An Open Source Software Reliability Tool: A Guide for Contributors

Karthik Katipally, Vidhyashree Nagaraju, *Student Member, IEEE,* Lance Fiondella, *Member, IEEE,*

Abstract

This document provides a detailed description of the procedure involved in integrating additional model to an open source software reliability tool (SRT). The open source software reliability tool automatically apply methods from software reliability engineering, including visualization of failure data, application of software reliability growth models, inferences made possible by these models, and assessment of goodness of fit. The tool provides a flexible architecture to allow reliability researchers or practitioner community to be able to easily integrate their models into the tool.

Index Terms

GitHub, R statistical programming language, software reliability, software reliability growth model

Acronyms

BM	Bisection method
CASRE	Computer-Aided Software Reliability Estimation tool
DSS	Delayed S-shaped
EM	Expectation-maximization algorithm
GM	Geometric model
GO	Goel-Okumoto model
IF	Interfailure times
JM	Jelinski-Moranda
FT	Failure times
FC	Failure counts
MTTF	Mean time ot failure
NHPP	Non-homogeneous Poisson process
NM	Newton's method
SMERFS	Statistical Modeling and Estimation of Reliability Functions for Software
SRGM	Software reliability growth models
SRT	Software reliability tool
Wei	Weibull SRGM

I. Introduction

THE modern society is highly dependent on software critical systems for safety and functionality. Therefore, these software intensive systems must be highly reliable to ensure safety. In other words, software reliability is a key to success of software systems, which is defined as the probability of failure free operation during a specific period of time in a specific environment. Typically software reliability growth models (SRGM) are used to fit the observed data, which enables important metrics such as number of remaining faults, reliability, mean time to failure (MTTF), optimal release time and so on. Software reliability engineering [1] is a well established field, which is over four decades old. In this period, researchers have made significant contribution to the software reliability by developing large number of models and methods in response to the growing needs of the practitioner community. However, research has not always reached it target audience, namely software and reliability engineers. Often, these audiences are reluctant to apply software reliability models because they lack either the knowledge of the underlying mathematics or the time to develop expertise and implement models in their work.

Several automated tools developed for software reliability engineering include: 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]. Although popular among the user community, the primary shortcoming of these previous tools is that they were implemented as desktop applications by a single researcher, which limited the models incorporated to those the individual researcher was familiar with. Another shortcoming was the a need of more polished user interface to allow the user with insufficient software reliability knowledge to access the tool without

K. Katipally, V. Nagaraju, and L. Fiondella are with the Department of Electrical and Computer Engineering, University of Massachusetts, Dartmouth, MA, 02747 USA e-mail: {kkatipally,vnagaraju,lfiondella}@umassd.edu.

Manuscript received XXX; revised XXX.

much difficulty. Also, these tools are dependent on certain operating systems, which may throw some challenges in terms of stability and robustness fo the tool.

To overcome the limitation of previous software reliability tools, an open source software reliability tool (SRT) is presented to promote collaboration among members of the international software reliability research community and users from industry and government organizations. The application is implemented in the R programming language, an open source environment for statistical computing and graphics. It is also accessible through a web-enabled framework web at *sasdlc.org* (System and software development life cycle). The code is accessible through the GitHub repository at *github.com/lfiondella/SRT*. Its architecture will enable incorporation of existing software reliability models into a single tool, enabling more systematic comparison of models than ever before. Presently, interfailure time (IF), failure time (FT), and failure count (FC) data formats are supported. Implemented functionality includes: two trend tests for reliability growth, two failure rate, Jelinski-Moranda (JM) [1] and geometric (GM) [1], and three failure counting models, Goel-Okumoto (GO) [5], [1], delayed S-shaped (DSS) [6], and Weibull (Wei) [7] models as well as two measures of goodness of fit. Inferences such as the time to achieve a target reliability or detect a specified number of additional failures have also been implemented.

Section **??**.

