Paper SM10

From Hassle to Harmony: Unifying Code Quality Experience in R and SAS

Michael Hedegaard Thomsen, Novo Nordisk A/S, Aalborg, Denmark

ABSTRACT

When writing code for clinical trials, we aim to reduce programming time while maintaining high code quality. One way to achieve this is through unit testing. At Novo Nordisk, we use both SAS® and R for clinical trial programming, but SAS does not provide unit testing out of the box. In a multilingual programming environment, how can we efficiently ensure high code quality across both SAS and R?

By unifying the tools used, we can minimise context switching and enable cross-language collaboration. We have implemented a unit testing framework for SAS, based on the `testthat` package in R, along with internal helper packages.

Additionally, we leverage Continuous Integration/Continuous Deployment (CI/CD) infrastructure prepared for R and Python validation. With this approach, we have been able to deliver automated testing with fewer resources, and additionally we have streamlined the development process and simplified delivering high-quality code.

INTRODUCTION

Working as a programmer in the pharmaceutical industry was for many years associated with developing analysis datasets in SAS. With little or no support for unit testing of code, programmers were left to manually test their code, and at best, reusing and copying test scripts from one trial to another. Some companies developed standard (or global) programs that would be reusable across trials, but this came with large investments in preparing, conducting and documenting manual unit tests. In Novo Nordisk, this led to abolishing standard programs for many years.

The introduction of R development tools and packages also brought attention to testing frameworks such as `testthat`. Unit testing is a general way for programmers to build quality into their products, and package developers in R use this to ensure stable libraries that can be trusted by a broader audience. Quality is an abstract metric which encompasses several parameters; unit testing is just one aspect. Others include versioning, linting, cross-platform testing, cross-version assessments, configurability, maintainability, extensibility etc.

In Novo Nordisk, we have created two internal git repositories for sharing SAS code related to ADaM and TFL programming. These are seen as template repositories for producing programming deliverables for a submission. The code is not supported by unit tests and governance is done by a self-organised group in an inner source manner (Juul, 2023). The extent to which code is tested, is to the discretion of the individual that creates the component or modifies it. A simple pipeline setup is in place, where a change can be approved by a pull request and upon approval, the file is copied into a file location, from where trial programmers can pick up the file for use in their projects (or copy from the main branch of the repo). The Novo Nordisk definition of a standard program requires unit testing and documentation of a component, and this is why we distinguish between the terms *template* and *standard* program.

We are currently experiencing a paradigm shift from a SAS focused programming experience towards a multilingual environment where other languages such as R and Python can be utilised for programming regulatory submissions.

Building on previous work exploring multiple testing approaches (Thomsen, 2023), this paper focuses on how we have removed some of the hassle from testing SAS macros, by developing and improving internal R packages that supports executing and testing SAS macros interactively in e.g. RStudio, and how we utilise a general pipeline setup to execute tests and document results automatically.

SETUP

Figure 1provide a high-level overview of our SCE environment, which is detailed further in (Thomsen, 2023). Basically, a Unix environment is used for storing clinical data and this is also the compute for the SAS installation (lower right corner). Second, a compute cluster for Python and R programming called SCE-R is available (upper right corner) and finally we have Microsoft Azure DevOps (ADO) for hosting git repositories, development planning, pipeline facilities and test reports.

