PharmaSUG 2021 - Paper SI-074

A Strategy to Develop Specification for R Functions in Regulated Clinical Trial Environments

Aiming Yang, Yalin Zhu, Yilong Zhang

Merck & Co., Inc., Kenilworth, NJ, USA

ABSTRACT

A well-written programming requirement specification is an essential document for a regulatory compliant environment and software development life cycle (SDLC). It is a technical document outlining the requirement and usage of a specific computer program. With the growing interest in using R for regulated clinical trial data analysis, there is a need to document specifications of R functions within an R package. The specification generation has direct implications on source code and file management. In this paper, we propose a strategy of using Roxygen to develop the specification of an R function. We will introduce concepts around R functions and Roxygen. We will also conduct demos for the specification generating process and present simple examples. We hope the proposed specification documentation process can help the pharmaceutical industry standardize the SDLC for internal R package development.

INTRODUCTION

In the regulatory statistical programming environment for clinical trials, the specification for statistical programing is a common required element and an essential document for the software development life cycle (SDLC) (https://www.r-project.org/doc/R-FDA.pdf**)**. R is an open-source programming language and software environment for statistical computing and graphics. In the R environment, functions are often structured in packages to share with others. An R package consists of R code, data, tests, examples and documentations in an isolated and pre-defined folder structure. The bundled R package is standardized and easily shared. Zhu et.al. (2020) proposed an R-package-oriented SDLC process for internally developed R functions within an organization. It facilitates a consistent understanding of the coding purposes, intention of the developer, and prespecified algorithms.

In this paper, we provide detailed instructions on how to develop a specification for an R function or a group of related functions through examples, discussions and a live demo. We will suggest a specification template for R programs and propose a strategy for automatically generating documentation. This strategy takes advantage of automatic inconsistency checking. It also allows for generating specifications in different formats, including help files in live R sessions, PDF files, or website references with only a few simple clicks.

Roxygen is an in-line documentation syntax for R packages. Using Roxygen can streamline the specification development and maintenance process. We will discuss the benefits for using Roxygen when writing specifications in R code. We will also introduce some commonly used Roxygen tags through examples, and discuss how it relates to:

1) Documenting a package/function's dependences and internal function
2) Documenting a class of multiple functions involving a process step and reusing the same input arguments
3) Performing automatic inconsistency checking between the specification and the actual function

In addition, we will present three types of specification documentation outputs: a package's function help files, a PDF user manual, and a pkgdown website's reference page. These documents provide additional material supporting in addition to basic R programming code. We will review the different contents displayed in the PDF user manual and/or HTML help files. In the live demo we will illustrate how to generate these documents. We hope by sharing this information we will help organizations and the industry build a standard process for documenting R functions.

R FUNCTION DOCUMENTATION

The R interpreter passes control to the R functions, along with arguments that may be required for the function to accomplish specific actions. R has many built-in functions, and users can also create their own functions. A function performs its task and returns control to the interpreter along with any result that may be stored in other objects. An R function is written to carry out a specific task. It may have inputs and/or outputs. All R functions can be found in the `R/` folder in the R package.

Documentation of an R function should describe the purpose of the function, its intended use and expected result, and other critical information such as relevant literature, as needed. It should describe to the user how the function should be used, the input arguments required, and the output value the function returns. Every parameter used in the function, including optional parameters, should be explained.

An R function specification defines the function's scope, business requirements, development environment, etc. In a regulated clinical trial environment, a specification document is a critical component for facilitating communication between a statistician and statistical programmer, therefore, a specification needs to be properly documented to define the requirement and logics behind each function. The specification can be part of the R function documentation and be included in the R code. The function's documentation including the specification can be written in "R documentation" (`.Rd`) format, which is a simple markup language similar to LaTeX. All `.Rd` files go in the `man/` folder. An example of a source package is shown in Figure 1. We will present how to create a package with our example functions and corresponding documentations in the demo session.

