**PhUSE 2010**

**Paper TS05**

# Unit Testing and Code Coverage Assessment with SASUnit: Key Technologies for Development of reliable SAS Macros

Patrick René Warnat, HMS Analytical Software GmbH, Heidelberg, Germany

## ABSTRACT

In order to assess the correctness of SAS macros, testing is indispensable. The concept of Unit Testing implies that tests themselves should be implemented as executable pieces of source code. As a result, tests can be repeated anytime, and negative side effects of software changes can be spotted easily. To check the correctness of SAS macros, all statements of a macro should be executed at least once during testing. This can be assessed with the Code Coverage measure, which describes the degree to which a macro has been tested.

This paper gives a general introduction as well as an update on recent developments on SASUnit; available for free from HMS Analytical Software GmbH. SASUnit is a unit testing framework for the development, execution and documentation of tests for SAS macros. A special focus of this paper is put on a new feature of SASUnit: the assessment of Code Coverage.

## INTRODUCTION

SAS® macros are used for encapsulation of SAS code which is used repeatedly in one or several other SAS programs.
An important part of the process to develop a macro is to assure that the macro implements the expected behaviour as specified by the according user requirements. A useful strategy to assure this is to test your macro thoroughly, starting in the early development phase and continuing trough the whole lifecycle of a macro, implying rigorous testing after maintenance releases, too.
This process can be streamlined with the use of the unit testing framework SASUnit. SASUnit is a unit testing framework for the development, execution and documentation of tests for SAS programs. Utilizing SASUnit, tests for a SAS macro are implemented as executable SAS source code and structured by test scenarios. In a SASUnit test scenario, you allocate static test data, call the program under test and then call assertion macros in order to compare actual with expected outcome. SASUnit consists of a set of SAS macros that build a framework to run your test scenarios and automatically build a report to document the executed tests and their results.
Big advantages of using SASUnit are that tests can be repeated easily anytime, and that tests implemented as source code extend the specification of user requirements in a formal way. In addition, developers can leverage SASUnit for automatic and standardized documentation of test results.

While writing unit test for your macros, it is useful to assure that all source code statements are executed at least in one test case. By doing so, it is avoided that parts of your source code remain untested, potentially containing undetected programming errors. The process to identify the parts of your source code that are executed during your tests is the determination of the so called code coverage of your tests. In this paper, the idea is presented to extend the SASUnit framework in order to optionally determine which statements in a tested macro have been covered by executed test cases.

The following paragraphs first give a more detailed introduction of SASUnit for those readers not yet knowing SASUnit. Next, a concept is presented describing how the assessment of code coverage could be integrated into SASUnit.

## SASUNIT

SASUnit is implemented as a set of SAS macros and a few operating system shell commands. SASUnit is open source software and available at sourceforge.org [1]. The benefits of SASUnit can be summarized as follows:

- Write test scenarios and test cases for SAS programs

- Use assertions to test the values of macro variables, the contents of SAS data sets, the existence of external files or the presence or absence of log messages
- Run tests and generate test documentation in batch mode
- Get clearly arranged test documentation, integrated with program documentation
- Measure performance of SAS programs
- Integrate non-automatic tests, for instance visual checks of report output
- Written purely on the basis of SAS macros and a few shell commands
- Available for SAS 8.2, 9.1 and 9.2

The usage of SASUnit is depicted in figure 1:

