一般社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

The Software Failure and Reliability Assessment Tool (SFRAT)

－A Platform to Foster Collaboration－

Vidhyashree NAGARAJU† Venkateswaran SHEKAR† Thierry WANDJI‡ and Lance FIONDELLA†

†University of Massachusetts Dartmouth, North Dartmouth, MA, USA
‡Naval Air Systems Command, Patuxent River, MD, USA
E-mail: †{vnagaraju,vshekar,lfiondella}@umassd.edu, ‡ketchiozo.wandji@navy.mil

Abstract This paper presents the application architecture of 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. The source code is accessible via a GitHub repository. The tool is presently being used by the United States Department of Defense to assess the reliability of software in Major Defense Acquisition Programs. 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

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, commonly defined as [1] “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 [2,3,4] have implemented tools to automatically apply software reliability growth models (SRGM) that fit software failure data to estimate the number of remaining faults, reliability, and make other useful inferences. 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) [2] which was incorporated into CASRE (Computer-Aided Software Reliability Estimation tool) [3] and SRATS (Software Reliability Assessment Tool on Spreadsheet) [4]. 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 require the user to pay a fee in order for them to be able to use that tool.

To overcome these shortcomings, we have developed the Software Failure and Reliability Assessment Tool (SFRAT), which is both free and open source. A goal 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.

The paper is organized as follows: Section 2 summarizes the current functionality of the tool. Section 3 provides a brief outline of the software architecture and model addition process using the Goel-Okumoto (GO) SRGM [5] as an example as well as the model testing and submission process. Section 4 offers conclusions and directions for future research.

2. Software Architecture

The SFRAT user interface has been implemented in Shiny [6], which is an R web framework that allows the user to access the tool with only a browser such as Internet Explorer or Google Chrome. An instance of the tool is available to the general public for testing at http://www.sasdlc.org. Data sets for use with the web instance, a user guide, and related research are accessible

from http://sasdlc.org/lab/projects/srt.html.

The tool supports Inter-failure Time (IF), Failure Time (FT) and Failure Count (FC) input data formats. Trend tests for reliability growth include the Laplace trend test [7] and running arithmetic average. In addition to these two tests, five models have been implemented which include two hazard rate models [1]: the Jelinski-Moranda (JM) and geometric (GM) as well as three failure count models: the Goel-Okumoto (GO) [5], delayed S-Shaped (DSS) [8], and Weibull (Wei) [9]. The tool includes two goodness of fit measures: the Akaike Information Criterion (AIC) [10] and predictive sum of squares error (PSSE) [11].

Functions implemented in the tool include:

1. Trend tests to verify a data set exhibits reliability growth
2. Predictions:
   - Number of failures that would be discovered with a specified amount of additional testing
   - Time required to achieve a target reliability
   - Time to remove N faults
   - Optimal release time and cost
3. Information theoretic and predictive measures of goodness-of-fit to compare models

3. Model Building and Testing

Figure 1 shows the SFRAT architecture.

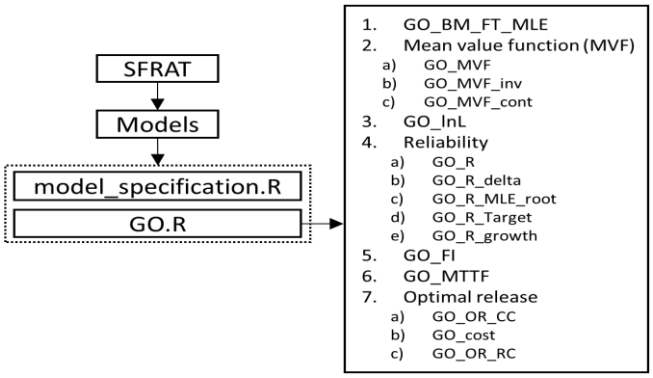

Figure 1: Model Architecture

Code for each model is placed in its own directory within the "models" folder. In this example, we create a folder called "GO" indicated with a dotted line to store the two files related to the Goel-Okumoto model.