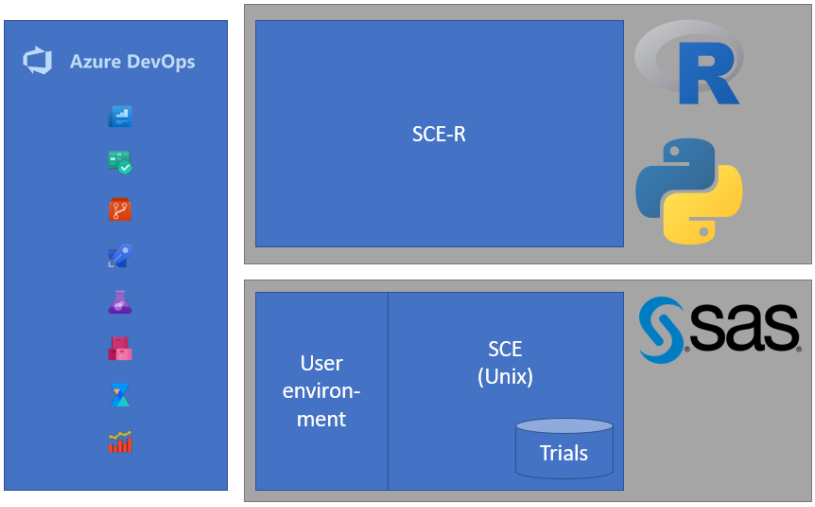

Figure 1. Programming landscape

Packages which are created in the SCE-R environment are developed according to a process whereby packages are required to pass unit tests and a code review before being deployed in production. This is controlled via the Git version control system and ADO pipelines and details will be explained later.

THE BUILDING BLOCKS

Transitioning from a Unix-based SCE with only a SAS installation and no source code control to a multilingual environment with automated task options, has sparked significant creativity for clinical data scientists (CDS) at Novo Nordisk. The work we have done to enable automated testing of SAS code relies highly on this, along with industry standard packages.

By (re-)using existing components, we are enabling a unified way of working cross language. This benefits both CDS' with no previous experience with R and ensures simpler maintainability. A short overview on relevant components will be described below.

THE `TESTTHAT` PACKAGE

The `testthat` package is a widely adopted unit testing framework for R. It provides simple and intuitive syntax for writing unit tests. Tests are organised in test files where several scenarios can be defined in blocks surrounded by the `test_that(description, code)` method. As input, two parameters are given. The `description` is a string naming the test scenario and this is used when reporting back the results of the tests. The second argument `code` is a block of code which can be divided into three sections:

1. *Setup*, where test data and other initialisation for the scenario is defined.
2. *Act*, where the functionality under test is executed. This is typically a step that manipulates the state of the test data.
3. *Expect*, which contains assertions on the state of the code after the *act*.

An example of a unit test implementation is provided in a later section, with the corresponding code illustrated in Figure 5.

The result of executing unit tests can be reported back in several ways. In the RStudio environment, where tests can be run through the console, the execution status is continuously printed as can be seen in Figure 2 to the right. When executing in an automated environment, machine-readable formats like `junit` are used instead, which makes it perfect for automated execution.

```
> devtools::test()
i Testing adamprogramming
✓ | F W  S  OK | Context
✓ |         1   0 | adamaddcorevars
✓ |     1      17 | adamadjudicationdupl [55.0s]
✓ |         7   0 | adamextractsdtm
✓ |         1   0 | adamfinalizer
✓ |         1   0 | adamimpbaseavg
✓ |             7 | adamimpbaselvpd [16.7s]
```

Figure 2. RStudio Unit Test Results

`testthat` is a cornerstone in our approach to align cross-language collaboration between R- and SAS programmers.

THE `NNREMOTE` PACKAGE

To be able to access and interact with clinical data (residing in the Unix part of the SCE) from the SCE-R environment, a package called `NNRemote` has been developed. This package serves as one of the core components for utilising R at Novo Nordisk. The responsibility of the package is to

- setup a connection to the clinical data repository by declaring mounts,
- enable execution of SAS code from R by use of `ssh` commands,
- read log files and `sas7bdat` datasets and convert into R readable format.

In the Unix SCE, users have a dedicated environment where they can work in isolation. When mounts between the Unix SCE and SCE-R have been established, `NNRemote` will use `ssh` to execute Unix commands, e.g. copying files, check connection, executing a `SAS()` command to execute code in a file, and clean up.

The `SAS()` command was an experimental proof of concept when first implemented. To be able to track changes during macro execution, a major refactoring of the function was done by implementing:

- Ability to execute pre- and post- code to setup e.g. test data.
- Prepare SAS code with
    - unique tagging (`%put` statements) of the log to distinguish the various steps performed during the call
    - logic to check for fileref, macro variable, and libname assignments during macro execution to track relevant changes
- More unit tests to improve code quality.

THE `SASUNITTEST` PACKAGE

The idea of using R to test SAS code arose from a coffee chat. We had a vision of being able to unit test SAS code to improve quality and efficiency in statistical programming and explored both homemade tools (Thomsen, 2023) and other existing tools such as `SASUnit` and `FUTS`.