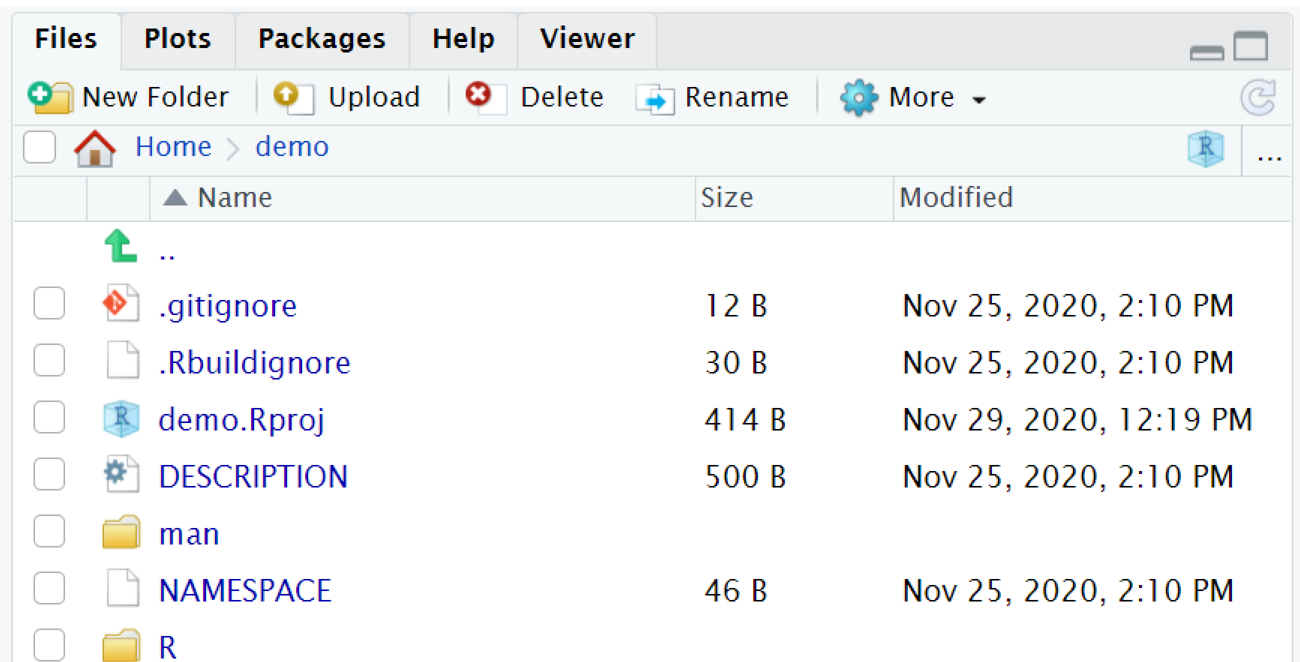

Figure 1: A typical R Package including R/ and man/ folder

By using the `Roxygen2` package to document R functions, a developer can simply develop an R function and write the corresponding documentation all together in an R file, without writing `.Rd` file directly. It simplifies the process to maintain R functions and its documentation in one place. In the later section, we

will illustrate how to embed R functions and specifications with potentially different outputs, by providing a template and some examples using Roxygen syntax.

RECOMMENDED R FUNCTION SPECIFICATIONS TEMPLATE

The `Roxygen2` (https://cran.r-project.org/web/packages/roxygen2/index.html) package can turn specially formatted Roxygen comments into R documentation (`.Rd`) files automatically. The typical `.Rd` file and Roxygen documentation are created for users to understand the R function already being developed and distributed. However, if a team is developing an R package for an organization, they also need a strategy to communicate the specification of an R function in the planning stage to ensure compliance and improve transparency of the SDLC. The proposed specification template leverages the existing Roxygen documentation and is straightforward to use.

The recommended specification template using the Roxygen comment tag is illustrated in the code below. Roxygen documentation starts with #' and comes right before the function to be documented. The first line is a title line. The title may be written as a sentence but without a period. The second line is a description which provides a general explanation of function's purpose. Long detailed sections may be organized by using the `@section` Roxygen tag. We recommend this section be used only to display the information to be include in the PDF manual, because R help files and package websites are typically for R package users instead of package developers. This can be achieved by using `\if{html}` for PDF manuals, `and` `\if{latex}` for R help files and package websites. Roxygen code blocks are used to ensure different content is displayed for the different type of documentation.

```
#' Function Title
#'
#' Function description, providing more contents than title
#'
#' @section Specification:
#' \if{latex}{
#' Write function description here.
#'
#'   1. step 1 of specification
#'   2. step 2 of specification
#'
#' }
#' \if{html}{
#'   The contents of this section are shown in PDF user manual only.
#' }
#'
#' @param var1 input variable 1
#' @param var2 input variable 2
#' @return the type of return value
#'
#' @examples
#' \dontrun{
#'     # define examples of the functions
#' }
#'
spec_tmp <- function(var1 = "Default Value", var2){
}
```