3.1. Model Specifications

model_specifcations.R describes the properties of the model being implemented. Every model folder in SFRAT requires a file with this name. Figure 2 shows the content of GO.R.

```
1. GO_input <- c("FT", "FC")
2. GO_methods <- c("BM")
3. GO_params <- c("aMLE","bMLE")
4. GO_numfailsparam <- c(1)
5. GO_fullname <- c("Goel-Okumoto")
6. GO_plotcolor <- c("green")
7. GO_Finite <- TRUE
```

Figure 2: model_specifications.R

The first line of Figure 2 indicates the types of input the model accepts. Here, the GO model accepts FT and FC data. For the purposes of demonstration, only the FT model functions are described here. Line 2 specifies the parameter estimation method, which in this case is the Bisection Method (BM). Additional methods such as Newton's Method (NM) can be included. The SFRAT runs the methods in the order specified and stops after a method converges. Line 3 indicates the model parameters, namely 'aMLE' and 'bMLE'. Line 4 identifies the position of the parameter defined in line 3 that represents the number of failures and stores this value in GO_numfailsparam. Here, aMLE in position 1 is the desired parameter of the GO_params vector. Line 5 stores model's full name in a string. Line 6 defines the color in which to plot this model. Line 7 is a Boolean value that indicates whether the number of failures is assumed to be finite.

3.2. Function Definitions in GO_BM.R

All other files besides the model specifications file should contain the functions described in this section. There are no naming conventions for the filenames besides model_specification.R. Therefore, functions can be arbitrarily divided across multiple files with different names within the folder.

3.2.1. Parameter Estimation (MLE function)

The MLE function computes the maximum likelihood estimates of the parameters of a model. Figure 3 shows the definition of the MLE function for the GO model.

```
GO_BM_FT_MLE <- function(tVec){
GO_params <- data.frame(
"GO_aMLE"=aMLE,
"GO_bMLE"=bMLE)
return(GO_params)
}
```

Figure 3: Parameter Estimation

The name of the function depends on the model name, estimation method, and type of input data and is formed by combining keywords defined in the model_specification.R

file using underscores. The general naming convention is *(Model Name)_(Method)_(Data Type)_MLE*. Thus, the function name for the Goel-Okumoto model implementing the Bisection Method for failure time input data is GO_BM_FT_MLE which accepts input vector *(tVec)*. This function must return a data frame with column names corresponding to the parameters with the format *(Model Name)_(Parameter Name)*. Figure 3 indicates a data frame containing columns GO_aMLE and GO_bMLE is returned.

3.2.2. Mean Value Function (MVF)

Figure 4 shows the format of the mean value function.

```
GO_MVF <- function(param, d) {
  n <- length(d$FT)
  r <- data.frame()
  fail_number <- c(1:n)
MVF<-param$GO_aMLE*(1-exp(-param$G
O_bMLE*d$FT))
  r <- data.frame(MVF,d$FT,rep("GO", n))
  names(r) <- c("Failure","Time","Model")
  return(r)
}
```

Figure 4: Mean Value Function

Similar to the convention described in Section 3.2.1, this function is named *(Model Name)_MVF*. The first argument to the function must be a data frame containing the parameter estimates *(param)* and the second argument a data frame containing the input data *(d)*. The function should return a data frame containing a list of MVF values evaluated at the failures times, the failure times, and model name, which are assigned with names “Failure”, “Time”, and “Model” respectively.

3.2.2.1. Inverse of Mean Value Function

This function calculates the inverse of the MVF. In other words, this function computes a time given a number of faults. Figure 5 shows the GO implementation of this function. The general naming convention is *(Model Name)_MVF_inv*. This function accepts the same inputs as the MVF function and outputs a data frame with columns “Failure”, “Time”, and “Model” corresponding to a vector of integral failures ($1 \leq i \leq n$), the failure times obtained from the inverse of the MVF, and the model name.

```
GO_MVF_inv <- function(param, d) {
      n <- length(d$FN)
      r <- data.frame()
cumFailTimes<--(log((param$GO_aMLE-d$FN
)/param$GO_aMLE))/param$GO_bMLE
r<-data.frame(d$FN,cumFailTimes,rep("GO",
n))
names(r) <- c("Failure","Time","Model")
return(r)
 }
```