II. Simplified Tool Architecture

The actual Implementation is simplified in the below diagram to let the contributors focus on the model implementation.

```
SYS1
  [1]      3    33   146   227   342   ...
 [13]    836   860   968  1056  1726   ...
 [25]   3098  3278  3288  4434  5034   ...
...      ...   ...   ...   ...   ...   ...
[133] 81542 82702 84566 88682
```

The above is a standard dataset which is a cumulative failure time data_set.

This data-set is in the form of list data structure. When a data-sheet which is formatted according to tool requirements is uploaded then the data takes up the below format.

```
> data <- data.frame("FT"=SYS1)
> generate_dataFrame(data)
       FT   IF  FN
1       3    3   1
2      33   30   2
3     146  113   3
...   ...  ...   ...
```

III. Model Architecture

This section describes how the software reliability growth models can be incorporated into the tool architecture. To ensure modularity, the model architecture has been decoupled from the tool using an interface called model-specifications, which is defined for each model in a separate file named 'model_specifications.R'. This model specifications interface provides the details required for computations whenever a user request is made through the application's graphical user interface. The server then delegates the request to the specific model. Model functionality is defined in a file named (MODEL)_(METHOD)_(TYPE).R. The bounded boxes (STYLE1) here represents the placeholder. First bounding box here represents MODEL. The MODEL could be replaced with possible short name of a model i.e. GO. Likewise the second bounding box represents the METHOD, which could take values like BM, EM etc. The third bounding box here represents the TYPE of the input model can take , can be either FT or FC. Similar bounding box convention with different style STYLE2 is used in the model specification file and the model implementation file. The parameters set in the model specifications file are substituted in the implementation details of the model for computation along with the request object parameters by server on each request.

Two kinds of placeholders for parameters, one for input request object and other for Model specifications parameters. User request object is sent from the client to the shiny end as input object and the parameters such as MODEL. TODO: stands for Model request.

```
SRT
└─models
   ├─model-1
   │  ├─model-1.R
   │  └─model_specifications.R
   ├─model-2
   └─model-3
```

1) Model configuration.
2) Model design.

A. Model Configuration

This section describes the interface file model_specifications.R.

```
# New model
MODEL_input <- c("FT"/"FC"/"IF")
MODEL_methods <- c("BM"/"NM"/....)
MODEL_params <- c("a1","a2",...,"an")
MODEL_type <- c("NHPP","Exp","S-shaped")
MODEL_numfailsparm <- c(Index of failure counts parameter)
MODEL_fullname <- c(string MODEL FULL NAME)
MODEL_plotcolor <- c(string COLOR)
MODEL_Finite <- bool TRUE/FALSE
# MODEL_prefer_method <- c("BM")
```

An example model-specifications file is shown below.

```
# Goel_okumoto model
GO_input <- c("FT")
GO_methods <- c("BM")
GO_params <- c("aMLE","bMLE")
GO_type <- c("NHPP","Exp")
GO_numfailsparm <- c(1)
GO_fullname <- c("Goel-Okumoto")
GO_plotcolor <- c("green")
GO_Finite <- FALSE
# GO_prefer_method <- c("BM")
```

B. Model Design

This section describes the functions that needs to be defined in the MODEL.R file. The check list of functions:

1) Estimating parameters: This section describes how to define a function which will evaluate estimates of the parameters for a given model. The template for this function is:

```
MODEL_METHOD_APPROACH <- function(x){
        ...
        MODEL_params <-  data.frame(
                         a1 =xxx,
                         a2 =xxx,
                         ..... ,
                         an =xxx)
        return( MODEL _params)
}
```

Example: GO - Goel-Okumoto

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

a) Naming convention: The name of the function should be defined as above.Explanation:TODO

b) Input and return types:

2) Mean Value Function: This is a MVF function.

```
MODEL_MVF <- function(MODEL_params,DATA)
{         ...
        MVF_list <- c(...) # Numeric
        MVF <- data.frame(MVF_list,
                 DATA$TYPE,
                 rep(MODEL,n))
        names(MVF) <- c("Failure","Time","Model")
        return(MVF)
}
```

