b * tn + 1)
  rightTerm <- 0
  for(i in 1:n){
    rightTerm <- rightTerm + 1/(b * (a * b * t[i] + 1))
  }
  bmle <- leftTerm + rightTerm
  return(bmle)
}

# Initial guesses for a and b parameters. These values may need
    to be adjusted depending on the input data
a <- 0.01;
b <- 0.01;

# Log-likelihood of initial guesses
params <- data.frame(``MO_aMLE'' = a, ``MO_bMLE'' = b)
LL <- MO_lnL(t, params)

# Placeholder error value
LLerror = 1;

# First calculation based on initial values. Interval, max
    iterations, and tolerance all may need adjustment depending
    on the input data. Relies on uniroot root finding utility,
    which optimizes a function with respect to the first input
    parameter. Here we are minimizing a based on initial guesses,
```

```
    and b based on the output of the minimization of a
  a <- stats::uniroot(ahat, interval = c(.02, .1), b = b, t = t, tn
    = tn, maxiter=100, tol=1e-10, extendInt=``yes'')$root
  b <- stats::uniroot(bhat, interval = c(.01, .1), a = a, t = t, tn
    = tn, maxiter=100, tol=1e-10, extendInt=``yes'')$root


  params <- data.frame(``MO_aMLE'' = a, ``MO_bMLE'' = b)
  # Log-likelihood of first iteration
  LLnew <- MO_lnL(params, t)
  # Error is current log-likelihood value minus previous value
  LLerror <- LLnew - LL
  # Current log-likelihood becomes previous value
  LL <- LLnew

  # Iterate until an arbitrarily small error is reached
  while (LLerror > 10^-10)
  {
    # Find roots of MLE equations
    a <- stats::uniroot(ahat, interval = c(.001, 1), b = b, t = t,
      tn = tn, maxiter=100, tol=1e-10, extendInt=``yes'')$root
    b <- stats::uniroot(bhat, interval = c(.001, 1), a = a, t = t,
      tn = tn, maxiter=100, tol=1e-10, extendInt=``yes'')$root

    # Calculate new Log-likelihood and error
    param <- data.frame(``MO_aMLE'' = a, ``MO_bMLE'' = b)
    LLnew <- MO_lnL(t, params)
    LLerror <- LLnew - LL

    LL <- LLnew

  }
  # a and b are the calculated parameter values. The solution is
    returned as a data frame with field names based on the params
    variable set in model_specifications.R

  solution <- data.frame(``MO_aMLE''= a, ``MO_bMLE''=b)
  return(solution)
}
```

5.2. Log-Likelihood Function

The log-likelihood function calculates the natural log of the likelihood function. It is equivalent to the negative mean value function evaluated at the final point of the input vector plus the summation of the natural logarithm of the failure intensity function. Its use within SFRAT is to solve parameter estimation equations during maximum likelihood estimation, and to calculate the Akaike Information Criterion. The function is named <Model_name>_lnL. Inputs are the data frame containing MLE parameters and a one column data frame containing failure times. The expected return is a single floating point value representing the log-likelihood of a given data vector and set of parameter estimates. Shown below is an example template taken from the Musa-Okumoto model, based on Equation (5) in Section 4. It is important to note that the function input parameters are declared in the order inputData, params for this function. Every other function follows the order params, inputData.

```
MO_lnL <- function(inputData, params) {
  # Params is a data frame named according to model_specifications.R

  # Define a parameter as calculated by the MLE function
  a <- params$MO_aMLE
  b <- params$MO_bMLE # Define b parameter

  # Row vector of failure times taken from input data
  t <- inputData
  n <- length(t) # Number of rows in the input
  tn <- t[n] # Final value of input vector

  # Leftmost term of lnL. Negative MVF at tn
  leftTerm = -log(a * b * tn + 1)/a

  # Right terms of lnL. Summation of log(MO_FI)
  rightTerm <- 0
  for(i in 1:n){
    rightTerm = rightTerm + log(b/(a * b * t[i] + 1))
  }

  # Return the log-likelihood
  lnL <- leftTerm + rightTerm
```

```
  return(lnL)
}
```

5.3. Mean Value Function

The mean value function (MVF) calculates the mean value for number of failures given a vector of times. The function is named <Model_name>_MVF. The inputs to the function are the parameter estimates, and the input data. The parameters are passed as a single row data frame with column headers named as defined in the MLE function (typically <Model_name>_aMLE, <Model_name>_bMLE, etc). The input data is a data frame with as many rows as there are data points, and two columns: IF and FT. IF (interfailure times) represents the time between failures, and FT (failure times) is the cumulative sum of IF, representing the times at which the failures occurred. The expected return of the function is a data frame with column names "Failure", "Time", and "Model". The failure column contains the result of the MVF, the time column is identical to FT, and the model column contains <Model_name> repeated for each row. Shown below is an example template taken from the Musa-Okumoto model, based on Equation (1) in Section 4.

