**Paper CT10**

**Validating R functions against SAS outputs: How RTest can build a bridge allowing to validate R.**

Sebastian Wolf, Freelance Software Developer, Munich, Germany

Dr. Sergej Potapov, Roche Diagnostics GmbH, Penzberg, Germany

**ABSTRACT**

The evaluation of clinical study data is highly regulated. Up to now data gets analyzed by SAS tools or validated special purpose software. R is a new player in the field. Validating R outputs against SAS can be hard due to different data formats and data storage. RTest is an open-source tool that allows testing R outputs against XML. This standard format can handle .NET, Java or SAS software outputs. Roche applied the tool to test R-packages against SAS or Analyse-it outputs. Those packages perform tasks like method comparison and Variance Component Analysis. This talk will show how to generate SAS results comparing them to R function outputs in an easy manner. RTest creates pretty human-readable outputs from the comparison. This allows overcoming most challenges of validating R and using it in a regulated field.

**INTRODUCTION**

Writing R code in a regulated environment influences people’s life, directly. Not only their life, but whether they stay alive. This an important fact about writing code in a clinical environment. A feature in software that was not tested could mean the software produces an outcome, that your doctor interprets wrong which can cause you pain, because he takes a wrong treatment decision.

The same accounts for people coding the micro controller software of your car’s steering wheel. If you steer left because you do not want to hit a wall, you do not want your car to steer right and leave you flat and dead.

This is why all software need to be deeply tested. Inside a clinical environment tests are checked by regulatory authorities like the FDA or the TÜV in Germany. Not every regulator will be a coder or there will even be people who have never seen code. Some tests will even be written by people who cannot code. Tools like specflow [1] or cucumber [2] would such people to write tests and read test results.

There is such tool for easy writing and reading tests in R, yet. As a team we decided to provide such a tool called RTest. This paper will introduce the basic functionalities about RTest.

WHAT IS SPECIAL ABOUT RTEST?

To explain which features we put in RTest [3] a basic testing workflow is described:

1 Testing code starts with writing code. Your R-package will contain functions, classes and methods. These shall be tested.

2 Writing the tests now mostly includes calls like this:

my\_function(x, y){sums\_up(x, y) return(z)}

x <- 3

y <- 4

z <- 7

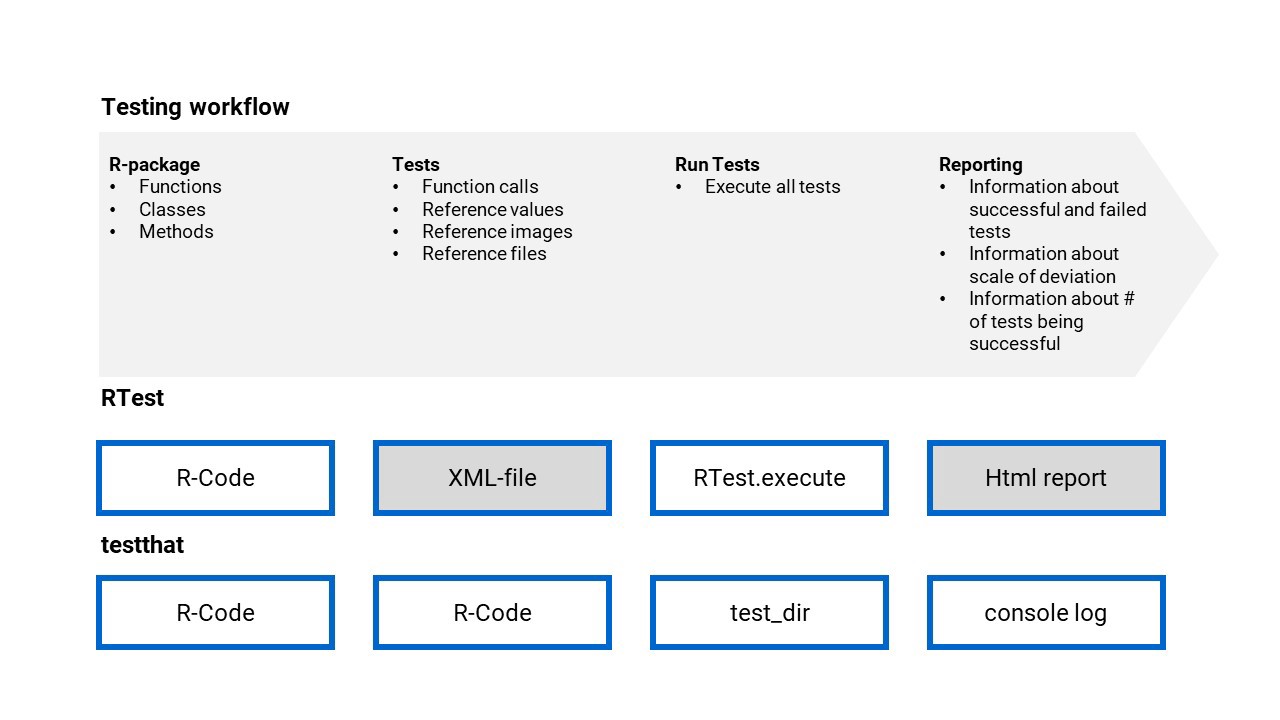
stopifnot(my\_function(x, y) == z)

It is easy to see, that your test will break if your function my\_function cannot sum up two values. You will create a bunch of such tests and store them in a separate folder of your package, normally called tests

3 Afterwards you can run all such tests. You can include a script in the tests folder or use testthat and runtestthat::test\_dir() .