Example: GO - Goel-Okumoto

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

a) Naming convention: MODEL_MVF.R

b) Input and return types:

3) MVF_INV: This section describes the inverse of mean value function definition.

```
(MODEL)_MVF_Inv <- function((MODEL)_params,(DATA))
{         ...
         MVF_list <- c(...) # Numeric
         MVF_INV <- data.frame(MVF_list,
                 (DATA)$(TYPE),
                 rep((MODEL),n))
         names(MVF_INV) <- c("Failure","Time","Model")
         return(MVF_INV)
}
```

Example: GO - Goel-Okumoto

```
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")
  r
}
```

a) Naming convention: (MODEL)_MVF_Inv.R

b) Input and return types: TODO:

4) MVF_MTTF: This section describes the inverse of mean value function definition.

```
(MODEL)_MTTF <- function((MODEL)_params,(DATA))
{         ...
         MTTF_list <- c(...) # Numeric
         MTTF <- data.frame(MVF_list,
                 (DATA)$(TYPE),
                 rep((MODEL),n))
         names(MTTF) <- c("Failure","Time","Model")
         return(MTTF)
}
```

Example: GO - Goel-Okumoto

```
GO_MTTF <- function(param,d) {
  n <- length(d$FT)
  r <- data.frame()
  currentTimes <- utils::tail(d, length(d$FT)-1)
  prevTimes <- utils::head(d, length(d$FT)-1)
  currentFailNums <- c(2:n)
  prevFailNums <- c(1:(n-1))
  IFTimes <- ((currentFailNums*currentTimes$FT)\tabularnewline
              /(param$GO_aMLE*\tabularnewline
              (1-exp(-param$GO_bMLE*currentTimes$FT))) \tabularnewline
              - ((prevFailNums*prevTimes$FT)\tabularnewline
              /(param$GO_aMLE*(1-exp(-param$GO_bMLE*prevTimes$FT))))
  r <- data.frame(c(1, currentFailNums),
                  c(((d$FT[1])/(param$GO_aMLE*(1-exp(-param$GO_bMLE*d$FT[1])))),
                  IFTimes),
                  rep("GO", n))
  names(r) <- c("Failure_Number","MTTF","Model")
  r
}
```

a) Naming convention: (MODEL)_MTTF.R

b) Input and return types: TODO:

5) MVF_FI: This section describes the inverse of mean value function definition.

```
(MODEL)_FI <- function((MODEL)_params,(DATA))
{         \ldots
         MTTF_list <- c(...) # Numeric
         MTTF <- data.frame(MVF_list,
                 (DATA)$(TYPE),
                 rep((MODEL),n))
         names(MTTF) <- c("Failure","Time","Model")
         return(MTTF)
}
```

Example: GO - Goel-Okumoto

```
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")
  r
}
```

a) Naming convention: (MODEL)_FI.R

b) Input and return types: TODO:

6) MVF_lnL: This section describes the inverse of mean value function definition.

```
(MODEL)_lnL <- function((DATA),(MODEL)_params)
{         ...
         lnL <- numeric
         return(lnL)
}
```

Example: GO - Goel-Okumoto

```
GO_lnL <- function(x,params){
  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$GO_bMLE*tn)) \tabularnewline
        + n*(log(params$GO_aMLE)) +n*log(params$GO_bMLE)
        + firstSumTerm
  lnL
}
```

a) Naming convention: (MODEL)_lnL.R

b) Input and return types: TODO:

7) MVF_cont: This section describes the inverse of mean value function definition.

```
% (MODEL)_lnL <- function((DATA),(MODEL)_params)
% {         ...
%          lnL <- numeric
%          return(lnL)
% }
```

