MACUMBA — a modern SAS GUI — debugging made easy

Michael Weiss

Bayer Pharma AG, Berlin, Germany

MACUMBA is an application for SAS® programming developed in house at Bayer. It combines the interactive development features of SAS for Windows, the possibility of a client — server environment and unique 'state of the art' features that are missing in other SAS development environments. This paper covers some of the unique features that are related to SAS code debugging. This paper will begin by comparing the systems special code execution modes with the way this would be performed in interactive SAS. This paper will continue by presenting an overview of the graphical implementation of the single step debugger for SAS macros and DATA Steps. Finally, this paper will highlight the main issues faced during the development of MACUMBA.

Keywords: Integrated development environment, User interface, Code execution, Debugger, Macro

Introduction

The MACUMBA development was started in 2007 as a graphical implementation of the SAS DATA Step debugger. Step by step (beside the daily work) many other features were added to support SAS program and macro development, data examinations and a much more. MACUMBA has been used by most of the SAS users in our company for their 'daily SAS work' for at least the last 3 years.

MACUMBA is so flexible and intuitive that it can be used for any SAS development purpose, though, as it has been developed within a pharmaceutical company, it does have features that are particular to the pharmaceutical industry and to the way we do things at Bayer.

The application is implemented in pure Java and uses the SAS IOM technology to access a SAS Object Spawner for 'interactive like' SAS code execution and data access. MACUMBA's user interface (Fig. 1) is intended to be similar to other common integrated development environments's like Eclipse.

MACUMBA uses 'views' to group functionalities or tasks. These views are grouped into 'view containers' which can be arranged by the user.

Currently, over 25 views are implemented. These include general purpose views like the 'Explorer View', SAS specific views like 'Data Set View' or 'SAS Editor View' and task specific views like the 'Download Study View' or the 'My Studies View'.

Correspondence to: Bayer Pharma AG, Muellerstr. 178, P300, 13353 Berlin, Germany. Email: michael.weiss@bayer.com

The views operate independently, though in some cases actions in one view trigger results in other views. For example executing SAS code in an Editor sends the SAS log and output to the corresponding 'Log View' and 'Output View'.

Special Code Execution Modes

The following special execution modes are provided by MACUMBA and explained in this paper:

- 'Run to Line', 'Run from Line' and 'Run Step' — execute a specific part of the program;
- 'Run in Template' — allows the definition of a custom code template that encloses the executed code;
- 'Run as Macro' — allows the execution of some code lines as a temporary SAS macro;
- 'Resolve Code' — resolves all macro code and provides the real executed SAS code by using MPRINT.

'Run to Line', 'Run from Line': the border patrol

'Run to Line' executes the code between start of the program and the current line by a simple click or keyboard shortcut.

'Run from Line' is the counterpart of Run to Line and executes the code from the current line until the program end.

This is especially useful when developing large programs, where the developer would need to scroll up and down a lot to highlight the code to be run, before submitting the code for execution.

'Run Step': a smart execution helper

'Run Step' allows the developer to run a particular part of a program (e.g. DATA Step, Macro) without the need to highlight the whole section of code in question.

