Paper SM08

The hassle of validating global SAS programs: Made easy(ier)

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

ABSTRACT

Efficiency through automation is a growing trend in statistical programming, and we in Novo Nordisk have recognized the need to streamline our global validation process for SAS® programs. Currently, this process is manual, comprehensive, and time-consuming, leading to a low turnover rate and frustrated end users who resort to making their own adaptations. This undermines the goal of having a global validated program and limits the number of programs that can achieve this status.

To address these challenges, we have developed an automated test framework using Azure DevOps CI/CD pipelines, SAS programs, and Git. This approach has enabled us to make rapid updates to the programs, while ensuring their functionality and providing full transparency throughout the workflow. By automating the validation process, we have reduced the workload on maintainers and increased the turnover rate, resulting in a more efficient and effective system.

INTRODUCTION

This paper describes the Novo Nordisk journey towards implementing CI/CD pipelines for unit testing SAS programs. It focuses on the problem of enabling automated testing abilities in an environment that was not intended for such a purpose and paving the way forward.

STANDARD PROGRAMS – HISTORIC PERSPECTIVE

At Novo Nordisk, we have used *standard programs* for many years. In short, standard programs are defined as programs that any programmer can take off the shelf and implement in a trial, without needing to review and sign off on the code. The code is “by standard” reviewed and approved for use in any trial.

Producing reusable code is a best practice approach that should be implemented whenever possible. However, reusable components come with a cost. The quality needs to be high, and this means documentation should be thorough, and testing of such code components should be done in a very structured and analytical way.

This also means that reusable components are expensive to maintain, and therefore there is a cost-benefit consideration when deciding whether to implement standard programs or not. If any trial can reuse the component, it may be cheap for the business to implement the component as a standard program. But if the business is only expecting to use the component in a very specific situation, this can imply that it will only be used on a few trials, and the effort spent to make it a standard program may be too high to accommodate the need.

A way to ensure the high quality of a component is to introduce unit tests. With predefined data, it is easy to validate the results and thus identify errors occurring in either a newly developed or a modified component. Long term, it will also reduce cost for maintaing the code.

For a component to be validated using unit tests, a number of test cases must be identified and documented. Code to support unit tests needs to be prepared to enable the re-execution of the test cases. Running the test cases must then be documented by inspecting results, logs, etc., and signed off on for the execution.

For many years, conducting a test run was a manual task, and to some extent, it still is today at Novo Nordisk. The documentation of running test cases involved looking through the SAS log to identify relevant messages, data sets to validate data, etc. A few test case examples related to a macro that can calculate the Body Mass Index (BMI) of a person are available in Appendix 1.

Manual processing comes with a cost. Developing, executing and documenting manual tests is a repetitive task, which is error-prone and can distract Statistical Programmers from other vital activities which are more related to their skillset. This introduces a risk that the test cases are not verified or documented correctly. Another risk is that the test run can fail and needs to be repeated when a bug is fixed. This both delays deployment and cost time and ressources. A solution to this is to enable automation of such processes.

ENVIRONMENT OVERVIEW

To set the scene for the remainder of this paper, it is important to get an overview of the Statistical Computing Environment (SCE) in which our Statistical Programmers develop code. A high-level overview of our environment is shown in Figure 1. For many years, the main component was the (UNIX) SCE (lower right corner), which contains trial data, a SAS installation, and a user environment. Later, the SCE-R was added, which today is an environment for developing in Python and R.

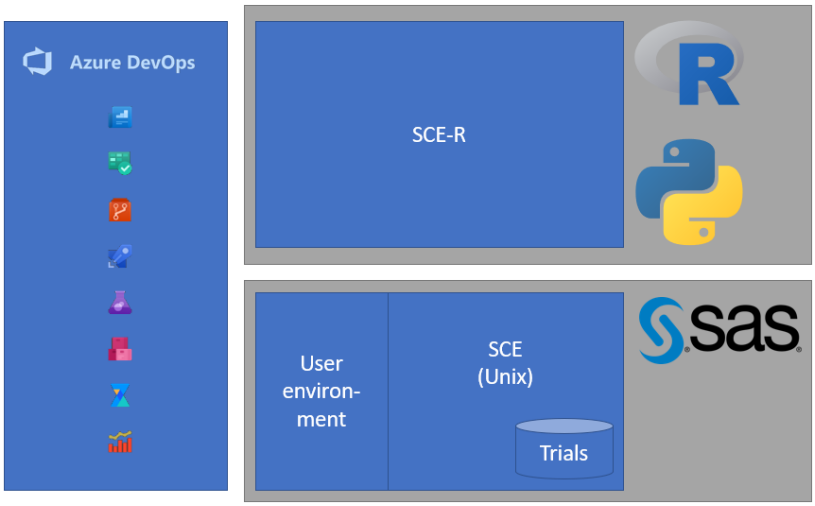


Figure 1. Programming landscape

With the introduction of the Azure DevOps (ADO) platform in Novo Nordisk, we gained access to a new set of tools that can enable automation to a greater extent. An important part of the platform is the ability to set up pipelines that can execute scripts and code on remote machines (physical or virtual). We also gained access to using Git repositories to store code.