```
MO_MVF <- function(params, inputData) {
  # Params is a data frame named according to model_specifications.R

  # Define a parameter as calculated by the MLE function
  a <- params$MO_aMLE
  b <- params$MO_bMLE # Define b parameter

  # Row vector of failure times taken from input data
  t <- inputData$FT
  n <- length(t) # Number of rows in the input

  # Calculate the mean value function based on the parameters and
     input failure time data
  MVF <- log(b * a * t + 1)/a

  # MVF is a vector of mean number of failures for each input time
  solution <- data.frame(``Failure'' = MVF, ``Time'' = t, ``Model''
     = rep(``GO'', n))
  return(solution)
}
```

5.4. Inverse Mean Value Function

The inverse mean value function calculates the inverse of the mean value. It is derived by finding the inverse of the MVF. Its use within SFRAT is to predict future failure times given failure counts. For example, if a data set includes 100 failures with listed times, the inverse mean value can be used to estimate what time the 101st failure will occur. The function is named <Model_name>_MVF_inv. Inputs are the data frame containing MLE parameters and a one column data frame containing failure numbers. The expected return is a data frame containing three columns: "Failure", "Time" and "Mode". Shown below is an example template taken from the Musa-Okumoto model, derived from Equation (1) in Section 4.

```
MO_MVF_inv <- function(params, inputData) {
  # Params is a data frame named according to model_specifications.R

  # Define a parameter as calculated by the MLE function
  a <- params$MO_aMLE
  b <- params$MO_bMLE # Define b parameter

  # Row vector of failure numbers taken from input data
  fn <- inputData$FN
  n <- length(fn) # Number of rows in the input

  # Calculate the inverse mean value function based on the
     parameters and input failure number data
  IMVF <- (-1 + exp(a * fn))/(b * fn)

  # IMVF is a vector of failure times for each input failure count
  solution <- data.frame(``Failure'' = fn, ``Time'' = IMVF,
     ``Model'' = rep(``MO'', n))
  return(solution)
}
```

5.5. Mean Time to Failure Function

The mean time to failure calculates an estimate for the number of failures that occur during a given interval. For a NHPP, it is equal to 1 divided

by the derivative of the MVF with respect to time.The function is named <Model_name>_MTTF. The inputs are the MLE parameters data frame, and a two column data frame containing a row vector of failure times, and a row vector of interfailure times. The expected return is a data frame containing three columns: "Failure_Number", "MTTF", and "Model". Shown below is an example template taken from the Musa-Okumoto model, based on Equation (3) in Section 4.

```
MO_MTTF <- function(params, inputData) {
  # Params is a data frame named according to model_specifications.R

  # Define a parameter as calculated by the MLE function
  a <- params$MO_aMLE
  b <- params$MO_bMLE # Define b parameter

  # Row vector of failure times taken from input data
  t <- inputData$FT
  n <- length(t) # Number of rows in the input

  # Calculate the mean time to failure based on the parameters and
     input failure time data
  MTTF <- (a * b * t + 1)/b

  # MTTF is vector of mean time to failure for each input time
  solution <- data.frame(``Failure_Number'' = c(1:n), ``MTTF'' =
     MTTF, ``Model'' = rep(``MO'', n))
  return(solution)
}
```

5.6. Failure Intensity Function

The failure intensity function calculates the failure intensity at each failure time. For a NHPP, it is equal to the derivative of the MVF with respect to time. The function is named <Model_name>_FI. The inputs are the MLE parameters data frame, and a two column data frame containing a row vector of failure times, and a row vector of interfailure times. The expected return is a data frame containing three columns: "Failure_Count", "Failure_Rate", and "Model". Shown below is an example template taken from the Musa-Okumoto model, based on Equation (2) in Section 4.