Published by Maney
DOI 10.1179/1757092112Z.00000000012

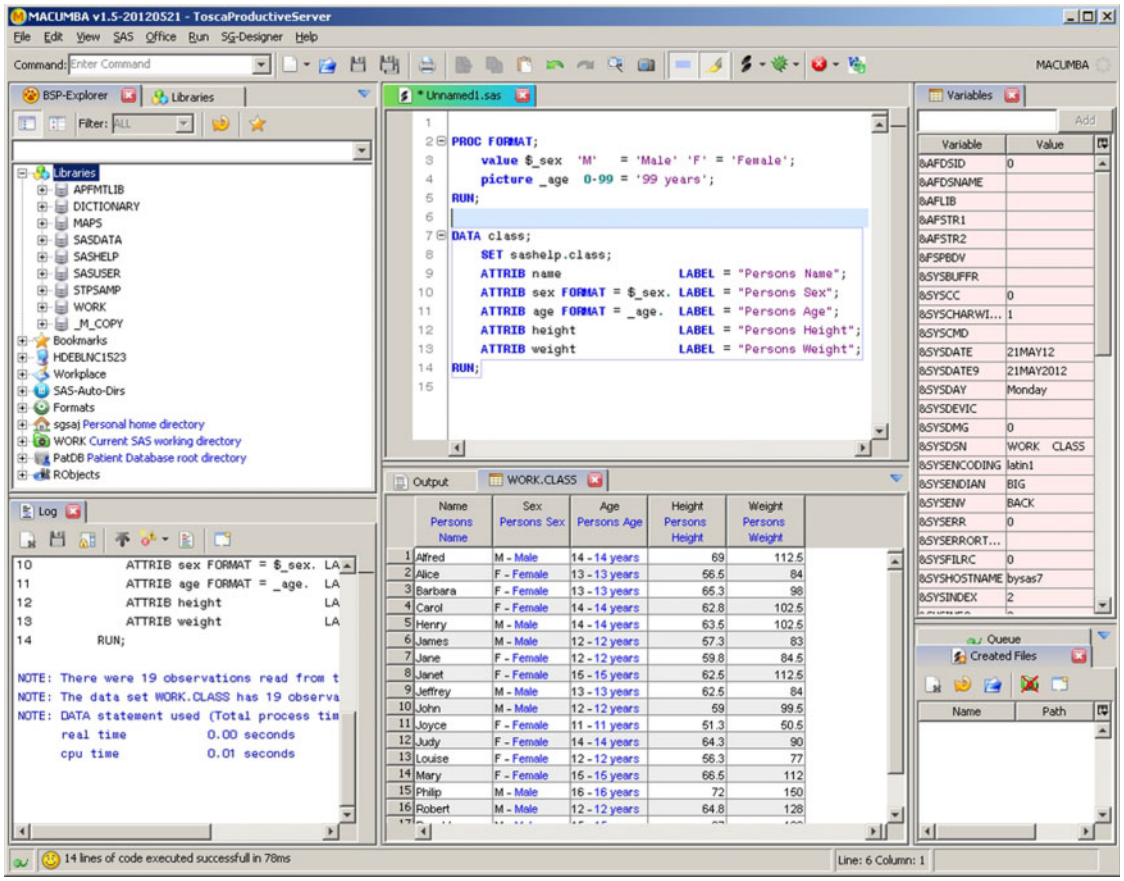

Figure 1 MACUMBA screenshot

When invoked, a dialog window (Fig. 2) appears. This window provides a list of possible steps to execute. For example, if invoked inside of a DATA Step, the entry 'DATA Step' is available. If invoked inside of a macro definition the '%MACRO <name> Definition' entry is available and so on. This provides a simple way to execute a section of code without the need to correctly select it.

Run Step works in the same way when using PROC SQL, by surrounding the SQL statement by 'PROC SQL' and 'QUIT'. This way individual parts of a long multistepped SQL procedure can be executed.

'Run in Template': execution with custom template

Run in Template allows the developer to define code that can be used to surround a section of a program when running it. This is particularly useful when developing programs that create a big RTF file with multiple tables. By defining an ODS RTF template (Fig. 3) in MACUMBA any section of the program can be executed with the ODS RTF code surrounding it. This allows, for example, layout adjustments for single tables without the need to put temporary code, like that shown below, into the program.

```
ODS RTF ...;
PROC REPORT ...; * Create first table;
RUN;
* Do some other stuff;
ODS RTF ...;
PROC REPORT ...; * Create second table;
RUN;
ODS RTF CLOSE;
* Do some other stuff;
PROC REPORT ...; * Create last table;
RUN;
ODS RTF CLOSE;
```

In this program, the marked code is added only for development of the second table. This code has to be removed in the final version. In these cases, it is always possible that this temporary code is not removed correctly. This may lead to unexpected results and may be hard to find later on in the development process.