However, these tools focused solely on the SAS environment, while we needed a solution that accommodated a multilingual context. If we could somehow build unit tests in the same (R-based) tool, the potential for re-use between languages, re-use of pipelines and upscaling of colleagues with a SAS background would be big.

A solution, the internal `SASunittest` package, was developed and demonstrated at PHUSE EU Connect 2023 (Thomsen, 2023) and PHUSE SDE Brussels 2024 (Johnsen and Phelps, 2024). This solution bridges `testthat` and `NNremote` by generating SAS code that execute the macro with the provided parameters in the test suite. An example is shown in Section IMPLEMENTING A UNIT TEST below.

THE SAS TEMPLATE REPOSITORIES

As mentioned in the introduction, we store SAS template programs in two git repositories, `adam_programming` and `tfl_programming`. The final goal of the project is to enable automated tests and documentation hereof.

To setup an automated flow for testing the SAS template code, we wrap the two repositories as R packages. This makes it possible to do development and interactive testing using RStudio. With this in place, we can then later validate the repositories using our pipeline flow as described below.

IMPLEMENTING A UNIT TEST

For an example of a unit test implementation, consider the `adamimpdtearly` macro. The macro description and signature are depicted in Figure 3 to the right. The macro imputes dates in an input dataset and adds a flag if imputation takes place. As can be seen, the rule imputes a missing day to the first day of the month, a missing month as the first month in the year, and if year is missing, no imputation is expected to take place.

Below in Figure 4, different test scenarios for the `adamimpdtearly` macro are shown.

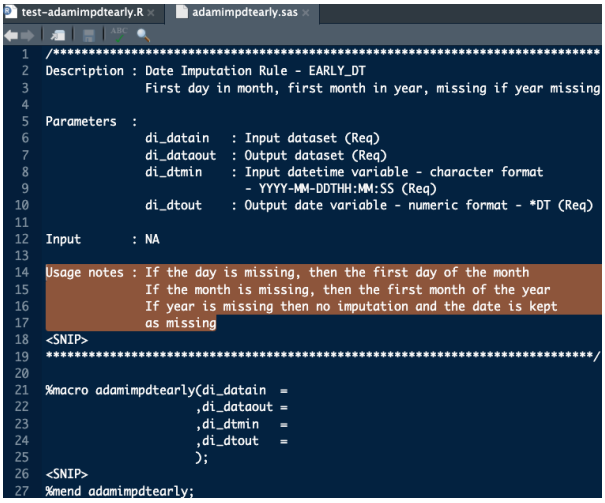

Figure 3. `adamimpdtearly` macro signature

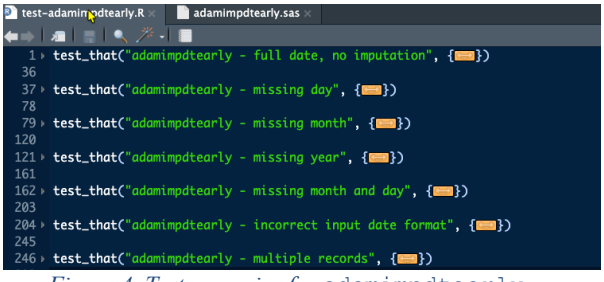

Figure 4. Test scenarios for `adamimpdtearly`

An example implementation for a scenario where no imputation should take place, is shown in Figure 5 on the right.

First, test data is defined. In this scenario a complete date with no need for imputation is defined in line 5.

In line 8, the function `SASunittest::r_function_from_sas_macro()`, creates an empty R function with a signature identical to the SAS macro. This provides the test developer with code completion in RStudio when defining parameters. Also, when calling this function, SAS code defining the macro, and calling the macro is generated. This happens in line 10-15. The variable `macro_code` will thus contain the following SAS code:

```
%macro adamimpdtearly(
  di_datain  =
 ,di_dataout =
 ,di_dtmin   =
 ,di_dtout   = );
/* macro code intentionally removed */
%mend adamimpdtearly;

* Macro call;
%adamimpdtearly(
  di_datain=rin.testdata
 ,di_dataout=test_out
 ,di_dtmin=AESTDTC
 ,di_dtout=dt_imp);
```