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 Git and ADO pipelines.

To simplify the development of SAS components (programs and macros) on UNIX SCE, we have developed an analogous strategy to the one used for package development in the SCE-R, using ADO Pipelines to circumvent some of the limitations in the UNIX SCE system architecture. In the course of our development, we have explored three different approaches to enable unit tests in our traditional SCE.

MACRO UNIT TEST TOOL (MUTT)

An initial initiative to enable automation of running test cases was the Macro Unit Test Tool (MUTT), which was presented at PHUSE EU Connect in 2022.

The concept was to make it simple to run test cases for data imputation macros and thus enable structured and automated testing of components. To allow for easy adaptation for a community that was largely unfamiliar with quality assurance via unit testing, it was important that the process of setting up test plans with test data was simple. In order to automate the testing process, the specifications and related data needed to be available in a data source that is easy to read from the SAS installation.

Using an Excel workbook as a user interface, MUTT enables different test suites to be defined in the first sheet. The data to be used for testing is then defined in successive sheets. An example is shown in Figure 2. The grey boxes refer to the input data set `ADAE_01`. The macro in scope for testing is called `adamimpdtearly` and the implementation can be seen in Appendix 2.

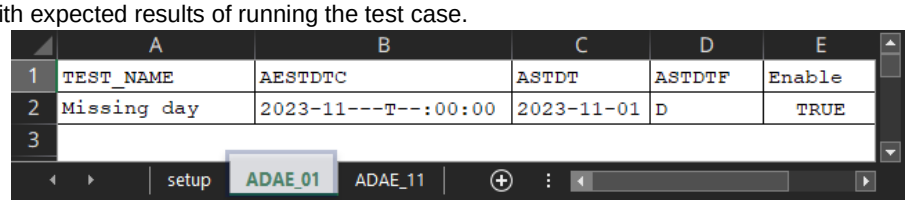
	A	B	C	G
1	Testsuite	Data	Parameters	Enable
2	Impute missing day to first in month	ADAE_01	di_datain = ADAE_01, di_dataout = ADAE_02, di_dtmin = AESTDTC, di_dtout = ASTDT	TRUE
3	Impute missing month to first in year	ADAE_11	di_datain = ADAE_11, di_dataout = ADAE_12, di_dtmin = AESTDTC, di_dtout = ASTDT	TRUE

setup | ADAE_01 | ADAE_11

Figure 2. MUTT test specification

The macro is used for ADaM programming to impute the start date to the earliest possible date if a date part of an SDTM variable is missing. The macro will create a copy of the input dataset, add two new variables to contain the (imputed) date, and a flag to signal to what level a date has been imputed (being set to "`M`" if the month is imputed and "`D`" if only the day is imputed and blank otherwise). In SDTM this could be an `AESTDTC`-variable having the value "`2023-11---`". Then `ASTDT` will be imputed based on `AESTDTC` by setting the date part as "`01`" (i.e., "`2023-11-01`"), and add a flag `ASTDTF` to the derived data set with a value of "`D`".

An example of a test case is shown in Figure 3 below. The input data is given in column `AESTDTC` (as specified in the Parameters column with `di_dtmin=AESTDTC` in the specification in Figure 2), and columns `ASTDT` and `ASTDTF` are prepopulated with expected results of running the test case.



	A	B	C	D	E
1	TEST_NAME	AESTDTC	ASTDT	ASTDTF	Enable
2	Missing day	2023-11---T--:00:00	2023-11-01	D	TRUE
3					

setup | ADAE_01 | ADAE_11

Figure 3. MUTT test cases

When MUTT is run, it searches for macros in the folder structure (leftmost picture in Figure 4), and if a corresponding Excel workbook with the same name as the macro is found (middle picture), it runs the test cases associated. The result data set from running a macro and its test data is stored in a result folder underneath the test specification (right picture).