Figure 5: Inverse mean value function

3.2.2.2. Continuous Mean Value Function

Figure 6 shows the continuous form of the MVF.

```
GO_MVF_cont <- function(param,x) {
cmvf<-params$GO_aMLE*
(1-exp(-params$GO_bMLE*x))
return(cmvf)
}
```

Figure 6: Continuous mean value function

Unlike the MVF function which returns a discrete set of values, this function returns the MVF evaluated with the MLEs *(param)* at a specified point in time (x). The naming convention for this function is *(Model Name)_MVF_cont*. The function returns a floating-point value of the number of faults predicted by the MVF.

3.2.3. Log Likelihood Function

Figure 7 shows the function to calculate the log likelihood of the model.

```
GO_lnL <- function(params,x) {
n <- length(x)
  tn <- x[n]
  firstSumTerm <- 0
  for(i in 1:n){
   firstSumTerm<-
firstSumTerm+(-params$GO_bMLE*x[i])
  }
lnL<--(params$GO_aMLE)*(1-exp(-params$GO_b
MLE*tn))+          n*(log(params$GO_aMLE))
+n*log(params$GO_bMLE) + firstSumTerm
  return(lnL)
}
```

Figure 7: Log-likelihood function

The function is named *(Model Name)_lnL* and its inputs are the MLEs *(params)* and the input failure data frame *(x)*. It returns the log likelihood as a floating-point number.

3.2.4. Reliability growth

Figure 8 shows reliability function of the GO SRGM,

```
GO_R <- function(params,d){
  n <- length(d$FT)
  r <-data.frame()
  cumulr <-data.frame()
  for(i in 1:n){
    r[i,1] <- d$FT[i]
    r[i,2] <- exp(-params$GO_bMLE*d$FT[i])
    r[i,3] <- "GO"
  }
  r <- data.frame(r[1],r[2],r[3])
  names(r) <- c("Time","Reliability","Model")
  r
}
```

Figure 8: Reliability of GO SRGM

where r[i,1] is the data format, r[i,2] the reliability equation, and r[i,3] the model name to retrieve the MLEs to plot the results.

3.2.4.1. Change in reliability with time

This function computes the change in reliability for a specified time delta. The naming convention is *(Model Name)_R_delta*.

```
GO_R_delta<-function(params,cur_time,
delta) {
  d <-
  exp(-(GO_MVF_cont(params,(cur_time+
delta)) -GO_MVF_cont(params,cur_time)))
  return(d)
  }
```

Figure 9: Reliability with mission time delta

Inputs include the model parameter estimates (*params*), time from which to calculate reliability (*cur_time*) (typically the end of testing), and mission time (*delta*). The function should return the reliability for the interval from *cur_time* to *cur_time+delta*.

3.2.4.2. Root finding utility

This function computes the time to achieve a specified target reliability given mission time. The function naming convention is *(Model Name)_R_MLE_root*.

```
GO_R_MLE_root<-
function(params,cur_time,delta, reliability){
    root_equation<-reliability -
exp(params$GO_aMLE*(1-exp(-params$GO_b
MLE*cur_time))
-params$GO_aMLE*(1-exp(-params$GO_bMLE
*(cur_time+delta))))
    return(root_equation)
  }
```

Figure 10: Root finding utility

The arguments supplied to the function are the estimate of the model parameters (*params*), time at which the reliability is to be calculated (*cur_time*), mission time delta (*delta*) and the target reliability (*reliability*).

3.2.4.3. Time to target reliability

This function uses results from the previously defined function to calculate the time required to achieve a target reliability. Figure 11 provides an outline of this function, but omits implementation details.

```
GO_Target_T<-function(params,cur_time,delt
a,reliability){current_rel<-
GO_R_delta(params,cur_time,delta)
  if(current_rel < reliability){
        sol<-#Code for finding target time
  } else {
        sol <- “Target reliability achieved”
  }
  return(sol)
  }
```

Figure 11: Time to target reliability

The function returns a numeric value or string depending on the current reliability. If the current reliability is less than the target reliability, calculations are performed to determine the time required to achieve the desired reliability. Otherwise, the string “Target reliability is achieved” is returned. The function naming convention is *(Model name)_Target_T* and the inputs include the **model parameters** estimates *(params)*, current time *(cur_time)*, mission time *(delta)* and target reliability *(reliability)*.

