An Open Source Software Reliability Tool: A Guide for Contributors

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

I. Introduction

SOFTWARE Reliability Tool (SRT) determines and estimates the reliability of a software by analysing the failure data. The analysis of data is done by different reliability models. The failure data could be of failure rate form or failure count form. The tool should adapt to the format presented to it and run models on the given data. The best fit model is chosen the end user. Software reliability is important for mission critical systems the models should represent the data as closely as possible. The conventional way to achieve more reliable estimates is to run different models on the given data and choose the best based on the goodness of fit measures (GOF). More models more choice and better prediction of software reliability. The tools available now to estimate the software are closed source. This is not only an attempt to design an open-source tool but also to provide framework for fellow researchers to contribute their own.

II. Simplified Tool Architecture

The actual Implementation is simplified in the below diagram to let the contributors focus on the model implementation. You can skip the details but its good to know the implementation details if you have to use an unconventional approach to implement your model.

```
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 the 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 fit the tool architecture. The model architecture is decoupled from the tool using interface called model-specifications. This interface is defined in a seperate file named 'model_specifications.R' for each model . This model specifications interface fills in the additional details required by the model for computation whenever there is a request from a user. The server only plays a role of delegating the request of the user to the specific model. The actual functionality is defined in a seperate file name MODEL_METHOD_TYPE_APPROACH.R. Summarizing the above the explanation:TODO. Programming a model is done in two steps.

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: {kkatipalli,vnagaraju,lfiondella}@umassd.edu.

Manuscript received XXX; revised XXX.

```
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:

- function to calculate the MVF
- function to calculate the MVF_Inv
- function to calculate the MTTF
- function to calculate the FI
- function to calculate the R
- function to calculate the lnL
- function to calculate the Faults remaining
- function to calculate the MVF_cont
- function to calculate the R delta
- function to calculate the R_root
- function to calculate the Time to Target Reliability
- function to calculate the R growth.

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

```
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:

```
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)/(param$GO_aMLE*(1-exp(-param$GO_bMLE*currentTi
  r <- data.frame(c(1, currentFailNums), c(((d$FT[1])/(param$GO_aMLE*(1-exp(-param$GO_bMLE*d$F
  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$GO_bMLE*tn)) + n*(log(params$GO_aMLE)) +n*log(params
  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)) -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)) -params$
  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,(interval_left + (interval_right-interval_left)/2),delta) < rel
        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, c(interval_left, interval_right),extendInt="yes", maxiter=maxiter, t
        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)
        })
    } else {
      sol <- Inf
    }
  } else {
    sol <- "Target reliability already achieved"
  }
```

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

The model designed can be submitted to SRT-tool through github. (This enables for validation of the model). The model submitted will go through a series of validation phases and lets the user know if everything is met or not for it to be accepted. (The tool is hosted on github and contribution is done typically through github). To start with submission the contributor should be a github user. From github perspective the contributor should fork srt-repository. After a successful fork process, the forked repository will be visible on their profile. This forked repository is your own repository and changes made to this repository does not affect the original srt-repository so user need not worry to experiment with the tool. Next step is to clone this forked repository to their local machines. Once forked repository is cloned to their local machines the user should be able to see models directory which is similar in structure mentioned in (ref). As a model contributor any additional or modifications required should be with in this models directory. For a new model contribution the contributors need to create a new folder with name same as the models short name. The contributor should make sure their model-design file is named with the model shortname and with an 'R' extension at the end. Along with the model-design file the tool expects the model-specifications file. These model specifications file and the model-design file should be within the model (MODEL) folder. After making sure that the above steps are done, the contributor can just submit the model for integration. To submit the model the changes made need to be pushed to your own repository in the first place. This can be done in two ways: 1. Terminal friendly approach. 2. GUI approach.

1) Terminal approach: The following commands should be executed for the pushing the code sucessfully. Through command line change to the srt clone-directory. Then check for git status by using command:

```
git status
```

This command lists the files changed so far. Next step is to add the changes made to you local machine repository. This step is done by using commands :

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

To make sure the above files are added to your local repository use the git status command as above. Next step is to commit the changes made to push them to remote forked-repository. This can be done using following commands:

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

Next step is to push the code to remote repository.

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

This pushes your local machine code to your online repository. (Please check if its been reflected on your online forked repository.). You can now refer to 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 is not mandotory phase but it ensures that the tool is working as defined. The github procedure from here: 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

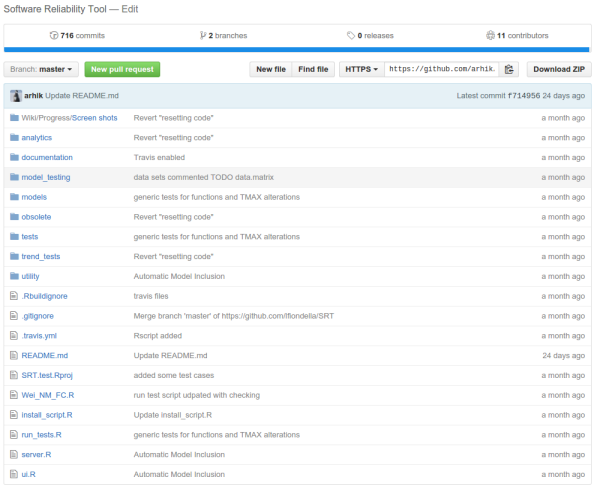

Fig. 1: GUI interface

Checklist : TODO

V. Model Testing

After sucessful submission of the model. The third party continuous integration service named Travis-CI will start to test the models with basic unit testing and then enters the integration testing phase. The unit testing makes sure that proper naming conventions are followed, required functions are defined and proper specifications are defined. The integration tests makes sure that the functions defined work as expected. The return types of each are tested if they are valid in terms of expected data-structure and the values which are valid for computations to carry on.

Click on the travis reference for additional details.

link.png

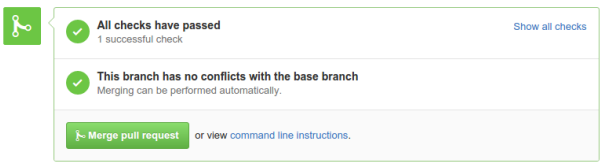

Fig. 2: Travis link

Fig. 3: Travis Interface

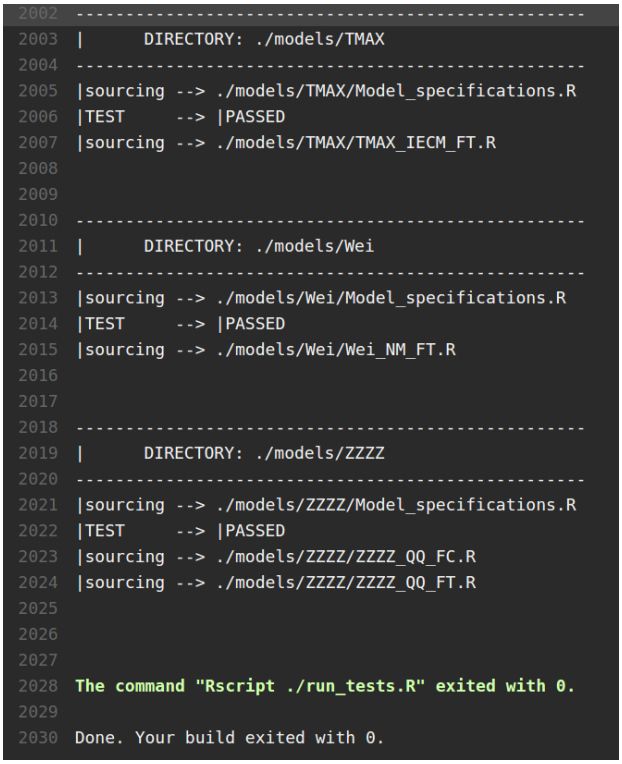

Fig. 4: Travis Build Log

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

The build log helps you to debug your model in case test fails. An example build log when the test passes is shown below.

If any of the test fails then the build log will have FAIL statements which needs further inspection in the model design or model specifications to rectify it.

VI. Conclusion

The tool is designed to encompass wide possibilities of approaches of determining the software reliability. The tool might evolve with the advances in the research of software reliablity. This also creates the platform for fellow researchers to focus on what needs to be done as opposed to what already done.

APPENDIX A
PROOF OF
ACKNOWLEDGMENT

The authors would like to thank...