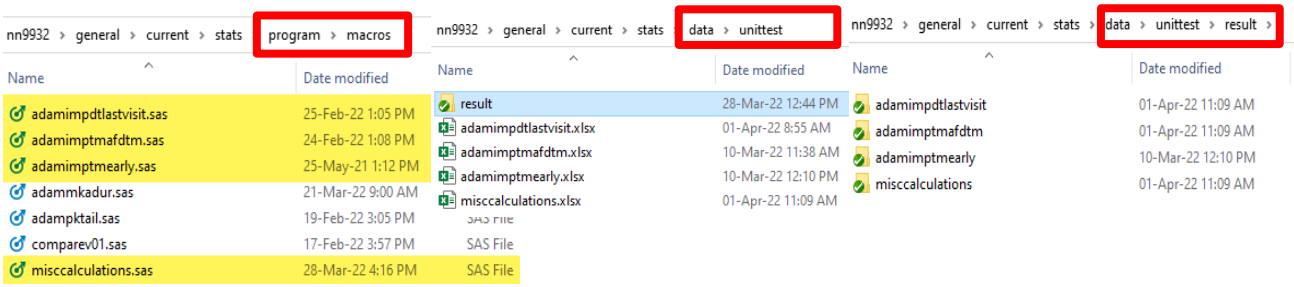


Figure 4. Folder structure for macros, test suites and test cases, and test results

Reporting of the test run is logged in the SAS Enterprise Guide log window. An example can be seen in Figure 5. The system was set up such that it would be possible to generate a result file to integrate into ADO, for example in the form of a JUnit report file.

```
1811  NOTE:    ================================================================
1812  NOTE:    Unit test results
1813  NOTE:    ================================================================
1814
1815  ERROR:   Macro adamimpdtlastvisit has  2 failed tests with input data 'ADAE'
1816  NOTE:    Macro adamimpdtlastvisit has 12 passed tests with input data 'ADAE'
1817  ERROR:   Macro adamimpdtlastvisit has  1 failed tests with input data 'ADAE_AOUT'
1818  NOTE:    Macro adamimpdtlastvisit has 13 passed tests with input data 'ADAE_AOUT'
1819  ERROR:   Macro adamimptmafdtm has  1 failed tests with input data 'ADAE'
1820  NOTE:    Macro adamimptmafdtm has  6 passed tests with input data 'ADAE'
1821  NOTE:    Macro adamimptmafdtm has  6 passed tests with input data 'ADAE2'
1822  NOTE:    Macro adamimptmafdtm has  5 passed tests with input data 'ADAE_OK'
1823  NOTE:    Macro adamimptmafdtm has  5 passed tests with input data 'adamimptmafdtm_adae.sas7bdat'
1824  NOTE:    Macro adamimptmearly has  3 passed tests with input data 'ADAE'
1825  NOTE:    Macro adamimptmearly has  3 passed tests with input data 'ADAE_NO_SECONDS'
1826  ERROR:   Macro misccalculations has  1 failed tests with input data 'input'
1827  NOTE:    Macro misccalculations has  4 passed tests with input data 'input'
1828  NOTE:    Macro misccalculations has  2 passed tests with input data 'linput'
```

Figure 5. Result of running MUTT

As convenient as the MUTT concept is for these use cases, the MUTT framework is limited by only being able to test components that do data manipulation. The lack of assertions other than those that would test for data equality made it necessary to look for additional components to integrate with MUTT. There are other unit test packages like FUTS and SAS Unit available, but these were deemed too hard to implement in the MUTT framework. Because of these limitations, the results were never integrated with ADO.

ADAM TOOLBOX MACRO TEST FRAMEWORK

The ADaM toolbox is a set of Novo Nordisk internal macros which can read trial metadata stored in Excel sheets and generate a study define.xml document, a zipped package with trial data to be validated using Pinnacle 21, and compare study metadata with Novo Nordisk standards and project standards. The toolbox is widely used for both preparing study data specifications and for submission preparation, and is thus a cornerstone in ADaM programming in Novo Nordisk.

Since the outcome of the toolbox is of vital importance for all trials, it is important that the functionality can be trusted. Thus, the components are standard programs and contain lots of documentation and heavy test cases.

Until recently, testing these components was done manually. This was very inefficient, so a framework for testing them was created. The toolbox consists of 7 macros and 7 helper macros (utilities).

Since the toolbox components play a significant role in the validation of standard programs, they are also version controlled using Git. This way, the maintainers can document changes made to a component during its lifetime. The testing framework is designed such that every macro in the toolbox has a corresponding folder in the Git repository containing test cases. The test cases are named with an ascending sequence number starting from 01.

The test cases execute the macro, possibly with a given set of (static) test resources that are available in a `datasets` folder (examples are SAS data sets, Excel sheets, files, etc.). During the test case execution, several assertions are tested. If all assertions pass, a positive result is returned (and vice versa). Each test case writes the result of running the test case to a file `test_res.txt` that is common among the test cases for a given component. A simple line is added to the file, indicating whether the test was successful or not.

As an example, consider a component `m_codelists`. For this component, the folder structure could look like in Figure 6.

```
+- m_codelists/
|  +- test_cases/
|  |  +- tc01.sas
|  |  +- tc02.sas
|  |  +- …
|  |  +- tc99.sas
|  |  +- test_res.txt
|  +- datasets/
|     +- <data>/
+- m_codelists.sas
```

Figure 6. Example folder structure for the ADaM toolbox test framework

Sometimes a correct execution of a test case should end with a message in the SAS log (could be Error, Warning, or any custom log message). To collect and validate these, a special test case is run after all other test cases have been executed. Here, the contents of all SAS logs are read and verified against the expected log messages for each component (where relevant). Again, the `test_res.txt` file is appended with the result.