3.2.4.4. Reliability growth

Figure 12 shows the function to plot the reliability growth of the GO model utilizing the function GO_R_delta.

```
GO_R_growth <- function(params,d,delta){
  r <-data.frame()
  for(i in 1:length(d$FT)){
    r[i,1] <- d$FT[i]
    temp <- GO_R_delta(params,d$FT[i],delta)
    if(typeof(temp) != typeof("character")){
      r[i,2] <- temp
      r[i,3] <- "GO"
    }
    Else
{
      r[i,2] <- "NA"
      r[i,3] <- "GO"
    }
  }
 g<-data.frame(r[1],r[2],r[3])
names(g)<-c("Time","ReliabilityGrowth","Model")
g
}
```

Figure 12: Reliability growth for specified mission time

The reliability growth function 'temp' utilizes the GO_R_delta function of Figure 9 to compute the reliability growth curve for test data 'd'.

3.2.5. Failure Intensity (FI)

The FI function computes the failure intensity function according to the model MLEs at each failure time. The naming convention is *(Model Name)_FI*.

```
GO_FI <- function(param,d) {
n <- length(d$FT)
r <- data.frame()
fail_number <- c(1:n)
failIntensity<-param$GO_aMLE*param$GO_
bMLE*exp(-param$GO_bMLE*d$FT)
r<-data.frame(fail_number,failIntensity,
rep("GO", n))
names(r)<-c("Failure_Count","Failure_Rate",
"Model")
  return(r)
}
```

Figure 13: Failure intensity

The inputs include the MLEs *(param)* and the input failure data frame *(d)*. The output data frame includes the columns "Failure_Count", "Failure_Rate", and "Model", which indicate the failure interval number, the failure intensity in that interval, and model name.

3.2.6. Mean Time To Failure (MTTF)

The MTTF function estimates the number of failures during a given interval according the MLEs. The function naming convention is *(Model Name)_MTTF*.

```
GO_MTTF <- function(param, d) {
    n <- length(d$FT)
  r <-data.frame()
  cumulr <-data.frame()
  for(i in 1:n){
r[i,1] <- i
r[i,2]<-(1/(params$GO_aMLE*params$GO_bML
E*(exp(-params$GO_bMLE*d$FT[i]))))
r[i,3] <- "GO" }
  r <- data.frame(r[1],r[2],r[3])
names(r)<-c("Failure_Number","MTTF","Model)
  return(r)
 }
```

Figure 14: Mean time to failure

The inputs of the function include the MLEs *(param)* and the input failure data frame *(d)*. The function returns a data frame containing the "Failure_Number", "MTTF" and "Model" for the failure interval number, model estimate of mean time to failure, and model name respectively.

3.2.7. Optimal release time

Figure 15 shows the optimal release time function routine, which requires cost parameters c1, c2, and c3 for the cost of fault removal during testing, operation, and cost per unit time respectively.

```
GO_OR_CC<-function(params,c1,c2,c3){
if(params$GO_aMLE*params$GO_bMLE<=(
c3/(c2-c1))){ t_opt=0
  }
else{  t_opt=(1/params$GO_bMLE)*log((para
ms$GO_aMLE*params$GO_bMLE*(c2-c1))/c3)
  }
  return(t_opt)
}
```

Figure 15: Optimal release time

t_opt represents the optimal release time of the GO model.

3.2.7.1. Cost at specified software lifecycle time

Figure 16 shows the function to compute the cost of releasing the software at time *t* with lifecycle duration (t_lifecycle).

```
GO_cost <- function(params, C1,C2,C3,t, t_lifecycle){
    return(C1*GO_MVF_cont(params,t) + C2*(GO_MVF_cont(params,t_lifecycle) - GO_MVF_cont(params,t)) + C3*t)
}
```

Figure 16: Optimal release cost

If t is the optimal time computed from Figure 15, then this function computes the cost at the time of release.