4 If one of your tests fails, the script will stop and tell you which test failed in the console.

The figure below compares the 4 steps being executed in testthat vs. RTest.While RTest stores reference values in XML files, testthat stores those in pure R-code. In both cases tests are executed by calling an R function. While testthat reports to the console, RTest will show the outcome of the tests in a HTML report



***Figure 1: Testing workflow in RTest and testthat:*** *A code testing workflow is needed to run tests against R functions. For that RTest and testthat can be used. Each of them uses different functionalities and file formats to perform the task.*

EXECUTION STEPS FOR RTEST

1. The definition of the tests
2. The reporting of the test execution

For the definition of the tests we decided for XML. Why XML? XML is not just easier to read than pure R-Code, it comes with a feature, that is called XSD; “XML schema definition”. Each XML test case can immediately be checked against a schema designed by the developer. It can also be checked against our very own Rtest.xsd. This means the tester can double check the created test cases before even executing them. This saves us a lot of time and gives a fixed structure to all test cases. XML also allows to directly export reference results from SAS calculations. A lot of other programming languages also support XML. It is easy to store tables, values or even text outputs inside this format.

The reporting was implemented in HTML. This is due to the many features HTML comes with for reporting. It allows coloring of test results, linking to test cases and including images. The main difference for reporting between RTest and testthat is that RTest reports every test that shall be executed, not only the failed ones. The test report will also include the value created by the function call and the one given as a reference. The reader can see if the comparison really went right. By this the test report contains way more information than the testthat console log.

AN EXAMPLE OF A TEST IMPLEMENTATION WITH RTEST

Please note the whole example is stored in a github gist [4]

**1**. Given a function that sums up two columns:

my\_function <- function(data = data.frame(x = c(1,2), y = c(1,2))){

stopifnot(dim(data)[2] == 2)

data[, "sum"] <- apply(data, 1, function(x){sum(x)})

return(data)

}

2. We want to have one successful and one non successful test. Both will have three parts in the XML file:

<params><reference><testspec>

params accounts for input parameters

reference for the output data.frame

testspec for whether the test shall run silently and what the tolerance is

For the successful test our test would look like this:

<my\_function test-desc="Test data.frame">

<params>

<RTestData\_input\_data param="data" name="test01" />

</params>

<reference>

<col-defs>

<coldef name="x" type="numeric" />

<coldef name="y" type="numeric" />

<coldef name="sum" type="numeric" />

</col-defs>

<row>

<cell>1</cell>

<cell>2</cell>

<cell>3</cell>

</row>

<row>

<cell>1</cell>

<cell>2</cell>

<cell>3</cell>

</row>

</reference>

<testspec>

<execution execution-type="silent" />

<return-value compare-type="equal" diff-type="absolute"

tolerance="0.001" />

</testspec>

</my\_function>

RTest allows to use data sets for multiple tests, we store those data sets in the input-data tag. This saves space in the file. The dataset test01 will be used here. Moreover, a test description can be given for each test. For each data.frame stored in XML the types of the columns can be given in col-defs . Here those are all numeric.

<input-data>

<data.frame name="test01">

<col-defs>

<coldef name="x" type="numeric" />

<coldef name="y" type="numeric" />

</col-defs>

<row>

<cell>1</cell>

<cell>2</cell>

</row>

<row>

<cell>1</cell>

<cell>2</cell>

</row>

</data.frame>

</input-data>

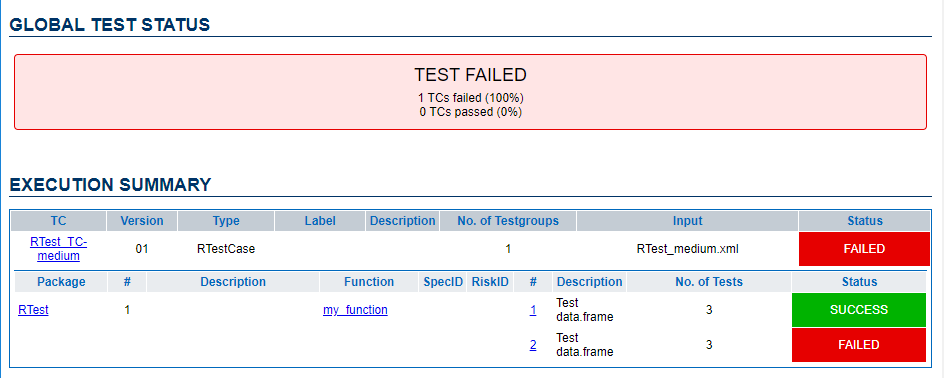
It’s a data frame where all values in the x column are equal to 1 and all values in the y column are equal to 2. The test shall create a data.frame with the sum column being 3 in each row.