Since a `test_res.txt` file is generated for each component, many `test_res.txt` files exist, and all of these must be checked in the end to look for failed test cases.

To gather the complete results, a shell script is run to check test case status. Each test case that fails will be reported, and a signal to fail the script (using `exit 1` code) is reported. If all test cases pass, the code `exit 0` is returned.

With this setup, it is possible to run all test cases and collect the results. Since this can all be scripted, the process can be automated. Thus, a pipeline is set up such that when code is pushed to the remote repository, the pipeline copies the contents of the repository to the SCE, executes test cases on the SCE (i.e., in a GxP validated environment), and collects and reports the results back to the pipeline. The effect of this is that the pipeline execution can be halted in case of an unexpected result. This will mean that the programmer receives a notification that the execution was erroneous, and thus a modification to either the code, test case, or test data must be performed before a new push is done.

An example of a successful execution of the pipeline is depicted in Figure 7. On the left-hand side of the figure, a green bullet with the text “Run all test cases” is seen. This is the step that runs all test cases, by issuing a command to run all SAS test programs defined in the `<macro>\test_cases\` folder. The next green bullet “Run check program” is the step that collects all the results and reports back to the pipeline. In this scenario, an exit code 0 indicates successful execution of the step.

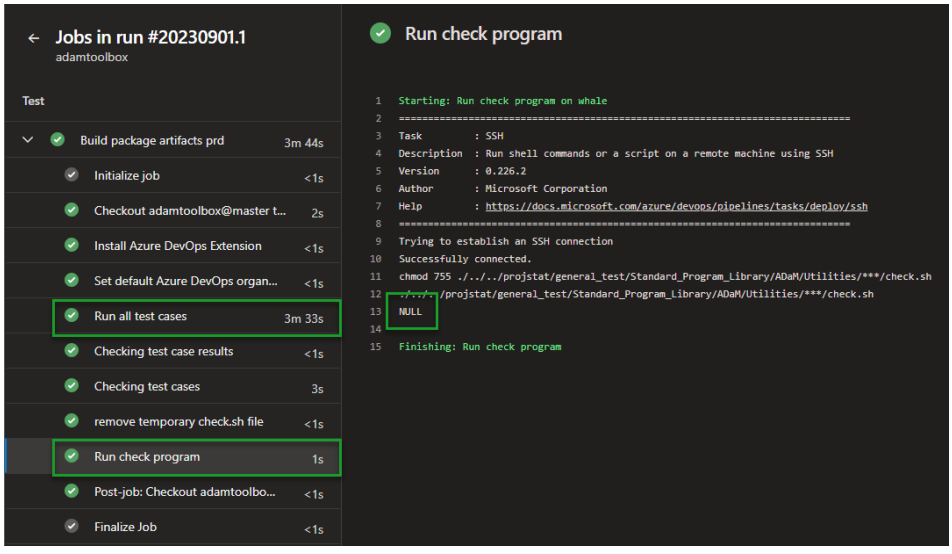


Figure 7. Example of successful pipeline execution

In Figure 8, an example of a failed execution is shown. As in the successful scenario above, the “Run all test cases” bullet is marked with a green checkmark. The interpretation is that all unit tests have been executed. However, this time the “Run check program” has a red bullet indicating a failure to complete the step. In the log for this step (right-hand side), the macro(s) that fail execution are listed, along with an indication of the test case(s) that failed. Here, the pipeline receives an exit code 1 to indicate that the step did not end as expected. This will halt the pipeline and thus indicate to the user that something went wrong.