test-adamimpdtearly.R | adamimpdtearly.sas

```
1  test_that("adamimpdtearly - full date, no imputation", {
2    # SETUP ----------------------------------------------------------------
3    testdata <- tibble::tribble(
4      ~USUBJID, ~AESTDTC,
5      "01", "2023-05-06T00:00:00"
6    )
7    # Macro function
8    adamimpdtearly <- SASunittest::r_function_from_sas_macro("adamimpdtearly")
9    # Prepare SAS code
10   macro_code <- adamimpdtearly(
11     di_datain = "rin.testdata",
12     di_dataout = "test_out",
13     di_dtmin = "AESTDTC",
14     di_dtout = "dt_imp"
15   )
16
17   # ACT ------------------------------------------------------------------
18   result <- NNremote::SAS(main_code = macro_code,
19                           data_input = list("testdata" = testdata), ) |>
20     SASunittest::post_process()
21
22   # EXPECT ---------------------------------------------------------------
23   imputed_data <- result$tables$test_out
24
25   # Check imputed date
26   expect_equal(as.character(imputed_data[["dt_imp"]]), "2023-05-06")
27
28   # Check format of imputed date
29   expect_equal(attr(imputed_data[["dt_imp"]], "format"), "DATE9.")
30
31   # Check imputation flag
32   expect_equal(as.character(imputed_data[["dt_impf"]]), "")
33 })
```

Figure 5. Test scenario implementation where no imputation takes place

With the SAS code prepared, the `SAS()` method of `NNremote` (lines 18-19) can now execute the code on the remote host. The method returns a list object with generated tables, table metadata, programs, macro variables, fileref and libname assignments, and logs. These returned objects can then be assessed in the test assertions.

The call `SASunittest::post_process()` in line 20, processes the SAS log and transforms warnings or errors into R warnings and errors. This step can be avoided if the test scenario should test for expected errors by using a function `SASunittest::expect_sasError(result, "imputation failed")`.

In this scenario, we extract the resulting dataset and do assertions on this in lines 26, 29, and 32.

UNIT TESTS PREPARED BY PROMPTING

Since the `adam_programming` and `tfl_programming` repositories didn't have any tests, a set of test suites needed to be built. The maintainers of the template repositories have a very limited allocation. To ask them to use the framework and create unit tests would be a huge burden and the same applies to the project group.

To accelerate test creation, we leveraged Generative AI through our internal ChatGPT instance. We developed two prompts to generate test case scenarios, test data and the R code to set up the unit tests using the `testthat`, `NNremote`, and `SASunittest` packages. Tests would be finalised with assertions (`expect_*` functions) defined manually and should undergo peer review. We ended up with unit tests for 22 macros with approximately 120 test scenarios containing 400 assertions.

AZURE DEVOPS PIPELINES

The last building block is the CI/CD pipeline. ADO supports pipelines that enables execution of automated tasks. Tasks can be triggered when actions on a git repository takes place, such as commits, pull requests etc. With the R eco system in Novo Nordisk, we have set up generalised CI/CD pipeline infrastructure (Bilgrau, Knoph and Larsen, 2023).

The ADO pipeline dashboard with stages is depicted in Figure 6 below. As can be seen, seven major stages are defined. It is beyond the scope of this paper to give all details, so an overview is provided here.