EXAMPLE WITH ROXYGEN COMMENT TAGS

The example code below uses the Roxygen template to create the function specification and documentation for a simple function to format a confidence interval.

```
#' Formatting a Confidence Interval
#'
#' Formatting number of digits displayed in a confidence interval.
#'
#' @section Specification:
#' \if{latex}{
#'   1. Checking x, x_low and x_up are numeric values
#'   2. Formatting x, x_low and x_up with assigned digits
#'   3. Assembling x, x_low and x_up to "x (x_low, x_up)"
#' }
#' \if{html}{
#' The contents of this section are shown in PDF user manual only.
#' }
#'
#' @param x the point estimator (vector).
#' @param x_low the lower bound of a confidence interval (vector).
#' @param x_up the upper bound of a confidence interval (vector).
#' @param digits the desired number of digits after the decimal point.
#' @return a confidence interval (vector)
#'
#' @examples
#' \dontrun{
#'     format_ci(0.001, -0.001, 1.2, digits = 1)
#'     format_ci(0.001, -0.001, 1.2, digits = 2)
#' }
#'
#'

format_ci <- function(x, x_low, x_up, digits){
}
```

From the simple example above, we have seen how to use:

- `@param`, a Roxygen tag to explain the meaning of each input argument of an R function (equivalent to parameters in a SAS macro).
- `@return` a Roxygen tag to describe the output value of an R function.
- `@section` a Roxygen tag to separate sections in the documentation.
- `@example` a Roxygen tag to provide examples for an R function. The code can be automatically executed while doing an R package compliance check (i.e., R CMD check), so the errors, if found, may be flagged. This is a useful feature compared to specifications for other statistical software such as SAS®.
- `\dontrun{}` prevents runing examples in the wrap code during the planning stage. Such code presents examples of scenarios or ways that the function should not be called.

The above includes inline documentation for the function's input, output, specification and corresponding examples. This information is essential for developers, validators, and users. Users may also practice using the example code on their own. The example code may be tested by R CMD check or `devtools::check()`. This code can also serve as a way for the developer to communicate with the validator to indicate the expectations of the validation. The HTML, PDF, or website reference version of the specifications may all be generated automatically through one line of code. The inconsistency between

specification and code may be checked by R package compliance checking tool such as `devtools::check()`. The specification spelling may be checked by `devtools::spell_check()`. Good programming practices can be checked by `lintr::lint_package()`.

DOCUMENT PACKAGE/FUNCTION'S DEPENDENCIES

There are multiple tags that may be employed for documenting a package or function's dependencies. Roxygen `@import` and `@importFrom` are tags to claim function usage from other R packages. Roxygen tag `@export` is used to make a function visible to users of an R package. A function without `@export` will be an internal function. `@inheritParams` is a Roxygen tag to re-use parameter documentations from another function. The use of this tag is illustrated in the code below:

First function `ae_sum` is defined as:

```
#' AE summary table
#'
#' @param population_from Data frame for population
#' @param population_where selection criteria, use tidyverse filter format
#' @param aearg Additional argument for AE Summary table
ae_sum <- function(population_from, population_where, aearg) {
#'
}
```

The second function `ae_count` defined as:

```
#' AE count table
#'
#' @param ctarg1 Additional argument for AE count table
#' @param ctarg2 Additional argument for AE count table
#' @inheritParams ae0sum
ae_count <- function(population_from, population_where, ctarg1, ctarg2) {
#'
}
```

In the example above, `@inheritParams ae_sum` means the second `ae_count` function will re-use the common parameters from the `ae_sum` function, namely `population_from` and `population_where`. This avoids repeating the same content, ensures consistency, and simplifies documentation maintenance. More details for object documentation can be found in Chapter 10 of Hadley's R packages book (https://r-pkgs.org/man.html).

BENEFITS OF USING ROXYGEN FOR R FUNCTION SPECIFICATIONS

Developing and maintaining R programs under R/ folder requires good specification documentation. The benefit of using Roxygen to document the specifications highlighted below.