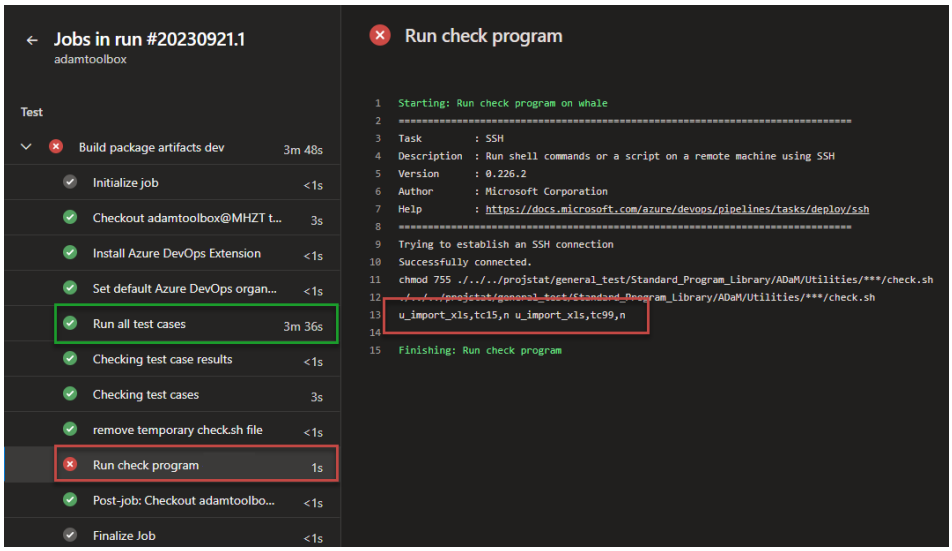


Figure 8. Example of failing pipeline execution

This framework only uses ADO and the SCE (refer to Figure 1 for an overview of the environment). Tests will be performed in a dedicated test SCE environment before deploying components in production.

There are a few caveats with this solution. The test cases are created as SAS programs, without the option to use a real unit test framework. Therefore, the reporting is programmed in-house and is not very specific in case a test fails – it is reported that a certain test case did not produce the expected outcome but with no details. On the ADO platform, it is possible to see test results in special reports. To do so, the test report must be in a format that is supported by the platform. This has not been implemented yet and will be a manual task. Furthermore, since the framework is custom-built, it requires insights into the framework to extend it with further test cases, etc.

FRAMEWORK FOR STANDARD PROGRAMMING

The current iteration of enabling unit tests for SAS code has the same ambition as the ADaM Toolbox Macro Test Framework described above: to validate SAS components and report results that can be used as documentation for testing the components in scope.

In this scenario, we wish to leverage the efforts that have been made in our SCE-R-environment to enable integration with SAS. We have developed an R-package called *NNremote* which facilitates integration with our traditional SCE. This package contains a `SAS()` function which can execute SAS code. This function sets up an environment for the user (be it a personal or a machine account) in a Unix home folder in the SCE (referring to Figure 1, this is the left-hand part “User environment” in the SCE). Doing so isolates the environment from the main project area (where our general programming framework and clinical data are stored) and thus enables execution in a clean and disposable environment. This makes it possible to iteratively test components in our GxP validated environment, without production code being influenced by any external dependencies.

Being able to execute SAS code via the SCE-R-environment brings the benefit that we can make use of standardised and well-documented R-packages. The *testthat*-package is the standard for doing unit tests in R and comes with a set of assertions which can be used for creating test cases.

When using *testthat*, unit tests are, by convention, defined in a `test\testthat\` folder relative to the root of the repository. This makes *testthat* aware of which tests to execute.

Besides the assertions that come with the *testthat* package, we need to enable more SAS-specific assertions (e.g., assertions related to the SAS log, libnames, etc.). For this purpose, we have created a package called **SASunittest**. The purpose of this package is to enable testing of SAS components in a simple way. The functionality can be summarised as follows:

- **use_SASunittest**: Sets up an R-project for unit testing SAS code. The function prepares an existing repository with an R-project file and integrates the *testthat* package. Among other things, this means a folder structure for unit tests is set up in the repo.
- **use_macro_unittests**: Loops through all code in the repository to search for SAS macro definitions. For each macro definition found, a template test-file is created in the `testthat` folder.
- **run_SAS_macro**: Given a macro name, the macro code is found and prepared to be sourced in the (SAS) SCE. If the macro needs arguments, these are given as an argument.
- **expect_<various>**: Assertions that relate to SAS code, i.e., checks for macro variables creation, libnames, SAS log, etc.

The *SASunittest* package wraps functionality in the *NNremote* `SAS()` function, making it possible to offload code to the SCE, execute the code, return an object with tables, macro variables, libnames, filenames, SAS log, and the SAS program that was executed.

Using this object, a number of assertions can be used for validating the code. This includes the *testthat*-specific assertions, as well as assertions made in the *SASunittest* package.

As an example, consider the SAS macro called `adamimpdtearly` which was also used in the example of the MUTT framework (refer to Appendix 2 to see the macro code). In Figure 9, a test case for the macro is shown. The test data is set up in lines 3-9, the arguments to the macro are defined in lines 11-14, and finally, the macro is called in lines 16-20.

```
1  test_that("adamimpdtearly impute missing day to first day in month", {
2
3    testdata <- '
4  data adae_01;
5  input aestdtc $ 1-19 astdt_expected :yymmdd10. astdtf_expected $ 32;
6  format astdt_expected yymmddd10.;
7  cards;
8  2023-11---T--:00:00 2023-11-01 D
9  run;  '
10
11   macroargs <- "di_datain  = adae_01,
12                 di_dataout = adae_02,
13                 di_dtmin   = aestdtc,
14                 di_dtout   = astdt"
15
16   sas_out <- SASunittest::run_SAS_macro("adamimpdtearly",
17                                         arguments = macroargs,
18                                         search_path = ".",
19                                         precode=testdata)
20
21   # Check for errors and warnings
22   SASunittest::expect_sasErrorFree(sas_out)
23   SASunittest::expect_sasWarningFree(sas_out)
24
25   testthat::expect_length(sas_out$tables, 2)
26   testthat::expect_equal(names(sas_out$tables),
27                          c("adae_01", "adae_02"),
28                          label = "Expected tables not present")
29
30   # Check number of elements in output data set
31   testthat::expect_length(sas_out$tables$adae_02$astdt, 1)
32
33   # Check that expected columns are available
34   testthat::expect_equal(names(sas_out$tables$adae_02),
35                          c("aestdtc","astdt_expected", "astdtf_expected", "astdt", "astdtf"))
36
37   # Check data is imputed as expected
38   testthat::expect_equal(as.character(sas_out$tables$adae_02$astdt),
39                          as.character(sas_out$tables$adae_01$astdtf_expected),
40                          label="Data is not imputed as expected: ")
41 })
```

