Paper CT10

When the Answer to Your Problems Is Just One Call (execute) Away

Luis Osorio, Chrestos Concept GmbH & Co. KG, Essen, Germany
Dr. Lara Suckut, Chrestos Concept GmbH & Co. KG, Essen, Germany

ABSTRACT

Recently, many pharmaceutical companies have reported a shift from traditional to metadata-based data structures and programming. This calls for a dynamic, data-driven programming style, which is uniquely represented in SAS® programming by the function "call execute". However, to date, this function has not received the recognition or utilization it deserves. We set the stage for this versatile tool, as learning its framework opens up a wide range of applications using pre-existing coding skills. We highlight the power of data-driven code generation and delayed execution from a data step and place it in the context of macro programming. We also advise on common pitfalls and support each application with explicit examples. We hope to leave the audience with a firm grasp of the opportunities and the ability to make use of this neglected function to support the current shift in the industry.

INTRODUCTION

More and more companies in the pharmaceutical industry are changing their infrastructure to more flexible databases, utilizing metadata to aid in the generation and quality control of study data. This shift requires dynamic, data-driven programming. A versatile tool to help in this endeavor is the function "call execute" [https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/mcrolref/n1q1527d51eivsn1ob5hnz0yd1hx.htm].

Call execute can be used as part of a data step and combines data set variable values with traditional SAS code, e.g. data steps or proc SQL to generate code for subsequent execution. This allows for very flexible code creation. The framework itself is easy to learn and can be combined with any personal coding style preferences. Additionally, the timepoint of execution of this generated code can be influenced.

The use of dataset variable values gives more control than using a myriad of macro variables. Additionally, actual data can be combined with metadata information for formatting, subsetting or conditional generation and information can be efficiently streamlined.

Here, we want to give an introduction to the framework of call execute and common pitfalls to avoid. We will then present examples of its application.

THE FRAMEWORK OF CALL EXECUTE

Call execute can be invoked with

```
Call execute(<argument>);
```

The argument can consist of:

- a character string in quotation marks. This character string is then executed as SAS code. Single quotation marks lead to delayed execution during program execution, while double quotation marks result in immediate execution during data step construction. The SAS code can be any valid SAS code, from data steps to macro invocations to proc SQL.
- a dataset variable without quotation marks. This variable can contain a character string to be executed as SAS code or data values to be concatenated with other character strings into a character expression to be executed as SAS code.

Call execute is invoked within a data step and therefore can use any dataset variables loaded within this data step. It can also be integrated into common SAS data step constructs like if clauses and loops. Alternatively, input variable values can be constructed following certain conditions or repetitions.

The data step around call execute can be of type data _NULL_ purely for the call execute construction or can be saved for accountability or debugging purposes.

EXECUTION ORDER

Call execute offers the manipulation of the execution time and order as another feature. This can be achieved by keeping the following in mind:

Macro invocations and variables are resolved immediately, while SAS statements within invoked macros only execute after a step boundary. Alternatively, the macro quoting function %NRSTR can be used to mask the macro statement, so that it is resolved after a step boundary as well. Macro invocations by call execute which produce SAS code are also resolved immediately, but the resulting SAS code is then executed after the data step boundary. [PhUSE 2009, Paper CC01, Can I CALL EXECUTE here? Vinod Kaippillil Mukundaprasad, Roche, Welwyn, UK]

SAS code without macro triggers is processed in the order of entry. Single quotation marks around a character string in the argument for call execute lead to execution during program execution, while double quotation marks lead to execution during data step construction. In practice, the difference in quotation marks affects only macro calls.

The following program example demonstrates its usage and pitfalls (colors link call execute code to log output):

```
data callexe;
       var1="testing";
       call execute("data test;");
       call execute('put "Testing "');
       call execute('"different use cases."');
       call execute(';');
       call execute('var1="start";');
       call execute('put "Single quotation marks :'||var1||'";');
       call execute("put 'Double quotation marks :"||var1||"';");
       call execute("%nrstr(%put Masked macro call with macro quoting in double
       quotation marks for )"||var1||";");                                          3
       call execute('%nrstr(%put Masked macro call with macro quoting in single
       quotation marks for '||var1||');');
       call execute('%put Macro call without macro quoting in single quotation marks
       for '||var1||';');                                                           4
       call execute("%put Macro call without macro quoting in double quotation marks
       for "||var1||";");                                                           2
       call execute('var1="end";');
       call execute("run;");
       var1="done";
run;
```