```
MO_FI <- function(params, inputData) {
  # Params is a data frame named according to model_specifications.R

  # Define a parameter as calculated by the MLE function
  a <- params$MO_aMLE
  b <- params$MO_bMLE # Define b parameter

  # Row vector of failure times taken from input data
  t <- inputData$FT
  n <- length(t) # Number of rows in the input

  # Calculate the failure intensity based on the parameters and
     input failure time data
  failIntensity <- b / (a * b * t + 1)

  # failIntensity is the calculated vector of mean time to failure
     for each input time
  solution <- data.frame(``Failure_Number'' = c(1:n), ``Time'' =
     failIntensity, ``Model'' = rep(``GO'', n))
  return(solution)
}
```

5.7. Continuous Mean Value Function

The continuous mean value function is the same as the mean value function, but it is calculated at a specific point in time instead of with a vector of discrete values. For most models, the code used for the can be copied directly from the MVF, ensuring that it is modified to accommodate a singular float instead of a vector. The function is named <Model_name>_MVF_cont. Inputs to the function are the MLE parameter data frame, and a float representing a singular failure time. The expected return is a floating point number of failure counts predicted at the input time. Shown below is an example template taken from the Musa-Okumoto model, based on Equation (1) in Section 4.

```
MO_MVF_cont <- function(params, failureTime) {
  # Params is a data frame named according to model_specifications.R

  # Define a parameter as calculated by the MLE function
```

```
  a <- params$MO_aMLE
  b <- params$MO_bMLE # Define b parameter

  # failureTime is a float representing the time to calculate
      number of failures at

  # Calculate the mean value function based on the parameters and
      input failure time data
  MVF <- log(b * a * failureTime + 1)/a
  # MVF is mean number of failures at the input time
  return(MVF)
}
```

5.8. Change in Reliability Over Time

The Change in Reliability Over Time function calculates the change in reliability given a time period. It is equal to the exponential of the negative mean value function evaluated at time current time plus delta, plus the mean value function evaluated at the current time. This function is called by the Root Finding Equation, Time to Target Reliability function, and the Reliability Growth function. The function is named <Model_name>_R_delta. Inputs are the data frame containing MLE parameters, a floating point value representing the "current" time, and a floating point value delta representing the change in time. The expected return is a single floating point value representing the change in reliability between the current time, and the interval created by delta. Shown below is an example template taken from the Musa-Okumoto model, based on Equation (8) in Section 4.

```
MO_R_delta <- function(params, cur_time, delta) {
  # cur_time is a float representing the ``current'' time, or
      starting point
  # delta represents the time interval to measure change in
      reliability over
  solution <- exp(-(MO_MVF_cont(params, (cur_time + delta)) +
      MO_MVF_cont(params, cur_time)))
  return(solution)
}
```

5.9. Reliability Growth Function

The reliability growth function calculates the reliability growth curve as a function of time. This function relies on the Change in Reliability Over Time function. The function is named <Model_name>_R_growth. The inputs are the MLE parameters data frame, a two column data frame containing a row vector of failure times, and a row vector of interfailure times, and a floating point value delta. The expected return is a data frame containing three columns: "Time", "Reliability_Growth", and "Model". Shown below is an example template taken from the Musa-Okumoto model.

```
MO_R_growth <- function(params, inputData, delta){

  # Params is a data frame named according to model_specifications.R

  # Row vector of failure times taken from input data
  t <- inputData$FT
  n <- length(t) # Number of rows in the input

  # Declare solution in advance
  solution <- data.frame()

  # Compute change in reliability at every point. If it produces an
      error at any point, replace the value with NA
  for(i in 1:n){
    solution[i, 1] <- t[i]
    solution[i, 3] <- ``MO''
    temp <- MO_R_delta(params, t[i], delta)

    if(typeof(temp) != typeof(``character'')){
      solution[i,2] <- temp
    }
    else{
      solution[i,2] <- ``NA''
    }
  }

  names(solution) <- c(``Time'', ``Reliability_Growth'', ``Model'')
  return(solution)
}
```

5.10. Root Finding Equation

The Root Finding Equation is used by the Time to Target Reliability function. The function is named <Model_name>_R_MLE_root. It relies upon the Change in Reliability function. Inputs are the data frame containing MLE parameters, a floating point value representing the "current" time, a floating point value delta representing the change in time, and a floating point value representing the target reliability. The expected return is a single floating point value representing the difference in the input reliability and the calculated change in reliability for the given interval. The function is used as the objective function in the optimization problem detailed as part of Time to Target Reliability. Shown below is an example template taken from the Musa-Okumoto model.