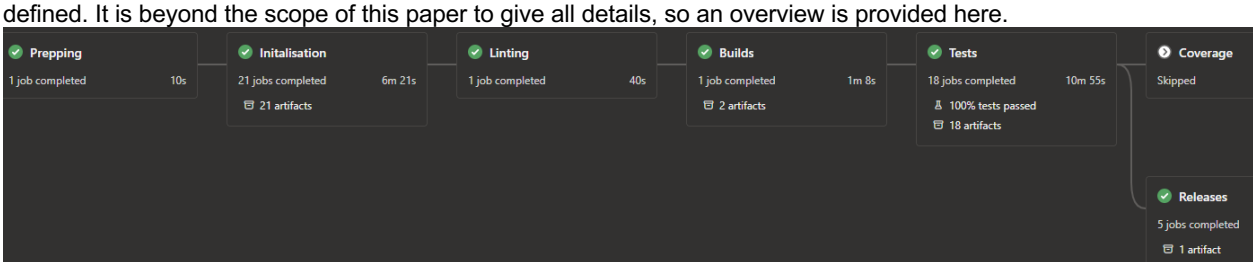

Figure 6. CI/CD Package Pipeline Stages

Our pipelines support repositories residing in our internal ADO, and in GitHub. **Prepping** is a matter of cloning the relevant repo. After this, **Initialisation** starts, where dependencies are checked against different versions of R. If a package is not compatible (e.g. having references to other packages that are not valid in the given version), the code will not be checked against this version. The next step is **Linting** where syntax and predefined rules for coding style are checked. For this project, we have not implemented specific linting for SAS, hence this is only relevant for the parts of the package that is R specific, which in turn is the implemented unit tests developed using the packages `testthat`, `NNremote`, and `SASunittest`. The **Build** stage builds artifacts (the package tarball and homepage) for later use in the Release stage. **Tests** are where the package unit tests are executed on all valid environments (R-versions) as determined in the initialisation step. If test step is successful, coverage can be tested. **Coverage** may be enabled for packages and is generally recommended but was bypassed for the current version, as described later. Finally, a **Release** step is executed. If all previous stages are successful, the package is installed on a package manager and documentation is published.

PUTTING IT ALL TOGETHER – THE INTERACTIVE SETUP

Before implementing unit tests for the two SAS template repositories, core team maintainers relied solely on manual validation processes. Without automated guardrails to test code changes, the team depended on manual code quality assurance practices, including ad-hoc testing procedures.

The model for setting up the project was agreed between the project team and the maintainer team. The project team's main responsibility was to provide an infrastructure to do unit testing of existing components. However, with around 20 macros, limited knowledge in the maintainer team of how to work with unit tests, limited time allocation, and only little experience working with R, it would be a huge burden to simply hand over the infrastructure. By doing ChatGPT prompting as described previously, the project team was able to establish a basic set of unit tests that would fit with the infrastructure, and enable many tests, including edge cases. These would still need to be verified by the maintainer team, but the burden was reduced to a minimum.

With the repositories, the packages mentioned above, and access to RStudio, it is a simple task to work interactively with the framework as can be seen in Figure 7. A member of the maintainer group can:

1. Clone the repository from an RStudio environment
2. Execute the `devtools::test()` method. The packages `testthat`, `SASunittest`, and `NNremote` ensures tests are executed in the SAS environment.
3. View test results displayed in the console window (as depicted in Figure 2)
4. If needed, make updates to either tests or macros and re-run steps 2 and 3
5. When code works as expected and tests pass, the code can be committed and pushed to the central repository

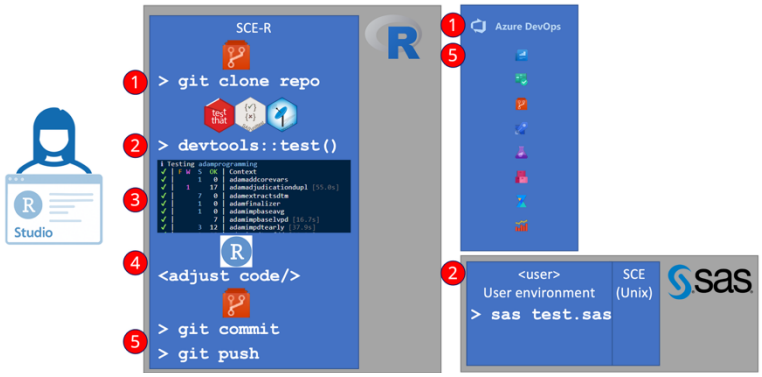

Figure 7. User workflow in interactive setup

If either a test or a macro needs updates/adjustments, this can be done either in RStudio (and interactively tested), or updates to the macro can be done in e.g., SAS Enterprise Guide, and testing can again take place in RStudio.

This interactive approach allows maintainers to rapidly iterate on code changes with immediate feedback, significantly reducing the validation cycle time.

PUTTING IT ALL TOGETHER – THE AUTOMATED SETUP

While interactive testing provides immediate feedback during development, the true value of automated testing is realised through continuous integration pipelines.