First, Roxygen simplifies the specification process. Code and specifications are in the same file. Code and specification updating can be easily synced. This is reflected in the recommended specification template and the example code shown above. **Second**, R document .Rd files can be generated automatically by one-line of R code `devtools::document()` as below:

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/demo_format_ci.R
\name{format_ci}
```

```
\alias{format_ci}
\title{Formatting a Confidence Interval}
\usage{
format_ci(x, x_low, x_up, digits)
}
\arguments{
\item{x}{the point estimator.}

\item{x_low}{the lower bound of a confidence interval.}

\item{x_up}{the upper bound of a confidence interval.}

\item{digits}{the desired number of digits after the decimal point.}
}
\value{
a confidence interval (vector)
}
\description{
Formatting numbers to a confidence interval test.
}
\section{Specification}{

\if{latex}{
\itemize{
  \item this is item 1
  \item  this is item 2
}
Write function description here.
  1. Checking x, x_low and x_up are numeric values
  2. Formatting x, x_low and x_up with assigned digits
  3. Assembling x, x_low and x_up to "x (x_low, x_up)"

}

\if{html}{
The contents of this section are shown in PDF user manual only.
}
}

\examples{
\dontrun{
    format_ci(0.001, -0.001, 1.2, digits = 1)
    format_ci(0.001, -0.001, 1.2, digits = 2)
}
 }
```

Third, specification documentation can be generated using only one-line of code `devtools::build_manual()`. This avoids a manual effort to maintain separate documentation for a function.. The document is in a nice format and follows a consistent style. When double programming is required, the PDF document (Figure 2) provides the necessary specification and ensures that the double programming programmer does not see the original source code.

format_ci *Formatting a Confidence Interval*

Description

Formatting numbers to a confidence interval test.

Usage

```
format_ci(x, x_low, x_up, digits)

format_ci(x, x_low, x_up, digits)
```

Arguments

x	the point estimator.
x_low	the lower bound of a confidence interval.
x_up	the upper bound of a confidence interval.
digits	the desired number of digits after the decimal point.

Value

a confidence interval (vector)

a confidence interval (vector)

Specification

- this is item 1
- this is item 2

Write function description here. 1. Checking x, x_low and x_up are numeric values 2. Formatting x, x_low and x_up with assigned digits 3. Assembling x, x_low and x_up to "x (x_low, x_up)"

Examples

```
## Not run:
    format_ci(0.001, -0.001, 1.2, digits = 1)
    format_ci(0.001, -0.001, 1.2, digits = 2)

## End(Not run)
## Not run:
    format_ci(0.001, -0.001, 1.2, digits = 1)
    format_ci(0.001, -0.001, 1.2, digits = 2)

## End(Not run)
```

Figure 2: Specification shown in PDF format

Fourth, A live HTML help file is also generated in one step as an `.Rd` file as shown in Figure 3.

format_ci {demo}

Formatting a Confidence Interval

Description

Formatting numbers to a confidence interval test.

Usage

```
format_ci(x, x_low, x_up, digits)
```

Arguments

`x`	the point estimator.
`x_low`	the lower bound of a confidence interval.
`x_up`	the upper bound of a confidence interval.
`digits`	the desired number of digits after the decimal point.

Value

a confidence interval (vector)

Specification

The contents of this section are shown in PDF user manual only.

Examples

```
## Not run:
    format_ci(0.001, -0.001, 1.2, digits = 1)
    format_ci(0.001, -0.001, 1.2, digits = 2)

## End(Not run)
```

Figure 3: Live help file in R

Fifth, a website reference may be generated in one-line of code using `pkgdown::build_reference()` as shown in Figure 4 below.

demo 0.0.0.9000 Reference

Formatting a Confidence Interval

Formatting numbers to a confidence interval test.