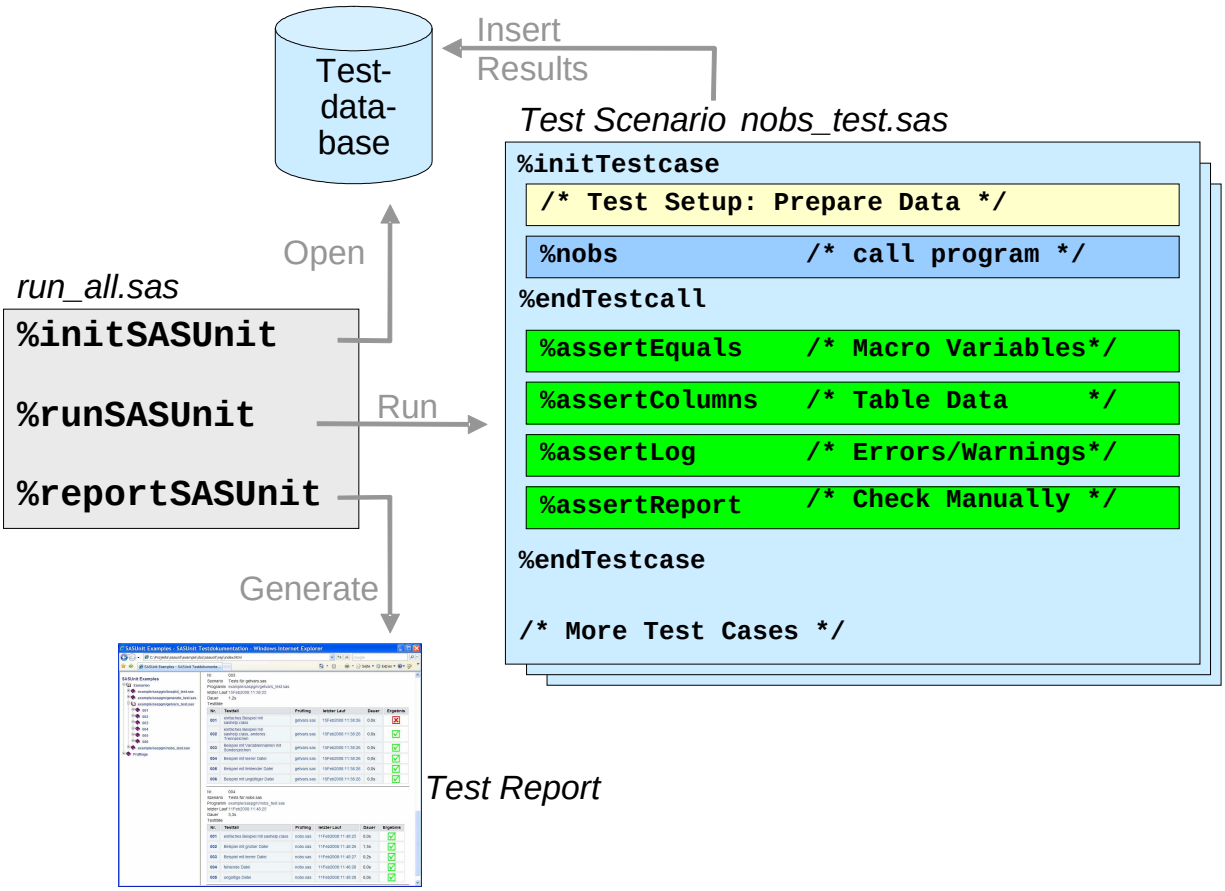


Figure 1: Overview of SASUnit usage (diagram taken from [2])

A test suite consists of one or more test scenarios. A test scenario is an executable program written in the respective programming language and contains a set of test cases. Each test case tests a certain aspect of the functionality of a unit under test. Many test cases might be needed to test complete functionality.
SASUnit is controlled by a program usually called run_all.sas, which represents a test suite. Within a test suite definition, the macro %initSASUnit creates or opens the test repository. %runSASUnit runs each test scenario (represented by different SAS program files) in its own SAS session. As the final step in a test suite definition, %reportSASUnit creates the report from the test repository.
A test case needs static data so that it can be repeated at any time. Static data can be stored in or referenced from the testing environment, or it can be generated in the test setup section of a test case. Every test case contains one execution of the unit under test with a certain constellation of parameters and the usage of the static test data. At the end of each test case, a set of assertions check for the correctness of the results delivered by the execution of the unit under test. Each assertion can have the outcome “OK” (signalled by green) or “failed” (red).

More detailed information on SASUnit can be found in [2].

## CODE COVERAGE OF UNIT TESTS

#### WHAT IS IT ABOUT?

The assessment of the source code coverage of unit tests is a formal approach to determine how well a unit of source code is tested. Code coverage analysis helps to assess the quality of a set of test cases. It is a way of white box testing, as code coverage assessment regards the source code structure of a tested software unit (opposed to black box testing, where the inner structure of a software unit is not of interest, only the results it produces for given input).