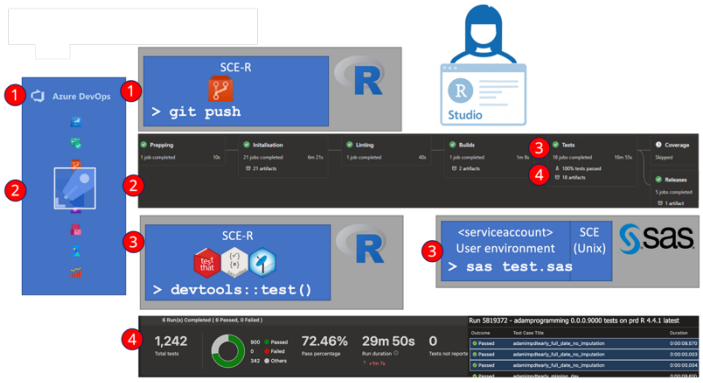

Figure 8. Pipeline workflow

Continuing from the interactive workflow in Figure 7, the automated workflow is shown in Figure 8:

1. The end user pushes changes to Azure DevOps.
2. This initiates a CI/CD pipeline workflow (for details refer Figure 6).
3. In the test phase, an R environment is created, where tests are executed in a similar way as in step 2 of the interactive workflow. Here, the user is not the end user but a service account (for details, see the next section).
4. Finally, the test results are stored as part of the ADO Test Lab, where they can be referenced based on the retainment settings of the pipeline run (examples are shown in a later section (see also Figure 10 and Figure 11)).

With the existing pipeline infrastructure and the building blocks described above, implementing automated testing required addressing several infrastructure and configuration challenges. The following subsections describe the key changes to enable fully automated testing in our CI/CD pipelines.

SERVICE ACCOUNT

Normally when a user interacts with an SCE this is done with personal credentials. To implement true automated task execution, this needs to be handled by pipeline agents and service accounts with proper access to network resources. Before the project started, we didn't have a dedicated service account for this purpose.

Most pipelines in the CI/CD setup in Novo Nordisk does not need to interact with the SAS environment. The `NNremote` package was built with unit tests that were interacting with Unix SCE and SAS, but these were not enabled in CI/CD pipelines due to the need of a service account. Hence these tests could only be executed by an end user in the interactive setup.

Thus, a service account was created, and pipelines were adjusted such the service account could be used for interacting with the Unix SCE. With this in place, unit tests in the `NNremote` package could now run on the CI/CD pipelines.

CONFIGURATION

At the time of creation, the `NNremote` package main objective was to establish access to the clinical data environment from the R environment. This was done by `ssh` commands and was successful. As mentioned previously, a `SAS()` method was implemented with the purpose of running SAS commands and getting logs and datasets back after execution.

Learnings from using it, identified a need to make it less coupled to the Unix SCE. The connection details were hardcoded inside the package, and this had to be loosened to prepare for future changes in the environment setup. This would make it easier to make changes to the infrastructure setup without recompilation and installation of new packages. Hence, a *variable group* in ADO was prepared for storing secrets, server names, service account credentials, timeout settings etc. By reading and passing these variables, hard coding was avoided. This is particularly true for the `NNRemote` package which was partly rewritten to make interaction with the SAS environment much more configurable.

MISSING DEBUG INFORMATION

During initial automated pipeline runs, we encountered test failures that were difficult to diagnose. When things go wrong in an automated environment, you need detailed information about what happened and where. Unlike interactive testing where developers can immediately inspect variables and execution state, automated testing requires comprehensive logging. Without sufficient debug information, troubleshooting these environment-specific failures was extremely time-consuming.

We therefore had to adjust the main packages to be much more verbose. As mentioned previously, it is important to be able to adjust this without needing to recompile and re-deploy packages. Thus, by creating a *DebugLevel* variable, we would be able to toggle debugging whenever things went wrong. By creating a *DebugLevel* as an integer (as an alternative to a simple boolean), the verbosity can be adjusted, depending on the need to track details.

This became very helpful when we experienced non-deterministic failures. Tests that were passing every time in a manual setting would sometimes fail in an apparent arbitrary way in the automated setup. Without information to use for debugging, problem solving would be like finding the needle in a haystack.