```
format_ci(x, x_low, x_up, digits)
```

Arguments

x the point estimator.

x_low the lower bound of a confidence interval.

x_up the upper bound of a confidence interval.

digits the desired number of digits after the decimal point.

Value

a confidence interval (vector)

Specification

The contents of this section are shown in PDF user manual only.

Examples

```
if (FALSE) {
    format_ci(0.001, -0.001, 1.2, digits = 1)
    format_ci(0.001, -0.001, 1.2, digits = 2)
}
```

Developed by First Last.

Site built with pkgdown 1.6.0.

Figure 4: Integrated Website Reference

Sixth, no documents outside of the R system (such as MS Word documents) are needed to maintain specification files.

Seventh, as described in the previous sections, comprehensive R built-in checks are available to ensure consistency between Roxygen documentation and function definition.

Eighth, multiple user-friendly outputs including a PDF manual, an HTML help file,and a pkgdown website come from the same source.

Ninth, continuous integration and development can be used to streamline the development flow of function and the corresponding specification.

DISCUSSIONS AND CONCLUSIONS

Specifications are essential documents for R function development in our highly regulated environment. R function specifications can be written using Roxygen tags which may be checked for consistency seamlessly. For an R package in the R folder, the general specification is defined by using Roxygen comments, including `@param` to define a function's input argument, `@return` to define a function's output, `@examples` to provide some simple examples. The detailed specification is documented in the Specification section (`@section` Specification). In general, if the detailed specification is documented in the `\if{latex}` block, and no specification contents are shown in the `\if{html}` block, then the specification comments will only be displayed in the PDF manual. If the function needs to be added to the `NAMESPACE` file to be accessible for users, `@export` can be used. To reduce duplications and maintain consistency among functions, `@inheritParams` can be used.

Since the detailed specification section can only be visible in the PDF manual, it ensures any independent validation can be done without referring to the developer's code but still in alignment with the developer's updates. The code logic and changing history can be well presented using LaTeX within the specification section. The R function's Roxygen documentation can include code examples which may be checked automatically for errors. Through the HTML help file or `pkgdown` website generated by inline documentation, the developers and users can see how the functions are to be used.

In contrast, there is no integrated or consistent way to document internally developed macro in the SAS domain. Each organization has its own way of handling the important process of documenting specifications and they often rely heavily on creating separate Microsoft Word documents. Those word documents can be challenging to maintain and need to be checked to ensure consistency for each SAS macro that is developed. Maintaining the SAS macro's specification in a word document requires a more manual effort than automatically generating an R function's specification as proposed in this paper. This automatic method only requires documenting information in one place using a Roxygen comment tags which then generates different output formats (help file, PDF user manual, pkgdown website) consistently.

As R plays a more important role in our industry's regulatory settings, the R code specifications becomes essential. Here we proposed using Roxygen documents specification tags within the R code to easily generate the R document file (`.Rd`) and automatically apply consistency and compliance checks for Roxygen documentation. The document can be created in a PDF format, a website reference, or an in-session help page seamlessly. We hope the strategy and information sharing here will help your organization, as well as the industry in general, build a standard process for creating specification documentation.

REFERENCES

The R Foundation for Statistical Computing (2018) R: Regulatory Compliance and Validation Issues A Guidance Document for the Use of R in Regulated Clinical Trial Environments https://www.r-project.org/doc/R-FDA.pdf

Zhu, Y., Jajoo, R., Bai, C., Nepal, S., Woodie, D., Anderson, K., Zhang, Y. R Package Oriented Software Development Life Cycle in Regulated Clinical Trial Environments https://www.lexjansen.com/phuse-us/2020/tt/TT12.pdf).

Wickham, H and Grolemund, G. R for Data Science (had.co.nz)

Wickham, H. and Bryan, J. (2019). R packages: organize, test, document, and share your code. (2nd Ed.) https://r-pkgs.org/

AKNOWLEDGEMENT

The authors thank the discussions and reviews from Merck programming group colleagues, specially, thank Lisa Spring, Simiao Ye, Huei-Ling Chen, and Amy Gillespie for their careful reviewing and inputs.

CONTACT INFO

Your comments and questions are valuable and appreciated. The authors can be reached at

Aiming Yang, Ph.D.
Company: Merck & Co., Inc.
Address: 126 E Lincoln Ave, Rahway, NJ 07065
Email: aiming_yang@merck.com

Yalin Zhu, Ph.D.
Company: Merck & Co., Inc.
Address: 126 E Lincoln Ave, Rahway, NJ 07065
Email: yalin.zhu@merck.com

Yilong Zhang, Ph.D.
Company: Merck & Co., Inc.
Address: 126 E Lincoln Ave, Rahway, NJ 07065
Email: yilong.zhang@merck.com