Example: GO - Goel-Okumoto

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

a) Naming convention: (MODEL)_lnL.R

b) Input and return types: TODO:

8) R_delta: This section describes the inverse of mean value function definition.

```
% (MODEL)_lnL <- function((DATA),(MODEL)_params)
% {         ...
%          lnL <- numeric
%          return(lnL)
% }
```

Example: GO - Goel-Okumoto

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

a) Naming convention: (MODEL)_R_delta.R

b) Input and return types: TODO:

9) R_Root: This section describes the inverse of mean value function definition.

```
% (MODEL)_lnL <- function((DATA),(MODEL)_params)
% {         ...
%          lnL <- numeric
%          return(lnL)
% }
```

Example: GO - Goel-Okumoto

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

a) Naming convention: (MODEL)_R_Root.R

b) Input and return types: TODO:

10) Target_T: This section describes the inverse of mean value function definition.

```
% (MODEL)_lnL <- function((DATA),(MODEL)_params)
% {        ...
%         lnL <- numeric
%         return(lnL)
% }
```

Example: GO - Goel-Okumoto

```
GO_Target_T <- function(params,cur_time,delta, reliability){

  f <- function(t){
    return(GO_R_MLE_root(params,t,delta, reliability))
  }

  current_rel <- GO_R_delta(params,cur_time,delta)
  if(current_rel < reliability){
    # Bound the estimation interval

    sol <- 0
    interval_left <- cur_time
    interval_right <- 2*interval_left
    local_rel <- GO_R_delta(params,interval_right,delta)
    while (local_rel <= reliability) {
      interval_right <- 2*interval_right
      if(local_rel == reliability) {
        interval_right <- 2.25*interval_right
      }
      if (is.infinite(interval_right)) {
        break
      }
      local_rel <- GO_R_delta(params,interval_right,delta)
    }
    if(is.finite(interval_right) && is.finite(local_rel) && (local_rel < 1)) {
      while (GO_R_delta(params,\tabularnewline
                  (interval_left + \tabularnewline
                   (interval_right-interval_left)/2),\tabularnewline
             delta) < reliability) {
        interval_left <- interval_left + (interval_right-interval_left)/2
      }
    } else {
      sol <- Inf
    }

    if (is.finite(interval_right) && is.finite(sol)) {
      sol <- tryCatch(
        stats::uniroot(f, \tabularnewline
                       c(interval_left, interval_right),\tabularnewline
                       extendInt="yes", \tabularnewline
                       maxiter=maxiter, \tabularnewline
                       tol=1e-10)$root,
        warning = function(w){
          #print(f.lower)
          if(length(grep("_NOT_ converged",w[1]))>0){
            maxiter <<- floor(maxiter*1.5)
            print(paste("recursive", maxiter,sep='_'))
            GO_Target_T(a,b,cur_time,delta, reliability)
          }
        },
        error = function(e){
          print(e)
          #return(e)
```

a) Naming convention: (MODEL)_Target_T.R

b) Input and return types: TODO:

11) R_growth: This section describes the inverse of mean value function definition.

```
% (MODEL)_lnL <- function((DATA),(MODEL)_params)
% {         ...
%          lnL <- numeric
%          return(lnL)
% }
```

Example: GO - Goel-Okumoto

```
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","Reliability_Growth","Model")
  #print(g)
  g
}
```

a) Naming convention: (MODEL)_R_delta.R

b) Input and return types: TODO:

IV. Model Submission

A model can be submitted to SRT through GitHub, which implements a series of model validation phases to provide the contributor with feedback so that they can understand if all of their modules have been accepted. To begin submission the contributor must be a GitHub user. From GitHub perspective the contributor should fork srt-repository. After a successful fork process, the forked repository will be visible within the contributor's profile. This forked repository is your own personal repository and changes made to this repository will not affect the original srt-repository, so a user does not need to worry that experimentation will destabilize the tool. The next step is to clone this forked repository to your local machine. Once the forked repository is cloned to your local machine, you should be able to see the models directory which is similar in structure mentioned in (ref). As a model contributor, any additions or modifications required should be performed within this models directory. To contribute a new model, you will need to create a new folder with name same as your model's short name. Make sure that your model-design file is named with the model shortname and possesses an 'R' extension at the end.