Different approaches have been defined to measure the code coverage of unit tests. Examples are:

- statement coverage: describes whether every statement has been executed during the tests
- decision coverage: assesses whether all Boolean expressions tested in control structures (such as the if-statement) evaluated to both true and false during tests
- path coverage: evaluates whether all logically possible execution branches have been evaluated during the tests.

For a detailed description and discussion of different approaches to code coverage assessment, please refer to [3].

#### WHY IS IT USEFUL?

Analysis of code coverage can identify parts of a unit of source code that are never executed in the associated test cases. As your unit tests are a way to assure good quality of your source code, code coverage analysis of your test cases is a tool to support you in assessment of the quality of your test cases. Using code coverage analysis, it is easily avoided that parts of your source code remain untested, potentially containing undetected programming errors.

#### HOW CAN IT BE IMPLEMENTED AS AN EXTENSION TO SASUNIT?

Statement coverage was chosen to be integrated in SASUnit, as it is straightforward to implement and as it is useful to identify code not executed during unit tests. The basic principle of the implementation is to scan the source code of the unit under test for blocks of source code that are only executed when a certain condition is met. Then, the source code is modified in order to insert macro calls that are used to signal whether a block of code has been executed or not during execution of test cases. Figure 2 depicts the principle.

The following strategy was chosen in order to implement the assessment of the code coverage of a set of test cases. First, at the start of a test scenario, the source code of the tested macro is scanned in order to identify blocks of (macro-)code like '%if %then %do' blocks (lines only containing comments are ignored during the scan). The line numbers of the start and end of all blocks of code are saved to a SAS table.

Next, the source code is modified by insertion of a marker macro tagging the start and end of code blocks. In order to identify the statements executed during a test case, it is not necessary to mark all statements, but only the statements deciding whether a sequence of consecutive statements is executed or not.

After insertion of the marker macro calls, the modified source code is used instead of the original code for execution of the associated test cases. While execution of the test cases, calls to the marker macro will be made, according to whether the blocks of code are executed in the test cases or not. The marker macro simply stores (in a list saved to a global macro variable) the line number at which it is called in the source code of the unit under test. Thus, after the execution of a test case, it can be determined which blocks of code have been executed during the test case.

Finally, the intersection of all blocks of code called in all test cases is determined and this result is compared with the set of all existing blocks of code found in the initial scan of the macro source code. Thus, it is possible to assess whether all blocks of code (and thus all statements) have been called at least once during a test case. Now, a HTML representation of the tested source code can be prepared, highlighting the statements of those blocks of code never executed during the runs of the test cases.

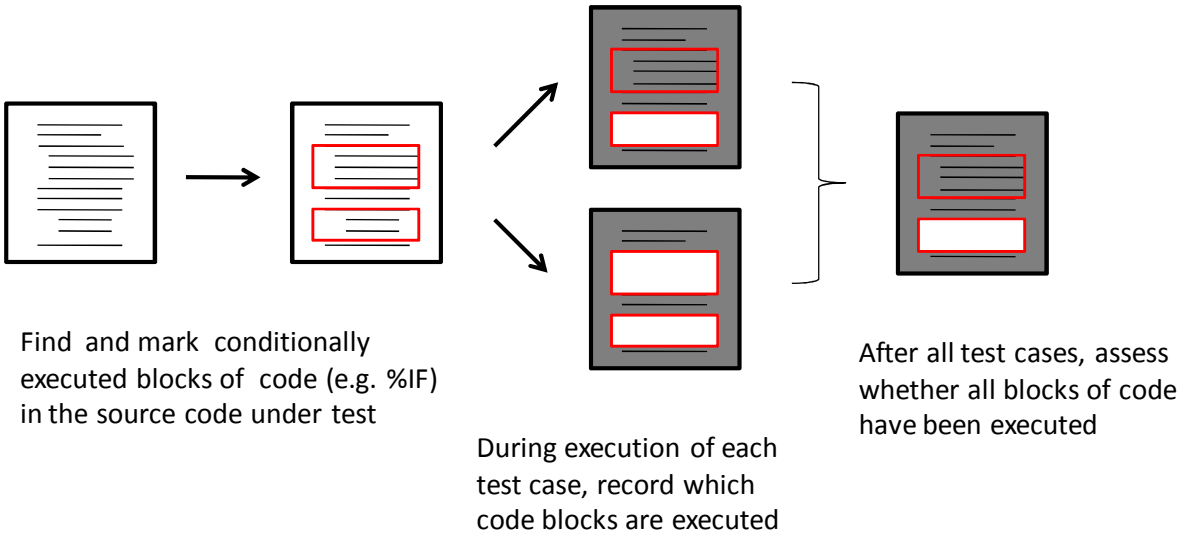


Figure 2: the basic principle of the statement coverage assessment implementation as presented in this paper.