```
MO_R_MLE_root <- function(params, cur_time,delta, reliability){
  root_equation <- reliability - MO_R_delta(params, cur_time, delta)
  return(root_equation)
}
```

5.11. Time to Target Reliability

The Time to Target Reliability function calculates an estimate for the amount of time required to achieve a target reliability. It is an optimization problem based off the root equation and the change in reliability over time.The function is named <Model_name>_Target_T. The inputs are the MLE parameters data frame, and floating point values representing the "current" time, the change in time delta, and the reliability target. The expected return is either a floating point value representing how much time it will take to reach the desired reliability, or a string stating the target reliability value has already been reached. Shown below is an example template taken from the Musa-Okumoto model. For most cases, this code can be duplicated, substituting in the appropriate change in reliability and root equations.

```
MO_Target_T <- function(params, cur_time, delta, reliability) {
  # Params is a data frame named according to model_specifications.R

  # Wrap root equation for use with optimization function
  f <- function(t){
    return(MO_R_MLE_root(params,t,delta, reliability))
  }
```

```
# Calculate the current reliability
current_rel <- MO_R_delta(params,cur_time,delta)
# If current reliability is less than target, calculate how long
    to get to target, else no work to do here
if(current_rel < reliability){
  # Bound the estimation interval
  sol <- 0
  interval_left <- cur_time
  interval_right <- 2*interval_left
  # Find reliability of bounded interval
  local_rel <- MO_R_delta(params,interval_right,delta)

  # Expand bounds if necessary
  while (local_rel <= reliability) {
    interval_right <- 2*interval_right
    if(local_rel == reliability) {
      interval_right <- 2.25*interval_right
    }
    # Break if bound is infinite
    if (is.infinite(interval_right)) {
      break
    }
    local_rel <- MO_R_delta(params,interval_right,delta)
  }
  # Close lower bound towards upper bound
  if(is.finite(interval_right) && is.finite(local_rel) &&
      (local_rel < 1)) {
    while (MO_R_delta(params,(interval_left +
        (interval_right-interval_left)/2),delta) < reliability) {
      interval_left <- interval_left +
          (interval_right-interval_left)/2
    }
  } else {
    sol <- Inf
  }
  # Solve the optimization problem around the bounded root
  if (is.finite(interval_right) && is.finite(sol)) {
    sol <- tryCatch(
      stats::uniroot(f, c(interval_left,
```

```
            interval_right),extendInt=``yes'', maxiter=maxiter,
            tol=1e-10)$root,
          warning = function(w){
            if(length(grep(``_NOT_ converged'',w[1]))>0){
              maxiter <<- floor(maxiter*1.5)
              MO_Target_T(params,cur_time,delta, reliability)
            }
          },
          error = function(e){
          })
      } else {
        sol <- Inf
      }
    } else {
      sol <- ``Target reliability already achieved''
    }
    return(sol)
}
```

5.12. Optimal Release Time Based on Cost

The Optimal Release Time Based on Cost function calculates an estimate for the optimal time to release software based on the cost. The inputs are the MLE parameters data frame, and three floating point values representing the cost of removing a fault during testing, the cost of removing a fault after release, and the cost of testing per unit time. The expected return is a floating point value representing the optimal release time. Shown below is an example template taken from the Musa-Okumoto model, based on Equation (10) in Section 4.

```
MO_OR_CC<-function(params, c1, c2, c3){
  # Params is a data frame named according to model_specifications.R

  # Define a parameter as calculated by the MLE function
  a <- params$MO_aMLE
  b <- params$MO_bMLE # Define b parameter

  # c1 is the cost of removing a fault during testing

  # c2 is the cost of removing a fault after release
```

```
  # c3 is the cost of testing per time unit

  # Calculate optimal release time as a factor of parameters and
      input costs
  t_opt <- (b * (c2 - c1) - c3)/(a * b * c3)

  return(t_opt)
}
```

5.13. Cost at Software Lifecycle Time

The Cost at Software Lifecycle Time function calculates an estimate for the total cost of the software for its entire lifecycle. The inputs are the MLE parameters data frame, and five floating point values representing the cost of removing a fault during testing, the cost of removing a fault after release, and the cost of testing per unit time, the testing time, and the time at which the software will have reached the end of its lifecycle. The expected return is a floating point value representing the software cost. Shown below is an example template taken from the Musa-Okumoto model, based on Equation (9) in Section 4.