Figure 9. A test case implementation for a macro that imputes dates

The major functionality takes place in lines 16-20. The `run_SAS_macro` function will:

- Locate and parse the SAS macro definition recursively within the search path
- Use the *NNremote* package to create an environment in the user area of the SCE (see Figure 1) for execution of the code
- Source the `adamimpdtearly` macro in this environment
- Execute the macro with the arguments `macroargs`. The parameter `dt_datain` in the `macroargs` refers to the data defined in lines 4-9, i.e., the data set `adae_01`

The return value, as stored in `sas_out`, is an object that contains a list of tables, macro variables, libnames, and filenames created during the execution, as well as the SAS log and the executed code. Based on this, assertions can be made:

- SAS errors or warnings are not expected. This is checked in lines 22-23
- The number of tables is checked in line 25. If the parameter `dt_dataout` is different from `dt_datain`, two data sets are expected. The names of the tables are also verified (line 26-28)
- The number of records is verified in line 31
- In lines 34-35, the variable names of the output data set are checked
- Finally, the imputed data is verified against the `astdt_expected` column in the input data set.

Using a general package to execute unit tests comes with a number of advantages. One of these is reporting. The *testthat* package in R provides a comprehensive and informative reporting system for the results of unit tests. When running tests, it automatically provides a report which can be displayed either in the console or a report can be exported in JUnit format, which can be integrated with ADO.

The code can be run interactively in an R-session which, in the successful scenario, will produce a list of results in the console of the IDE (see Figure 10). If any assertion yields a negative result (e.g., if the expected result is defined as "`2023-01-01`"), the report will instead display the failed assertion, as can be seen in Figure 11.

```
> devtools::test_active_file()

══ Testing test-adamimpdtearly.R ═══════════════════════════════════════════
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 27 ] Done!
```

Figure 10. Successful test run from console

```
══ Testing test-adamimpdtearly.R ═══════════════════════════════════════════
[ FAIL 1 | WARN 0 | SKIP 0 | PASS 6 ]

── Failure (test-adamimpdtearly.R:38:3): adamimpdtearly impute missing day to first day in month ──
Data is not imputed as expected:  (`actual`) not equal to as.character(sas_out$tables$adae_01$astdt_expected) (`exp
ected`).

`actual`:   "2023-11-01"
`expected`: "2023-01-01"

[ FAIL 1 | WARN 0 | SKIP 0 | PASS 6 ]
```

Figure 11. A failed assertion

With the end goal of being able to execute unit tests using CI/CD pipelines, reporting must also be done in a format that can integrate into dashboards. For this, the JUnit format fits perfectly. In Figure 12, a dashboard from ADO is shown. On the left-hand side, a summary can be seen, and on the right-hand side, the details for each test case are shown.