3.2.7.2. Optimal time to achieve target reliability

Figure 17 shows the function to compute the optimal release time given a specific reliability target,

```
GO_OR_RC<-function(params,x,R){ s=(1/params$GO_bMLE)*((log(params$GO_aMLE*(1-exp(-params$GO_bMLE*x))))-log(log(1/R)))
    return (s)
}
```

Figure 17: Optimal release time to achieve specific reliability

where R denotes the user specified reliability and x the mission time.

3.3. Model Submission

A model can be submitted to SFRAT through GitHub, which applies model validation to provide the contributor with feedback. The contributor should first use their GitHub account to fork the SFRAT repository to their account. Changes to this copy will not affect the original repository. After implementing a model, integrate and test the stability of code in the local copy and push it to the forked repository once debugging is complete.

A pull request must be made to integrate a new model into the main SFRAT repository. This request must include an overview of the model and proposed changes to the tool. The maintainers of the SFRAT repository review the additions. Accepted additions are merged with the main repository. The integration service Travis-CI performs basic unit testing on the models to confirm proper naming conventions have been followed and the corresponding input and output specifications are valid to ensure functions behave as expected. If any test fails, the Travis-CI build log contains error statements. To remove errors before submission, contributors can run these tests on their local copy of the tool by executing the 'run_tests.R' script provided in the root directory.

4. Conclusion and Future Research

The SFRAT tool has been designed to incorporate a wide variety of software reliability models. It is anticipated that the tool architecture and corresponding work flow will grow to include additional inferences possible from software reliability models. The open source nature of the tool and extensible architecture enable a platform for collaboration.

Future research will seek to further standardize the SFRAT architecture and incorporate functionality sought by the user community.

Acknowledgments

This work was supported by (i) the Naval Air Warfare Center (NAVAIR) under contract N00421-16-T-0373 and (ii) the National Science Foundation (#1526128).

Literature

[1] M. Lyu, Ed., *Handbook of Software Reliability Engineering*. New York, NY: McGraw-Hill, 1996.

[2] W. Farr and O. Smith, "Statistical modeling and estimation of reliability functions for software (SMERFS) users guide," *Naval Surface Warfare Center, Dahlgren, VA, Tech. Rep*. NAVSWC TR-84-373, Rev. 2, 1984.

[3] M. Lyu and A. Nikora, "CASRE: A computer-aided software reliability estimation tool," *IEEE International Workshop on Computer-Aided Software Engineering*, pp. 264–275, 1992.

[4] H. Okamura and T. Dohi, "SRATS: Software reliability assessment tool on spreadsheet (Experience report)," *IEEE International Symposium on Software Reliability Engineering*, pp. 100-107, 2013.

[5] A. Goel, "Software reliability models: Assumptions, limitations, and applicability." *IEEE Transactions on Software Engineering*, no. 12, pp. 1411–1423, 1985.

[6] The Comprehensive R Archive Network, *https://cran.r-project.org/* and Shiny by RStudio, A web application framework for R, "*http://shiny.rstudio.com/*", Last accessed 2016.

[7] H. Ascher and H. Feingold, "Repairable Systems Reliability: Modeling, Inferences, Misconceptions and Their Causes." *Marcel Dekker*, Inc, 1984.

[8] S. Yamada, H. Ohtera, and H. Narihisa, "Software reliability growth models with testing-effort." *IEEE Transactions on Reliability*, vol. R-35, no. 1, pp.19-23, 1986.

[9] S. Yamada and S. Osaki, "Reliability growth models for hardware and software systems based on nonhomogeneous Poisson process: A survey." *Microelectronics and Reliability*, vol. 23, no. 1, pp. 91–112, 1983.

[10] H. Akaike, "A new look at the statistical model identification." *IEEE Transactions on Automatic Control*, vol. 19, no. 6, pp. 716–723, 1974.

[11] L. Fiondella and S. Gokhale, "Software reliability model with bathtub-shaped fault detection rate." *Annual Reliability and Maintainability Symposium*, pp. 1-6, 2011.