Having detailed information in a log is an improvement but we also wanted to see from where (from which test scenario) the error occurred. Hence the test names should also be added to the log before the debug information is written. To do this, we made an override of the `test_that()` function as can be seen in Figure 9 to the right. This made it possible to follow the logic flow when automated tests were executed.

```
test_that <- function(desc, code)
{
  filename <- ""
  if (NNremote_options("DebugLevel") > 0) {
    cat(sprintf("\n%1$s %2$s Testcase: %3$s %1$s \n",
                "====", filename, desc))
  }
  testthat:::test_that(desc, { code})
}
```

Figure 9. Overriding the `test_that()` *function*

NON-DETERMINISTIC FAILURES

With more logging capabilities in place, it was possible to pin the reasons for unit tests failing when not running interactively. During implementation, we discovered that network connections were less stable than expected. This led to tests failing in a non-deterministic manner. The message "`kex_exchange_identification: read: Connection reset by peer`" was seen in the log on various occasions.

The solution to this was to implement try-catch-retry facilities around `ssh` calls. With a configuration for setting up command timeout, max attempts, and lag period before retrying, this part was stabilised. With the improvements, we could still find connection error messages in the logs, but the tests recovered by retrying the calls and this made the `NNremote` package much more robust.

REDUCE NUMBER OF ENVIRONMENTS

Another issue with the general pipeline setup is that the tests are executed on different R versions and for three staging environments (development, validation and production). Currently we support five R versions with a locked set of packages and in addition we also test the latest R version with the most recent packages from CRAN, resulting in 18 total configurations (6 R versions x 3 environments). These are determined in the **Initialisation** step as depicted in Figure 6.

This extensive testing ensures cross-version compatibility in R package development scenarios where version dependencies must be verified. However, in this scenario it only adds complexity and time overhead since the SAS code is not depending on R versions.

For this reason, we restricted the package to the latest Novo Nordisk supported R version, and thus reduced the number of tests to a minimum:

- Tests on newest version of R installed on our system.
- Tests on newest version of R with latest packages from CRAN.
- Tests performed on all three environments (development, validation, and production).

This means we have been able to reduce the number of configurations from 18 to 6.

CODE COVERAGE

Since code coverage is part of the pipeline setup, we explored ways of enabling this. By using the SAS option `mcoverage`, we were able to enable macro coverage. However, we quickly ran into the problem that data step execution would appear as having 100% code coverage. We investigated possible solutions for handling this but due to time constraints, prioritisation, and the risk of giving false positives, we decided not to implement this part until we could guarantee a correct code coverage.

SAS LOG DIFFERENCE FROM INTERACTIVE VS AUTOMATED EXECUTION

SAS logs are made for human readability and possibly printing. Metadata such as page numbers and execution timestamps would interfere the interpretation of the logs. The obvious choice was to adjust a setting like `pagesize` to ensure only one log page was produced.

However, we also experienced that our use of an internally developed Unix command to execute SAS code, was suddenly adding information to the SAS log. From an operating perspective there was a need to add debug information to track issues on performance in the Unix SCE. The immediate effect was that tests that checked length or positions of specific text in the log, suddenly failed and needed adjustments. Situations like these were beyond the project group's control, and we therefore needed to do changes to these tests on an ongoing basis.

DOCUMENTING TESTS

As mentioned earlier, we decided to use the existing pipeline infrastructure. This has the benefit that we did not need to put efforts into documenting test execution as this is part of the pipeline infrastructure. Hereby we could use the dashboards provided by ADO out of the box. On the right side in Figure 10, an overview on executing tests for the `adam_programming` repository is shown.