The log then states:

```
 70         data callexe;
 71         var1="testing";
 72         call execute("data test;");
 73         call execute('put "Testing "');
 74         call execute('"different use cases."');                    } 1
 75         call execute(';');
 76         call execute('var1="start";');
 77         call execute('put "Single quotation marks :'||var1||'";');
 78         call execute("put 'Double quotation marks :"||var1||"';");
 79         call execute("%nrstr(%put Masked macro call with macro quoting in double
quotation marks for )"||var1||";");
 80         call execute('%nrstr(%put Masked macro call with macro quoting in single
quotation marks for '||var1||');');
 81         call execute('%put Macro call without macro quoting in single quotation
marks for '||var1||';');
 82         call execute("%put Macro call without macro quoting in double quotation
marks for "||var1||";");
 Macro call without macro quoting in double quotation marks for "||var1||"   2
 83         call execute('var1="end";');
 84         call execute("run;");
 85         var1="done";
 86         run;

 86       !
 Masked macro call with macro quoting in double quotation marks for testing  3
 Macro call without macro quoting in single quotation marks for testing      4

NOTE: The data set WORK.CALLEXE has 1 observations and 1 variables.

 NOTE: CALL EXECUTE generated line.
 1          + data test;
 2          + put "Testing "
 3          + "different use cases."
 4          + ;
 5          + var1="start";
```

```
 6          + put "Single quotation marks :testing";
 7          + put 'Double quotation marks :testing';
 8          + %put Masked macro call with macro quoting in single quotation marks for
testing;
 Masked macro call with macro quoting in single quotation marks for testing
 9          + var1="end";
 10        + run;

 Testing different use cases.
 Single quotation marks :testing
 Double quotation marks :testing
```

Afterwards, the output data set callexe has variable var1 with value “done”. However, for call execute, the value “testing” is used, as this was the value during the call.
The output data set test has variable var1 with value “end”. This variable was constructed during the call and has no relationship with the variable var1 in the data set callexe.

PITFALLS
The example also highlights how to avoid common pitfalls.
One issue is that unbalanced quotation marks need to be avoided. For this, text strings within call execute arguments need to be framed in the alternative set of quotation marks not used for the argument itself. If the argument exceeds 256 characters, then it has to be split over several lines unless the option noquotelenmax is activated (1).
Another issue is that the order of argument execution needs to be considered. A macro call in double quotation marks is executed before the data step is executed, therefore can not use the var1 variable here (2). Using single quotation marks (3) or masking the macro call (4) will resolve the issue. Also remember that when calling macro variables as text in a resulting SAS code, double quotes are needed for execution.
Additionally, while most arguments can be separated over several lines, comments within /**/ must be in the same argument [Paper 027-30 CALL EXECUTE: A Powerful Data Management Tool, Denis Michel, Johnson & Johnson Pharmaceutical Research and Development, L.L.C].

POWER-UP TOOLS FOR CALL EXECUTE

Due to the possibility of using variables of a dataset in proc sql and data steps, CALL EXECUTE is very useful when used together with metadata and dictionaries.

According to the SAS documentation, a dictionary is a read-only SAS view table that contains information about SAS libraries, SAS datasets, SAS macros, and external files that are in use or available in the current SAS session [https://documentation.sas.com/doc/en/lrcon/9.4/p13650yz8q04csn1m2xojl26ljsc.htm]. These metadata are distributed to specific dictionary tables. One of these DICTIONARY tables also contains the settings for SAS system options that are currently in effect.

Call execute, in combination with metadata, permits reading variable specifications – such as formats and code for variables derivation - to create variables automatically. The dataset information in the **dictionaries** allows for execution of dynamic code for example for all data sets inside a specific library.

EXAMPLE 1: IMPLEMENTATION WITH LIBRARIES

According to CDISC guides, during a submission it is necessary to include all codelists for the variables used. One possible application of call execute and dictionaries is loading all the formats required in a SAS session assuming that we have saved all necessary codelists in a library called “codelist”.

By the creation of the library “codelist”, SAS automatically includes the information of all the members of the library in the dictionary called “MEMBERS”.

This information can then be read from “MEMBERS(where=(libname=”codelist”))” and using call execute in a data step where this dataset is set, we can automatize the creation of all the formats required for the study.