In addition to the model-design file, the tool requires the model-specifications file. These model specifications and model-design files should be within the model (MODEL) folder. After performing these steps, you can submit the model for integration. To submit the model, the changes made must first be pushed to your own repository. This can be accomplished in two ways, namely the Terminal friendly or GUI approach.

1) Terminal approach: Execute the following sequence of commands to push the code. Through the command line, change to the srt clone-directory. Then, check for git status with:

```
git status
```

This command lists the files changed so far. Next, add the changes you made to your local machine repository with:

```
git add MODEL_METHOD_APPROACH.R
git add model_specifications.R
```

To make sure these files are added to your local repository, use the git status command as above.

The next step is to commit the changes made to push them to remote forked-repository with the following command:

```
git commit -am 'Message for easy trace back'
```

Finally, push the code to remote repository.

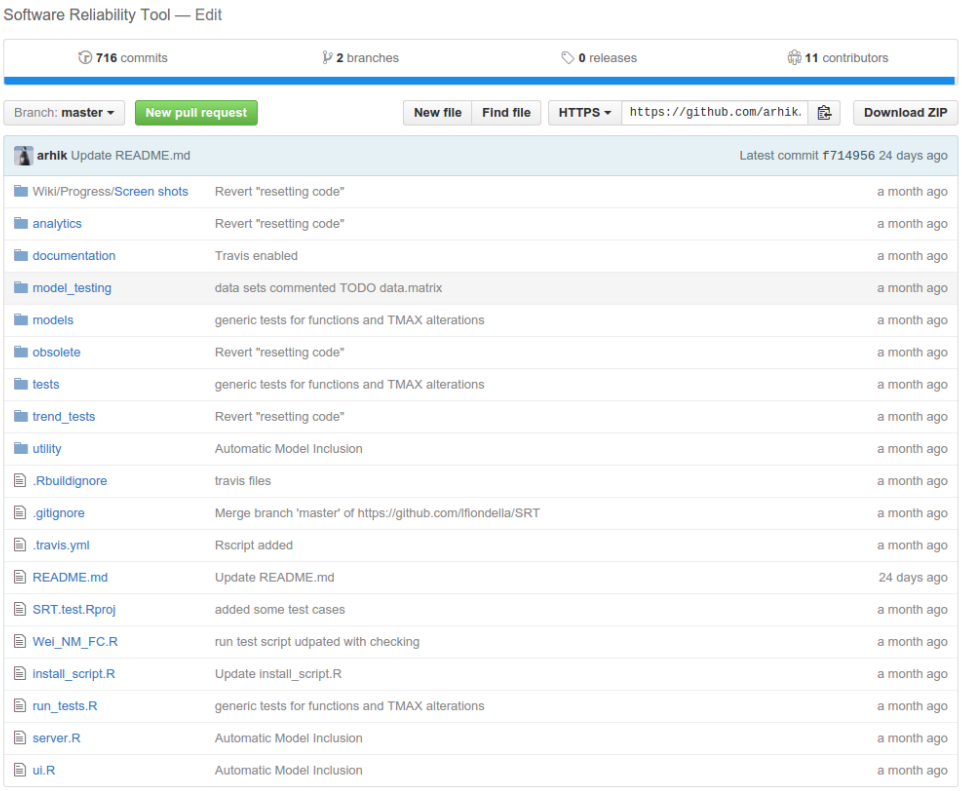
```
git push origin master
TODO: remotes add
```

This pushes your local machine code to your online repository. Be sure to check this operation is reflected in your online forked repository. Refer to the pull-request section to submit your code to the original repository.