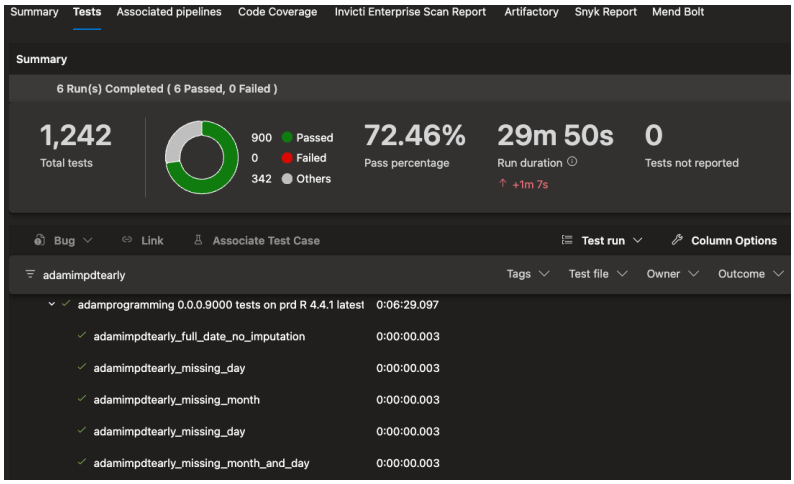

Figure 10. Tests Results in Pipeline Dashboard

As can be seen, 900 assertions were *Passed*. The 342 with outcome *Others*, are tests that were marked to be skipped because they needed thorough review of the logic to ensure correctness.

Recall that numbers displayed in the figure are the result of executing tests in six environments. It is possible to select tests executed in one environment only. This can be seen in Figure 11 below.

Figure 11. Unit Tests Documentation from ADO Test Planning

END USER ADOPTION

The end users can be grouped in two: The template maintainers of the `ADaM_programming` and `TFL_programming` repositories, and the CDS' working with clinical trials. So far we have only handed the solution over to the template maintainer group. We are confident that the unit testing framework will ensure higher quality in SAS programming for both groups.

A remaining challenge is how to identify standard programs in a trial, and how to handle different versions of these. Standard programs in Novo Nordisk does not have to be signed by the CDS using them, but even if we can unit test components, we still lack tooling that supports different versions of components/macros, and utilities that can identify whether a program needs to be signed or not. Before this is in place, end users will not get the full benefits from this framework.

For this reason, the maintainers have not done the final peer review of the unit tests that were prepared for the two template repositories and hence the framework is not fully implemented.

CONCLUSION

It has always been a hassle to validate standard programs for use in statistical programming in Novo Nordisk. By utilising both existing and newly developed components, we have established an infrastructure for automated unit testing of SAS components. The path was not straightforward and required investments in terms of adjustments to existing pipeline setup and R packages.

With the implemented changes, we have matured existing packages by improving robustness, enabling logging facilities, and ensuring a looser coupling to infrastructure by enabling configuration in pipelines and packages.

By sharing a common framework for unit testing between SAS and R developers, we can minimise context switching and enable cross-language collaboration. This helps traditional SAS programmers develop their R programming skills.

The unit testing framework is a major advancement towards harmony for validating standard programs. The next step in our journey to a general framework for standard programming may explore how we can provide tooling to control versions and identify components that needs review.

REFERENCES

Bilgrau, A. E. (2022). "R Package Development at Novo Nordisk." PHUSE EU Connect 2022, Presentation TT03.

Bilgrau, A. E., Larsen, S. F., and Knoph, A. S. (2023). "Novo Nordisk's Journey to an R based FDA Submission"

Johnsen, N. S., and Phelps, M. (2024). "Rewriting Our Global Validation Process for SAS Programs Using R, Azure DevOps and Git: A Case Study of Code Validation in a Multilingual Environment." PHUSE SDE Brussels 2024, Presentation 06

Juul, J. R. (2023). "InnerSource: A Stepping Stone Towards Open Source in Statistical Programming." PHUSE EU Connect 2023, Paper TT13.

Karpow, A. (2017). "Catching Up: Continuous Integration Pipelines for Clinical Analysis." PHUSE EU Connect 2017, Paper TT03.

Thomsen, M. H. (2023). "The Hassle of Validating Global SAS programs: Made Easy(ier)." PHUSE EU Connect 2023, Paper SM08.

Wright, J. (2006). "Drawkcab Gnimmargorp: Test-Driven Development with FUTS." SUGI 31, Paper 004-31.

ACKNOWLEDGMENTS

The project and components described in this paper are results of several Novo Nordisk colleagues' efforts. Special recognition goes to Matthew Phelps, Nicolai Skov Johnsen, Steffen Falgreen Larsen, and Anders Bilgrau.

CONTACT INFORMATION

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

Michael Hedegaard Thomsen
Novo Nordisk A/S
Alfred Nobels Vej 27
DK-9220 Aalborg Oest
Denmark
Email: mhzt@novonordisk.com
Website: www.novonordisk.com