```
MO_cost <- function(params, c1, c2, c3, t, t_lifecycle){
  # Params is a data frame named according to model_specifications.R

  # Define a parameter as calculated by the MLE function
  a <- params$MO_aMLE
  b <- params$MO_bMLE # Define b parameter

  # c1 is the cost of removing a fault during testing

  # c2 is the cost of removing a fault after release

  # c3 is the cost of testing per time unit

  # t is testing time

  # t_lifecyle is software lifecyle time
```

```
  # Calculate optimal release time as a factor of parameters, input
     costs, and software lifecycle time
  cost <- 1/a((c1 - c2) * log(1 + a * b * t) + c2 * log(1 + a * b *
     t_lifecycle)) + a * c3 * t
  return(cost)
}
```

5.14. Optimal Release Time Based on Reliability

The Optimal Release Time Based on Reliability function calculates an estimate for the optimal time to release software based on the reliability of the software. The inputs are the MLE parameters data frame, and two floating point values representing testing time and the estimated reliability of the software based on the model in question. The expected return is a floating point value representing the optimal release time. Shown below is an example template taken from the Musa-Okumoto model, based on Equation (11) in Section 4.

```
MO_OR_RC<-function(params, t, R){
  # Params is a data frame named according to model_specifications.R

  # Define a parameter as calculated by the MLE function
  a <- params$MO_aMLE
  b <- params$MO_bMLE # Define b parameter

  # R is reliability

  # t is testing time

  # Calculate optimal release time as a factor of parameters and
     reliability
  opt_t <- -1/b * (log(log(1 + a * b * t)/a) - log(log(R)))
  return (opt_t)
}
```

6. Debugging and Testing a Model

After the files have been created, the next step is to verify the model's functionality. The best way to do so is to run the model with test data. If the

model has been tested with some other data set before, it is best to use that; otherwise SFRAT includes a number of sample data sets in the model_testing directory, in the files model_data.csv and model_data.xlsx. This guide uses the SYS1 data set from Lyu's Handbook for Software Reliability to verify the Musa-Okumoto model.