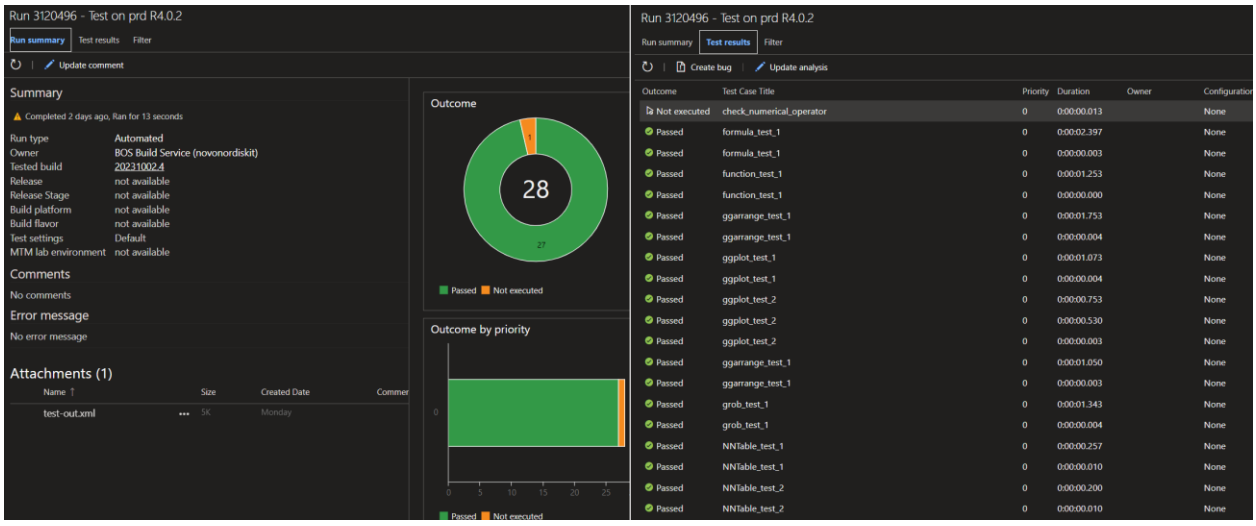


Figure 12. Dashboard in ADO displaying a test run

The report gives an overview of the number of passed test cases and can be used as part of the documentation for a test run. The individual test results are similar to the manual test report which was generated prior to this framework.

CONCLUSION

Enabling automated unit testing of SAS code can be done in many ways. The amount of effort needed depends on the approach taken, and the uptake in the organisation will also depend heavily on this. Using a well-documented framework will not only make it easier to implement test cases but also to integrate results for automation.

By implementing test cases in a unit test framework, and integrating test reports with CI/CD pipelines, the manual process of executing tests and verifying the outputs of these can be avoided. This means the workload on maintainers can be potentially reduced, and an increased turnover rate can be expected.

Using a single unit test framework for both SAS- and R-programmers will make it possible for users in both groups to share a common language around implementing unit tests. As a side effect to this, we may be able to introduce the R-environment to a set of users who have a traditional SAS background. This may trigger their journey towards becoming multilingual programmers and statisticians, proficient in the two main programming languages used for biostatistics work.

ACKNOWLEDGMENTS

The frameworks described above are results of several Novo Nordisk colleagues' efforts. Special recognition goes to Frederik Vandvig Heinen, 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
Work Phone: +45 30790904
Email: mhzt@novonordisk.com

APPENDIX 1: MANUAL TEST CASE EXAMPLES

This appendix shows two examples of test cases and test run. The test cases are focused around calculating the BMI at baseline. Verifying the outcome (e.g., data and log messages) is a manual inspection task.

Test case 1 is a straightforward calculation using the data in column *Input Description* and the derived (expected) data is expressed in column *Expected Outcome Description*. The calculated result can be seen in the variable `find_val_in_std_unit` which is calculated to `38.89`. The person responsible for running the test case has signed for the execution in the *Executor* column and noted that the test passed.

Test Case no.	Test Case Description	Input Description	Expected Outcome Description	Executor	Passed/ Failed	Com-ments
1	Both Weight and Height is valid	subj_id=5001 trl_contr='BIAsp-1530' trl_id='BIAsp-1526' assm_type='BASELINE' topic_cd='WEIGHT' find_val_in_unit=154 find_std_unit='KG' possible_range='N' err_cviol='N' vis_id=20 subj_id=5001 trl_contr='BIAsp-1530' trl_id='BIAsp-1526' assm_type='BASELINE' topic_cd='HEIGHT' find_val_in_unit=1.99 find_std_unit='M' possible_range='N' err_cviol='N' vis_id=20	subj_id=5001 trl_contr='BIAsp-1530' trl_id='BIAsp-1526' topic_cd='BMI' find_val_in_std_unit=38.89 find_std_unit='KG/M^2' vis_id=20 appl_find_rule='BMI0'	PITO	Passed	

Test case 18 is an example of a test case where an error is expected in the SAS log. The error occurs because the weight is assessed three times at baseline (`vis_id 20`). In this scenario the BMI cannot be calculated, which can be seen in the *Expected Outcome Description* data part where the variable `find_val_in_std_unit` is set to missing.

