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

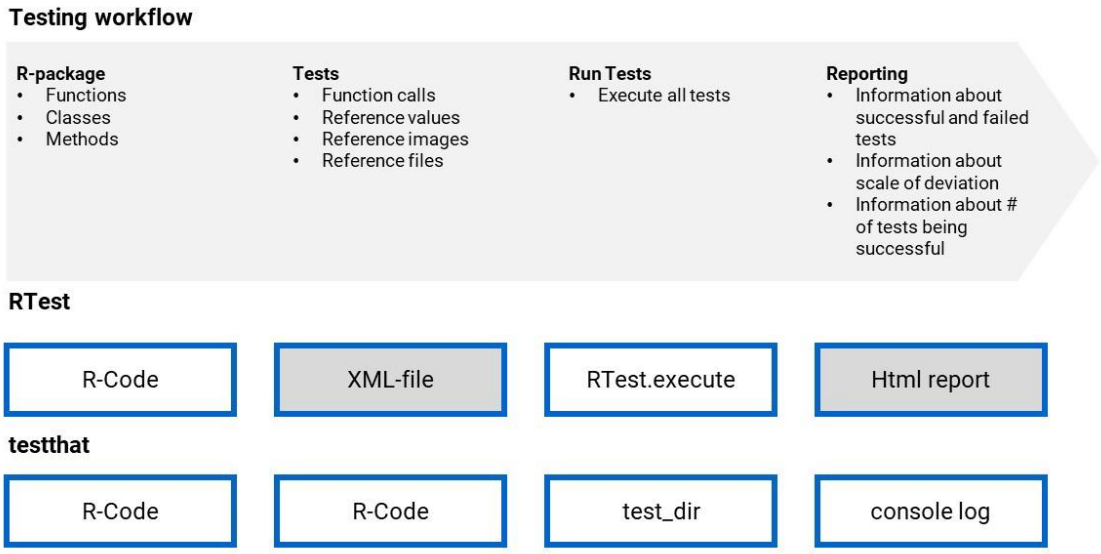


Figure 1: Testing workflow in RTest and testthat: *A code testing workflow is needed to run tests against R functions. For that RTest and testthat can be used. Each of them uses different functionalities and file formats to perform the task.*

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.

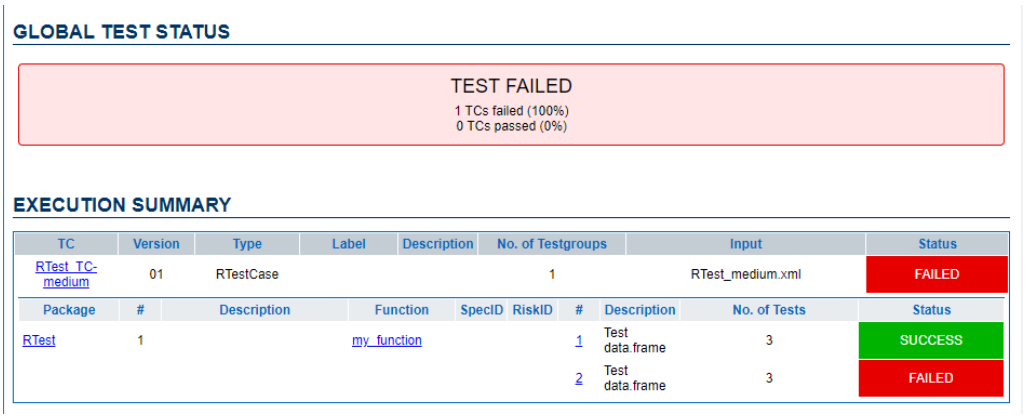
```
<input-data>
<data.frame name="test01">
  <col-defs>
    <coldef name="x" type="numeric" />
    <coldef name="y" type="numeric" />
  </col-defs>
  <row>
    <cell>1</cell>
    <cell>2</cell>
  </row>
  <row>
    <cell>1</cell>
    <cell>2</cell>
  </row>
</data.frame>
</input-data>
```

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:



GLOBAL TEST STATUS

TEST FAILED
1 TCs failed (100%)
0 TCs passed (0%)

EXECUTION SUMMARY

TC	Version	Type	Label	Description	No. of Testgroups	Input	Status
RTest_TC-medium	01	RTestCase			1	RTest_medium.xml	FAILED

Package	#	Description	Function	SpecID	RiskID	#	Description	No. of Tests	Status
RTest	1		my_function			1	Test data.frame	3	SUCCESS
						2	Test data.frame	3	FAILED

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:

Test	No. of Exectations	User	System	Real	Status
Check return value (data.frame).	13	0.05	0	0.04	FAILED

#	Test	Received	Expected	Info	Status
1	Equal Row Number	2	2		SUCCESS
2	Equal Column Number	3	3		SUCCESS

#	Test	Column	Received	Expected	Info	Status
3	Equal Column Name	1	x	x		SUCCESS
4	Equal Column Name	2	y	y		SUCCESS
5	Equal Column Name	3	sum	sum		SUCCESS

#	Test	Row	Received	Expected	Info	Status
6	Equal Row Name	1	1	1		SUCCESS
7	Equal Row Name	2	2	2		SUCCESS

#	Test	Row	Column	Received	Data Type	Expected	Data Type	Diff Type	Compare Type	Tolerance	Info	Status
8	Equal Value	1 (1)	1 (x)	1	double	1	double	absolute	equal	0.001		SUCCESS
9	Equal Value	1 (1)	2 (y)	2	double	2	double	absolute	equal	0.001		SUCCESS
10	Equal Value	1 (1)	3 (sum)	3	double	3	double	absolute	equal	0.001		SUCCESS
11	Equal Value	2 (2)	1 (x)	1	double	1	double	absolute	equal	0.001		SUCCESS
12	Equal Value	2 (2)	2 (y)	2	double	2	double	absolute	equal	0.001		SUCCESS
13	Equal Value	2 (2)	3 (sum)	3	double	3.5	double	absolute	equal	0.001	3 not equal to 3.5.	FAILED
											1/1 mismatches [1] 3 - 3.5 == -0.5	

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:

Attached Packages

Package	Version	Build	package_md5(Package)
RTest	1.2.3	2018-12-29	Namespace: f76fe938dded6a4a99afb60ed035c2b2
magrittr	1.5	2014-11-22 19:15:57	Namespace: c6813e52b2a5ca029a0622fbf08b2f53
base64	2.0	2016-05-10 23:57:02	Namespace: 4e1c4156abf5846e21132f73da71a9d5
XML	3.98-1.9	2017-06-19 12:43:32 UTC	tar(Package/R): 0a750fc07523a54d300f148a9d4efc95
magick	2.0	2018-10-05 10:00:03 UTC	tar(Package/R): 472f8403de403db7d1aca44cb0f6ed44
testthat	2.0.1	2018-10-13 08:40:03 UTC	tar(Package/R): 20c73a44fec4e85146f4561cf5a33829
rj	2.0.5-2	2018-02-19	tar(Package/R): fcda5ae72cf427f1af157c756089521e

Loaded, but not attached Packages

Package	Version	Build	package_md5(Package)
Rcpp	0.12.17	2018-05-09	tar(Package/R): c353b02758e111998a1efc60775b26a8
compiler	3.4.2	2017-09-28	tar(Package/R): 762142b0bf27598e91c9b8387cd851d5
remotes	2.0.2	2018-10-30 11:30:10 UTC	tar(Package/R): 22b4bd3a780a13dbaabb88bc95d586f3
prettyunits	1.0.2	2015-07-13 04:09:56	tar(Package/R): 4531077f589d05cdf2c4edbe2d382f9d
tools	3.4.2	2017-09-28	tar(Package/R): 6858e48b9fcebb6618c20c623c289eea
digest	0.6.12	2017-01-26	tar(Package/R): a4e8d8dbc9ded6905a22c2d4ea223a23
pkgbuild	1.0.2	2018-10-16 20:00:21 UTC	tar(Package/R): 09e9ede57770b0d8e3a945bde018b537

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 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