The first step is to select the desired data set in the file browser on the first tab of SFRAT. Switch to the "Set up and Apply Models" tab, and select only the model under test in the dialog box labelled "Choose one or more models to run, or exclude one or more models". Click the "Run Selected Models" button. Figure 3 shows the results for the Musa-Okumoto model.

The Musa-Okumoto model is a reasonable fit to the data, implying the model was implemented correctly. This can be further tested by using additional data sets to verify the model fits most cases. A model verified in this way is probably correct, however the only way to truly prove that is to compare against a known correct source.

The next step is to verify that all SFRAT functionalities work for a given model. In the second tab, cycle the drop down menu labelled "Choose a plot type" through all of its options. Monitor the console output of the IDE being used to run SFRAT; if an error message occurs, check if it lists the function where it happened. Otherwise, use browser() statements to step through functions one at a time and find the error. A good first step is to review every function described in 5 and ensure parameter inputs and returned values match exactly the formats described. Something as simple as mislabelling a data frame column could create unexpected results.

Selecting the "Query Model Results" or "Evaluate Models" tabs automatically generates a table. Figure 4 shows the table generated in the "Query Model Results" tab. Figure 5 shows the table generated in the "Evaluate Models" tab. If every column of this table has a value, then each function involved in generating it is syntactically correct. If values are missing, a contributed function must have an error. Every time changes are made to the contributed code, SFRAT must be relaunched, a data set must be selected, and the model must be run on the "Set up and Apply Models" tab.

7. Submitting a Model

Once a model has been implemented and tested, it may be submitted to be included as a part of SFRAT. Submissions are handled in the form of pull requests to the Github repository containing SFRAT, located

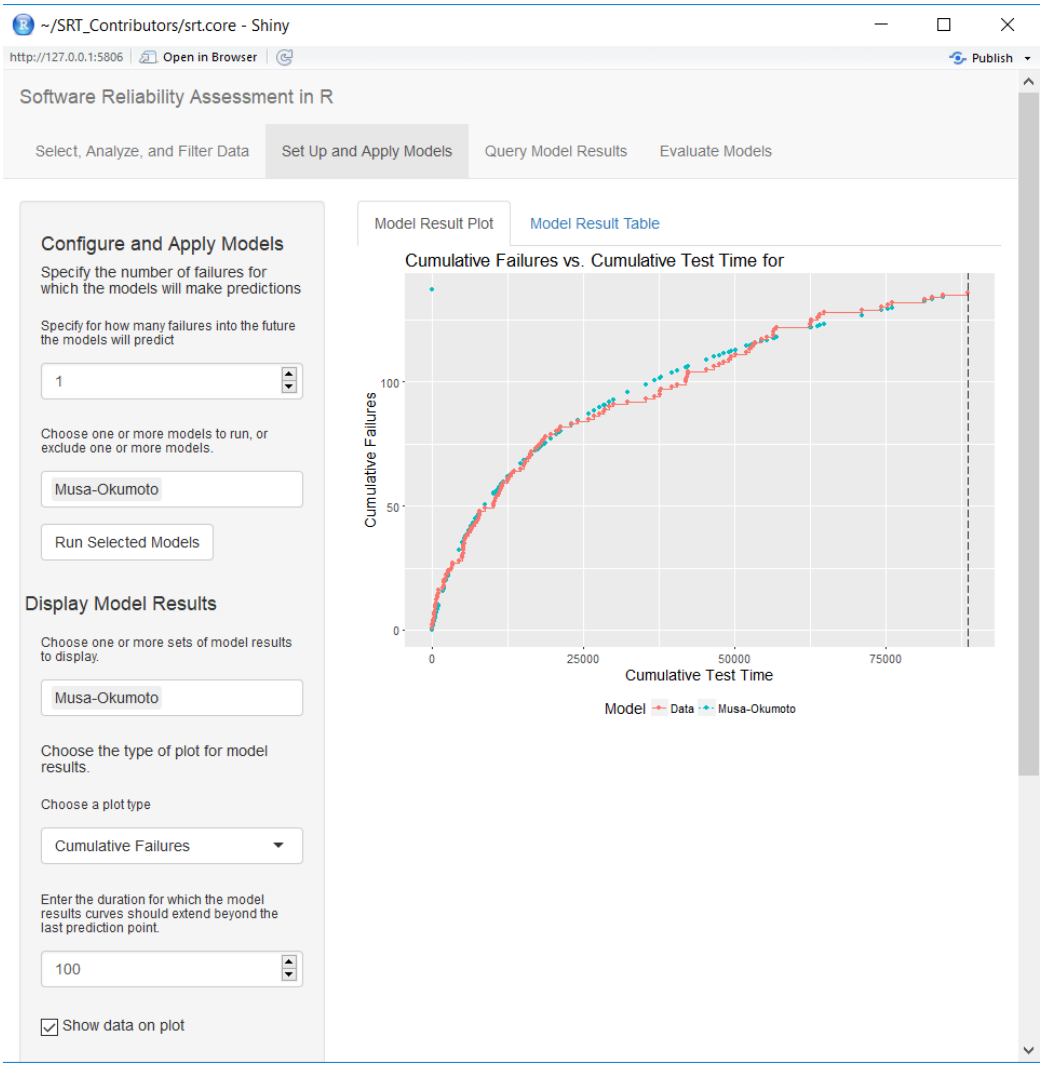

Figure 3: Musa-Okumoto model results plot

at https://github.com/LanceFiondella/srt.core. Contributors familiar with Github may submit a pull request using any method desired. Otherwise, the process of making a pull request through the webpage is described below.