Test Case no.	Test Case Description	Input Description	Expected Outcome Description	Executor	Passed/ Failed	Com-ments
18	Several valid values for Weight at Baseline for Trial <actual trial id> and Subject <actual subject id>	subj_id=5007 trl_contr='BIAsp-1530' trl_id='BIAsp-1530' assm_type='BASELINE' topic_cd='WEIGHT' find_val_in_unit=89 find_std_unit=KG possible_range='N' err_cviol='N vis_id=20 subj_id=5007 trl_contr='BIAsp-1530' trl_id='BIAsp-1530' assm_type='BASELINE' topic_cd='WEIGHT' find_val_in_unit=79 find_std_unit=KG possible_range='N' err_cviol='N vis_id=20 subj_id=5007 trl_contr='BIAsp-1530' trl_id='BIAsp-1530' assm_type='BASELINE' topic_cd='WEIGHT' find_val_in_unit=88 find_std_unit=KG possible_range='N' err_cviol='N vis_id=20 subj_id=5007 trl_contr='BIAsp-1530' trl_id='BIAsp-1530' assm_type='BASELINE' topic_cd='HEIGHT' find_val_in_unit=1.83 find_std_unit='M' possible_range='N' err_cviol='N' vis_id=20	In LOG: ERROR: Several valid values for Weight at Baseline for Trial BIAsp-1530 and Subject 5007 subj_id=5007 trl_contr='BIAsp-1530' trl_id='BIAsp-1530' topic_cd='BMI' find_val_in_std_unit=. find_std_unit='KG/M^2' vis_id=20 appl_find_rule='BMI10'	PITO	Passed	

APPENDIX 2: ADAMIMPDTEARLY MACRO

```
/*************************************************************************
Description : Date Imputation Rule - EARLY_DT
              First day in month, first month in year, missing if year missing.

Parameters  :
              di_datain   : Input dataset (Req)
              di_dataout  : Output dataset (Req)
              di_dtmin    : Input datetime variable - character format
                            - YYYY-MM-DDTHH:MM:SS (Req)
              di_dtout    : Output date variable - numeric format - *DT (Req)

Input       : NA

Usage notes : If the day is missing, then the first day of the month
              If the month is missing, then the first month of the year
              If year is missing then no imputation and the date is kept
              as missing

              Assumptions/pre-requisites:
              The input dataset specified for parameter di_datain should exist.
              The input date(time) variable specified for parameter di_dtmin
              should exist in the input dataset and should be character format
              - YYYY-MM-DDTHH:MM:SS.
              Depending upon the output date variable specified (xxDT) for
              the parameter di_dtout, a flag xxDTF would be created.
              xxDTF would contain M if month is imputed, and D if only day
              is imputed.

Example     : %adamimpdtearly(di_datain  = adae_01
                             ,di_dataout = adae_02
                             ,di_dtmin   = aestdtc
                             ,di_dtout   = astdt
                             );

Programmer  :
*************************************************************************/

%macro adamimpdtearly(di_datain  =
                     ,di_dataout =
                     ,di_dtmin   =
                     ,di_dtout   =
                     );

   %put NOTE: ************************************************************;
   %put NOTE- Macro &sysmacroname. is executing.;
   %put NOTE- ************************************************************;

   *-------------------------------------------------------------------------*
   * Begin processing the date - applying date imputation rule - EARLY_DT.   *
   *-------------------------------------------------------------------------*;
   ***************************************************************************;
   data &di_dataout.;
      set &di_datain.;
      format &di_dtout. DATE9.;

      * Separate the date part out from the datetime input variable.
        Since this is just the date derivation;
      if index(&di_dtmin.,'T') then dt_part = strip(scan(&di_dtmin.,1,'T'));
```

```
      else                          dt_part = strip(&di_dtmin.);

      * As per SDTMIG, a partial datepart (YYYY or MM or DD) would contain just
        one hyphen. And hyphen is also used as a separator.
        So pre-process the hyphens, to just keep the separator hyphens;
      dt_part_1 = tranwrd(dt_part,'--','-');

      * Insert a space between two hyphens, if consecutive, indicating a missing part
        in the date;
      dt_part_1 = tranwrd(dt_part_1,'--','- -');

      * Separate out the year, month and day part of the date;
      _yr = input(strip(scan(dt_part_1,1,'-')),BEST.);
      _mo = input(strip(scan(dt_part_1,2,'-')),BEST.);
      _dy = input(strip(scan(dt_part_1,3,'-')),BEST.);

      * If all three present, then not a partial date and hence no data derivations;
      if nmiss(_yr, _mo, _dy) = 0 then do;
         &di_dtout. = input(dt_part_1,YYMMDD10.);
      end;
      else do;
         * If the year is present, then proceed with the data derivation for missing
           month or day.
           For missing year, there would be no data derivation and the date would
           remain missing.
           Since DI01 is EARLY imputation, a missing day or month would be assigned 1.
           Ensure that a flag variable is initialized for any imputation done;
         if _yr NE . then do;
            if _dy = . then do;
               _dy = 1;
               _dyfg = 'D';
            end;
            if _mo = . then do;
               _mo = 1;
               _mofg = 'M';
            end;
            &di_dtout. = mdy(_mo, _dy, _yr);

            * Set the flag variable as per ADaMIG:
              M when month is imputed, D when day is imputed
              M when both month and day are imputed;
            &di_dtout.f = coalescec(_mofg, _dyfg);
         end;
      end;
      drop dt_part dt_part_1 _yr _mo _dy _dyfg _mofg;
   run;

%mend adamimpdtearly;
```