2) Web-GUI approach: The new model should be tested against the standard data-sets and the results of the data-sets should be saved in the same folder. This helps other contributors ensure that the tool is working as defined. The GitHub procedure is as follows: TODO.

A. Pull-Request

In the GitHub GUI browse to the forked repository as shown in figure. TODO: Needs different image and correct annotation.



pull request.png

Figure 1: GUI interface

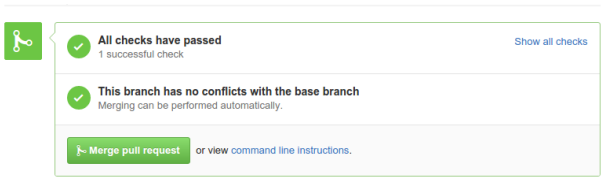
Checklist : TODO

V. Model Testing

After successfully submitting a model, the third party continuous integration service Travis-CI will test the models with basic unit testing and then enters the integration testing phase. Unit testing makes ensures proper naming conventions are followed and that required functions are and proper specifications are defined. Integration testing ensures that the functions defined behave as expected. The return types of each function are tested to verify that they produce the expected data-structures as well as values which are valid for computations to carry on.

Click on the travis reference for additional details.

This redirects you to the build phase of the model. The interface contains the build log of the tool.



link.png

Figure 2: Travis link

Figure 3: Travis Interface

```
2002 ------------------------------------------------------
2003 |      DIRECTORY: ./models/TMAX
2004 ------------------------------------------------------
2005 |sourcing --> ./models/TMAX/Model_specifications.R
2006 |TEST     --> |PASSED
2007 |sourcing --> ./models/TMAX/TMAX_IECM_FT.R
2008
2009
2010 ------------------------------------------------------
2011 |      DIRECTORY: ./models/Wei
2012 ------------------------------------------------------
2013 |sourcing --> ./models/Wei/Model_specifications.R
2014 |TEST     --> |PASSED
2015 |sourcing --> ./models/Wei/Wei_NM_FT.R
2016
2017
2018 ------------------------------------------------------
2019 |      DIRECTORY: ./models/ZZZZ
2020 ------------------------------------------------------
2021 |sourcing --> ./models/ZZZZ/Model_specifications.R
2022 |TEST     --> |PASSED
2023 |sourcing --> ./models/ZZZZ/ZZZZ_QQ_FC.R
2024 |sourcing --> ./models/ZZZZ/ZZZZ_QQ_FT.R
2025
2026
2027
2028 The command "Rscript ./run_tests.R" exited with 0.
2029
2030 Done. Your build exited with 0.
```

Figure 4: Travis Build Log

The build log helps you to debug your model in case testing fails. Figure 4 shows an example build log when the test is passed.

If any of the test fails then the build log will contain FAIL statements which require further inspection in the model design or model specifications to resolve.

VI. Conclusion

The tool is designed to encapsulate a wide variety of software reliability models. It is anticipated that the tool architecture and corresponding work flow will evolve to incorporate more method to asses software reliability throughout the life cycle. The open-source nature of the tool provides a platform for the international software reliability researcher community to incorporate results from past decades as well as to define a research agenda for the future.

Acknowledgment

This work was supported by (i) the Naval Air Systems Command through the Systems Engineering Research Center, a Department of Defense University Affiliated Research Center under Research Task 139 : Software Reliability Modeling and (ii) the National Science Foundation (#1526128).

References

[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," in *IEEE International Workshop on Computer-Aided Software Engineering*, 1992, pp. 264–275.
[4] H. Okamura and T. Dohi, "SRATS: Software reliability assessment tool on spreadsheet (experience report)," in *International Symposium on Software Reliability Engineering*, Nov 2013, pp. 100–107.
[5] A. Goel, "Software reliability models: Assumptions, limitations, and applicability," *IEEE Transactions on Software Engineering*, no. 12, pp. 1411–1423, 1985.
[6] 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, apr 1986.
[7] 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.