The following code would then be an example of how to use call execute to get the desired formats in the SAS session:

```
 1   PROC SQL;
 2       create table codlst as select *
 3       from dictionary.members
 4       where libname="codelist";
 5   END;

 6   DATA _NULL_;
 7       set codlst;
 8       CALL EXECUTE("proc format library=codelist
 9       cntlin=" || memname || "run;" );
10   RUN;
```

Notice that in lines 1) to 5) the information of the codelist names is saved in the data set "codlst". The name of each file with the codelist information is then passed to the SAS session through CALL EXECUTE in lines 7 to 10.

EXAMPLE 2: COMPUTE MEANS OF MULTIPLE PARAMETERS

We show an example of using call execute to compute the mean of multiple parameters (unknown concrete name and dataset name) at screening by sex for each population (ITT and safety here). As input, we take ADaM datasets, here advs and adsl.
We need to transpose advs to get the paramcd variables and aval results as parameters.
We then use call execute to invoke proc sql to list all variables ending with "bl" or containing "Blood Pressure" in the variable name as well as the respective datasets containing these variables. We also include visit, population flag variables and sex for later filtering and selection. All variables going into call execute are saved in the dataset call_exec_stats for later reuse and accountability.

```
data call_exec_stats;
      pop="ITT|SAF";
      by_param="sex";
      params=":bl|:Blood_Pressure:";
      call execute("proc sql noprint;");
      call execute("create table indata as");
      call execute("select distinct memname,catx('.',memname,name) as varname from
      dictionary.columns");
      call execute("where lowcase(libname) eq 'work' and lowcase(memname) like 'ad%'
      and not(lowcase(memname) eq 'advs')");
      call execute(" and (lowcase(name) eq 'subjidn' or lowcase(name) eq 'visit' or
      lowcase(name) eq "||quote(lowcase(by_param)));
      do __z_j=1 to countw(pop,"|");
            call execute("or lowcase(name) like
            '"||lowcase(strip(scan(pop,__z_j,"|")))||"fl'");
      end;
      do __z_i=1 to countw(params,"|");
            call execute("or lowcase(name) like
            '"||transtrn(lowcase(scan(params,__z_i,"|")),':','%')||"'");
            if __z_i eq countw(params,"|") then call execute(");");
      end;
      call execute("quit;");
run;
```

We then merge all datasets and select the parameters from the list and filter for Screening visit.

```
data call_exec_stats2;
      set indata end=eof;
      length params vars datasets $500;
      retain params;
      retain datasets;
      retain vars;
      if index(lowcase(vars),strip(scan(lowcase(varname),2,'.'))) ne 0 or
      lowcase(strip(scan(varname,2,'.'))) eq "visit" or
      prxmatch('/.*fl$/i',lowcase(strip(scan(varname,2,'.')))) then vars=vars;
```

```
        else vars=catx(' ',lowcase(vars),lowcase(scan(varname,2,'.')));*prevent
        duplicated variables;
        params=ifc(index(lowcase(params),strip(scan(lowcase(varname),2,'.'))) ne 0,
        params, catx(',',params,varname));*prevent duplicated variables;
        datasets=ifc(index(lowcase(datasets),strip(lowcase(memname))) ne 0, datasets,
        catx(',',datasets,memname));*prevent duplicated datasets;
        if eof then do;
               call execute("proc sql noprint;");
               call execute("create table inds as");
               call execute("select "||params||" from ");
               do __z_i=1 to countw(datasets,",");
                     if __z_i eq 1 then do;
                           call execute(scan(datasets,__z_i,","));
                           if index(params,scan(datasets,__z_i,",")||".visit") ne 0
                           then call execute("(where=(visit='Screening'))");
                     end;
                     else do;
                           call execute(" inner join "||scan(datasets,__z_i,","));
                           if index(params,scan(datasets,__z_i,",")||".visit") ne 0
                           then call execute("(where=(visit='Screening'))");
                           call execute("on
                           "||strip(scan(datasets,__z_i,","))||".subjidn eq
                           "||strip(scan(datasets,__z_i-1,","))||".subjidn");
                     end;
                     if __z_i eq countw(datasets,",") then call execute(";");
               end;
               call execute("quit;");
               output;
        end;
run;
```