**EXAMPLE**

A very simple example macro is listed below:

```
/*Code Coverage Test Macro*/
%MACRO ccTestMacro1(binaryInput);
   %LOCAL printTxt;
   %LET   printTxt = A value not equal to 1 was given.;
   %IF &binaryInput EQ 1 %THEN %DO;
      %LET printTxt = A value equal to 1 was given.;
   %END;
   %PUT &printTxt;
%MEND  ccTestMacro1;
```

When this macro is scanned for blocks of code, two blocks are detected: the 'macro-mend' block and the 'If-then-do-end' block.

The resulting modified macro code contains calls to a marker macro:

```
/*Code Coverage Test Macro*/
%MACRO ccTestMacro1(binaryInput);  %markBlockBorder(g_marker_Memory, 2 )
%LOCAL printTxt;
%LET   printTxt = A value not equal to 1 was given.;
%IF &binaryInput EQ 1 %THEN %DO;  %markBlockBorder(g_marker_Memory, 5 )
%LET printTxt = A value equal to 1 was given.;
%END;  %markBlockBorder(g_marker_Memory, 7 )
%PUT &printTxt;
%MEND  ccTestMacro1;  %markBlockBorder(g_marker_Memory, 9 )
```

The macro %markblockBorder simply appends a line number to a macro variable with the name g_marker_memory. After a run of a test case, the line numbers stored in g_marker_Memory are evaluated. Using the information which blocks of code exist in the source code, it is determined, which of these blocks have been executed.

Finally, after execution of all test cases for a unit under test, this information is used to determine which statements have been called at least in one test case.

If, for example, only a test case exists with the following call to the macro:

```
%ccTestMacro1(0);
```

Then a report with the source code will be generated with the if-then-do block marked with orange font, as the condition of the if-block is not met and the associated statements are not executed:

```
/*Code Coverage Test Macro*/
%MACRO ccTestMacro1(binaryInput);
```

```
%LOCAL printTxt;
%LET printTxt = A value not equal to 1 was given.;
%IF &binaryInput EQ 1 %THEN %DO;
%LET printTxt = A value equal to 1 was given.;
%END;
%PUT &printTxt;
%MEND ccTestMacro1;
```

If a test case with the macro call `%ccTestMacro1(1)` is added, all statements (expect lines only containing comments) will be colored in green.

#### DISCUSSION

The implementation of code coverage as a feature of SASUNIT is currently at an experimental stage. More development work has to be done. At the moment, only code blocks defined by macro statements are supported, the assessment of data step logic might possible with the use of the Call Symputx command.
Although the integration of code coverage in SASUNIT is still work at progress, first results suggest it to be a useful help in the assessment of the quality of test cases. Thus, this new feature might be released as part of the SASUnit macro framework in near future.

## CONCLUSION

Using SASUnit may require minimal more effort during the initial development phase of your SAS macros, but it will save you a lot of time during development of maintenance/change releases due to the possibility of automated regression tests and (in near future) the code coverage assessment. In addition, the documentation of your tests is generated automatically by SASUnit, this feature alone is worth the effort to integrate SASUnit in your development process.

## REFERENCES

[1] SASUnit download on sourceforge.org, see http://sourceforge.net/projects/sasunit/.
[2] Mangold A., Warnat, P.: Automatic Unit Testing of SAS Programs with SASUNIT. Paper RA07, PhUSE 2008.
[3] Wikipedia: Code Coverage, http://en.wikipedia.org/wiki/Code_Coverage, retrieved 2010-09-01.

## CONTACT INFORMATION

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

Dr. Patrick R Warnat
HMS Analytical Software GmbH
Rohrbacher Str. 26
69115 Heidelberg
Germany
Fax: +49 6221 60 51 - 0
Email: patrick.warnat@analytical-software.de
Web: http://www.analytical-software.de