We can easily let the test fail by changing the reference tag and instead of having just 3 in the sum column we can add a 3.5 to let the test fail. The whole test case can be found inside the github gist with 90 rows [4].

3. The execution of the test case is just one line of code. You shall have your working directory in the directory with the XML file and my\_function shall be defined in the global environment.

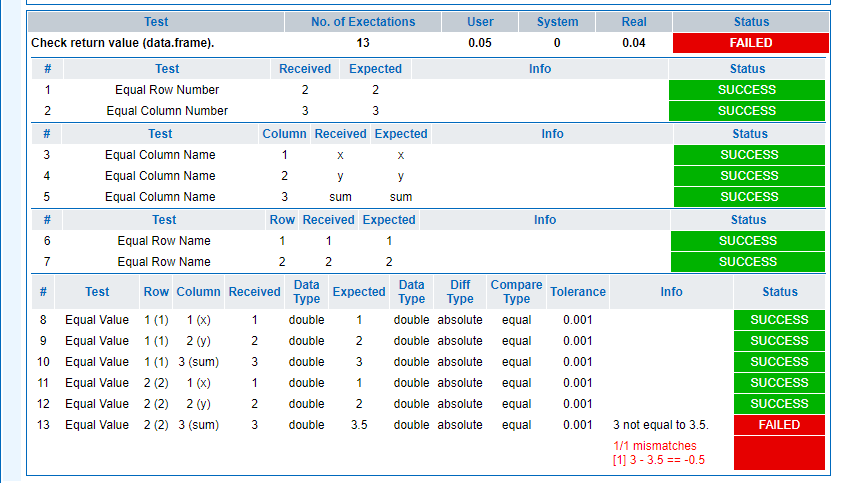
RTest.execute(getwd(),"RTest\_medium.xml")

4. The test report now contains one successful and one failed test. Both will be visualized:



***Figure 2: RTest test report header:*** *The RTest report header shows how many test cases were executed and which functions were executed. The shown test report shows a failed and a successful test case. Successful test cases are labeled with the word SUCCESS and a green background. Failed test cases are labeled with the word FAILED and a red background.*

additional information is given on all tests. For the test that failed we caused it by setting the sum to be 3.5 instead of 3. It’s reported at the end of the table:



***Figure 3: RTest test report details:*** *RTest reports show the evaluation time of each single test. Reference (Expected) values and execution values (Received) are given in separate columns. In case of a mismatch of reference values and execution values the difference is given. Tables are tested for their size, row names and column names as to be seen in the upper section of the report.*

Moreover, the Report contains information on the environment where the test ran:



***Figure 4: RTest test report session information:*** *The RTest report contains a section with all packages that were loaded inside the R session. This allows full reproducibility of the test run. In theory it is possible to run inside the exact same environment again by reinstalling all packages with the dedicated version.*

The test report not only shows for each test how it was executed, but also the execution time, if it was successful, the reference value and the outcome. Someone who knows what the software shall do from the[algorithm description](https://medium.com/datadriveninvestor/why-do-we-need-human-readable-tests-for-a-programming-language-1786d552f450#87af) can now by reading the test case and the test report, see what was tested and also see if this makes sense. For co-workers who are new to the project it is also way easier to find into the project. Reading test cases and report outcomes allows them to see in a minute which parts of the project still have problems or which functions are not yet tested.

**CONCLUSION**

Understanding how R software was validated now does not need an R programmer anymore. The environment presented here allows people to see how the software was tested. Human readable tests will make statistical software more fail-proof, easier to understand and more sophisticated. As R’s way from of a research environment into a clinical environments or even the car industries environment took place already, the process is not finished, yet. Many more tools will be needed to allow regulatory authorities to trust in such a big open-source project. Human readable test cases are a first step in helping companies to support the validity of their open-source solutions. Using R and a good testing framework will make people’s life more safe, because you’ll have not only great statistical tools, but great *validated* statistical tools.

**REFERENCES**

[1] Specflow : Human readable tests in .NET <https://specflow.org/getting-started/>

[2] Cucumber.io: Human readable tests for FrontEnd development <https://cucumber.io/docs/guides/overview/>

[3] RTest Documentation: <https://zappingseb.github.io/RTest/index.html>

[4] RTest github Gist <https://gist.github.com/zappingseb/0f5dabe94c7d284bc543469c50a4213c>

**CONTACT INFORMATION**

Your comments and questions are valued and encouraged. Contact the author at:

Sebastian Wolf

Freelance Software Developer

+49 1520 8847192

sebastian@mail-wolf.de

www.mail-wolf.de

Dr. Sergej Potapov

Team Lead Biostatistics

Roche Diagnostics GmbH

[sergej.potapov@roche.com](https://mail.google.com/mail/?view=cm&fs=1&tf=1&source=mailto&to=sergej.potapov@roche.com)

Brand and product names are trademarks of their respective companies.