Finally, we calculate the means per sex per population for all variables selected.

```
data call_exec_stats3;
        set call_exec_stats2; set call_exec_stats(drop=params);
        do __z_i=1 to countw(pop,"|");
               call execute("ods output summary=sexmean"||scan(pop,__z_i,"|")||";");
               call execute("proc means data=inds(where=("||scan(pop,__z_i,"|")||"fl eq
               'Y')) n mean std median min max printalltypes descendtypes StackODSOutput
               maxdec=2;");
               call execute("class "||by_param||";");
               call execute("var "||transtrn(lowcase(vars),lowcase(by_param),'')||";");
               call execute("title Mean per sex for "||scan(pop,__z_i,"|")||";");
               call execute("OUTPUT out=sexmean n=N mean=Mean std=SD median=Median
               min=min max=max;");
        end;
run;
ods output close;
```

Advantages of using call execute in this case are, that the exact dataset and variable names or numbers do not have to be known in advance, selection based on wildcards for variables is possible. Also, the calculation can be performed for any number of populations or other parameters, with no need to repeat code.
This example also shows how to use proc sql within call execute and reuse the generated dataset from the data step around call execute, which is also helpful for debugging and traceability.

OTHER USE CASES

Apart from the example given here making use of metadata, there are many other applications where call execute comes in handy. Since discussing these in detail would be beyond the scope of this paper, we will simply give some suggestions and recommended resources for further reading.

- SAS has published a blog post on using call execute for data driven programming [https://blogs.sas.com/content/sgf/2017/08/02/call-execute-for-sas-data-driven-programming/]. The post highlights the role of call execute in repeating tasks for different datasets based on a metadata dataset listing all sas datasets.

- Usage of SAS dictionary tables in a more extensive manner has been explored in a previous Phuse [Accessing SAS® metadata and using it to help us develop data-driven SAS programs. Iain Humphreys, 2009].
- Another useful post shows how to create dynamic macro calls with varying arguments or parameters, generate conditional data processing steps, or dynamically create SAS code based on variable values [https://www.listendata.com/2016/02/sas-call-execute-made-easy.html]. Especially the dynamic macro calls are an often-used application [Call Execute: Let Your Program Run Your Macro. Artur Usov, 2014].
- In general, call execute is useful whenever information from different data sources needs to be connected [CALL EXECUTE: A Powerful Data Management Tool. Denis Michel, 2005]. For example, call execute can be used for data evaluation, e.g. setting flags if a certain range from another dataset is met [Creating Metadata-Driven Program Logic with the SAS Call Execute Routine - An Example with CTC Grading of Laboratory Results. Robert W. Graebner, 2011]. It is also possible to bring in comments from one Excel file to a dataset [Creating Data-Driven SAS® Code with CALL EXECUTE. Hui Wang, 2015].

Outside of source, SDTM, or ADaM datasets, call execute can also be useful for combining information from custom datasets like protocol generation tools or custom databases, especially if they follow a one source- multiple applications approach for their data.

CONCLUSION

We have introduced the functionality and possibilities of the function call execute, and provided a starting point for further reading. We hope that this will aid in navigating modern programming and data structures, and that the function finds its deserved place among the valued tools SAS provides.

REFERENCES

SAS® Programming Documentation, 2024:
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/mcrolref/n1q1527d51eivsn1ob5hnz0yd1hx.htm

https://documentation.sas.com/doc/en/lrcon/9.4/p13650yz8q04csn1m2xojl26ljsc.htm

https://blogs.sas.com/content/sgf/2017/08/02/call-execute-for-sas-data-driven-programming

Can I CALL EXECUTE here? Vinod Kaippillil Mukundaprasad, 2009

CALL EXECUTE: A Powerful Data Management Tool. Denis Michel, 2005

Accessing SAS® metadata and using it to help us develop data-driven SAS programs. Iain Humphreys, 2009

Call Execute: Let Your Program Run Your Macro. Artur Usov, 2014

https://www.listendata.com/2016/02/sas-call-execute-made-easy.html

Creating Metadata-Driven Program Logic with the SAS Call Execute Routine - An Example with CTC Grading of Laboratory Results. Robert W. Graebner, 2011

Creating Data-Driven SAS® Code with CALL EXECUTE. Hui Wang, 2015

CONTACT INFORMATION

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

Author Name: Luis Osorio
Company: Chrestos Concept GmbH & Co. KG
Address: Girardetstraße 1-5, D-45131 Essen
Email: luis.osorio@chrestos.de
Website: www.chrestos.de