Figure 6 shows the toolbar for submitting a new pull request. Ensure that the dropdown menu has the development branch selected, and press the "Fork" button in the top right. From the forked repository, change the branch dropdown to develop and click the "Upload new files" in the bottom

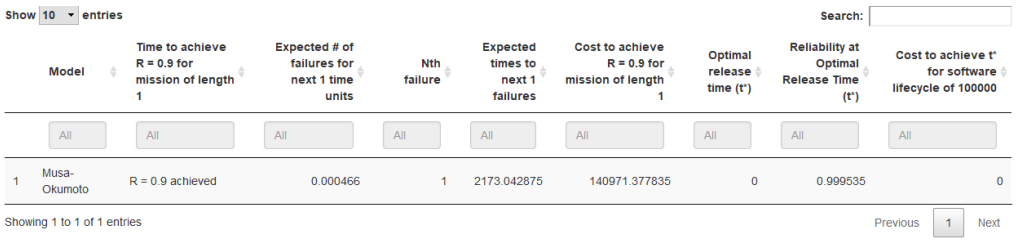

Figure 4: Musa-Okumoto model prediction table

Figure 5: Musa-Okumoto model evaluation table

right. Upload the files in your /models/<Model_Name>/ directory, and the install_script.R file if it was changed. Add an appropriate commit comment, and click "Commit changes". Finally, click "New pull request" on the bottom right of the original toolbar.

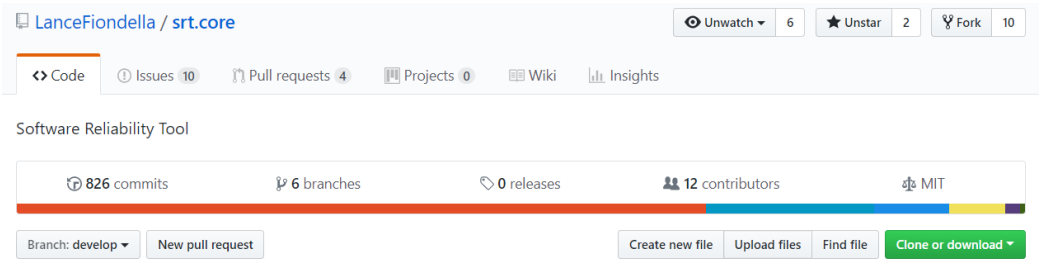

Figure 6: Forking and Uploading Models

Ensure that the toolbar at the top of the pull request lists the base fork as LanceFiondella/srt.core on the develop branch, from the head fork of <Github Username>/srt.core on the develop branch. The pull request will go through automated integration testing, and eventually be reviewed by a member of the SFRAT team and merged into a release.

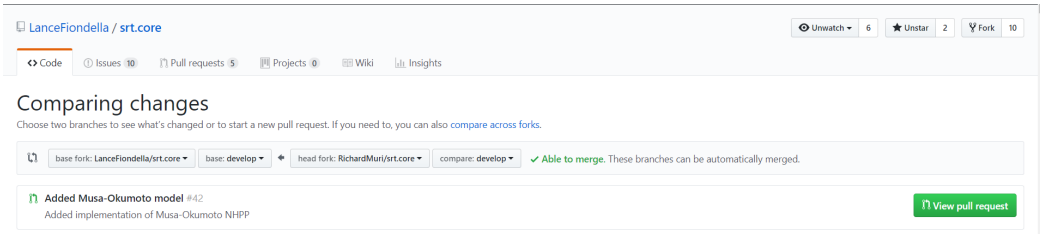

Figure 7: Submitting a Pull Request

Farr, W.H., Smith, O.D., 1993. Statistical modeling and estimation of reliability functions for software (smerfs) users guide. Naval Surface Warfare Center, Virginia .

Lyu, M.R., Nikora, A., 1992. Casre: a computer-aided software reliability estimation tool, in: Computer-Aided Software Engineering, 1992. Proceedings., Fifth International Workshop on, IEEE. pp. 264–275.

Lyu, M.R., et al., 1996. Handbook of software reliability engineering .

Musa, J.D., Okumoto, K., 1984. A logarithmic poisson execution time model for software reliability measurement, in: Proceedings of the 7th international conference on Software engineering, IEEE Press. pp. 230–238.

Nagaraju, V., Fiondella, L., Zeephongsekul, P., Jayasinghe, C.L., Wandji, T., 2017. Performance optimized expectation conditional maximization algorithms for nonhomogeneous poisson process software reliability models. IEEE Transactions on Reliability 66, 722–734.

Okamura, H., Dohi, T., 2013. Srats: Software reliability assessment tool on spreadsheet (experience report), in: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, IEEE. pp. 100–107.