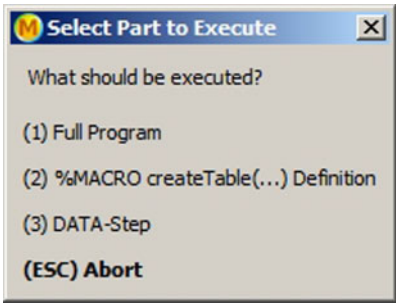

Figure 2 Run Step dialog window

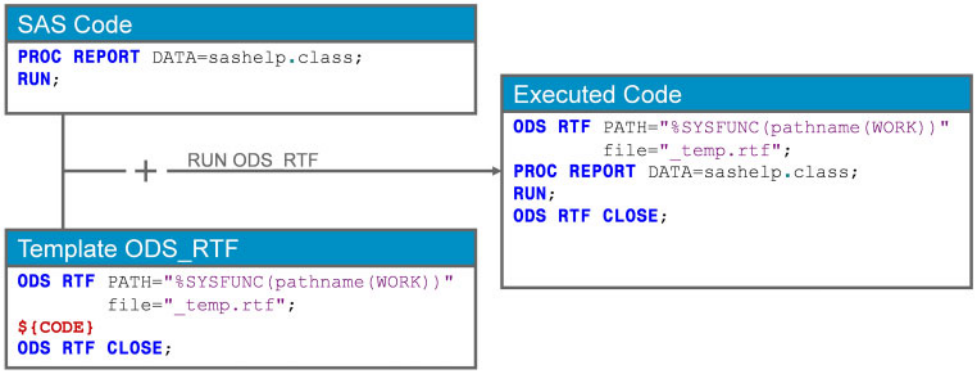

Figure 3 'Run in Template' example

'Run as Macro': partial macro execution

'Run as Macro' checks the selected code for surrounding %MACRO definitions and displays a dialog that allows a custom initialization of the macro parameters and required macro variables. A temporary macro is then created and executed based on the selected macro code and the entered information (Fig. 4).

This is particularly useful when developing, testing and debugging macros and the code that needs to be executed contains macro commands like %IF or %DO.

For example if from the following macro:

```
%MACRO abc(param1=Y, param2=N);
    %PUT A lot of stuff is to be done;
    %LOCAL i;
    %DO i=1 %TO &max.;
        %IF %SYSFUNC(mod(&i., 2)) EQ 0
        %THEN %DO;
            %PUT &param1. &param2.
            (i=&i.);
        %END;
    %END;
    %PUT a lot more stuff is to be done;
%MEND;
```

The following part is to be executed separately:

```
%DO i=1 %TO &max.;
    %IF %SYSFUNC(mod(&i., 2)) EQ 0
    %THEN %DO;
        %PUT &param1. &param2.
        (i=&i.);
    %END;
%END;
```

Then, the following error message will be issued:

```
ERROR: The %DO statement is not valid in
open code.
```

Additionally, the following warning is issued, in case the macro variable param1 does not exist:

```
WARNING: Variable param1 can not be
resolved;
```

The common solution would be to modify the macro code accordingly, execute it and undo the modifications afterwards. This is time consuming and could cause issues if the modifications are not undone correctly.

'Resolve code': a graphical MPRINT implementation

When talking about SAS code debugging, it is hard to not directly have the MPRINT system option in mind. Almost every SAS programmer knows the MPRINT feature and knows that often the output can be more confusing than helpful. To get the MPRINT output without copying and pasting it line by line from the log, the MFILE option can be used to send it directly into a new file. This provides a

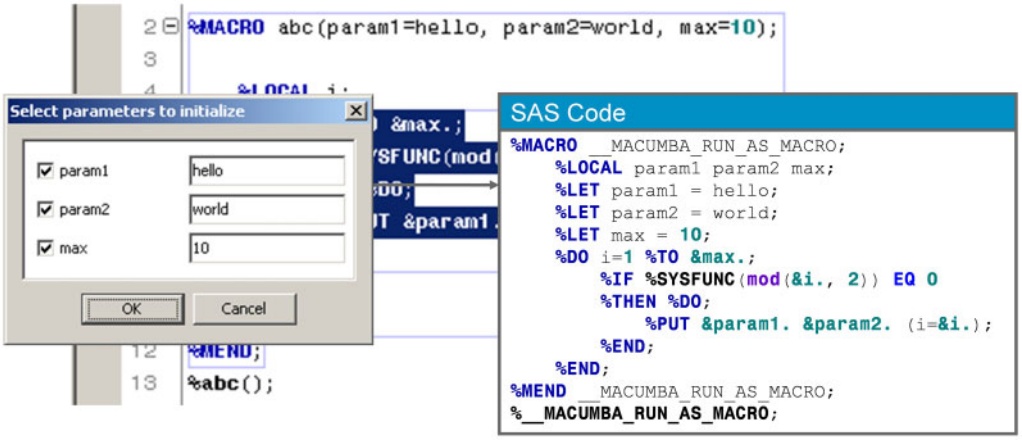

Figure 4 'Run as Macro' overview

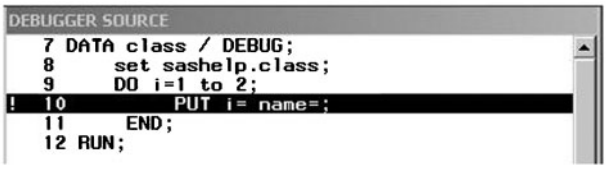

Figure 5 SAS DATA Step debugger

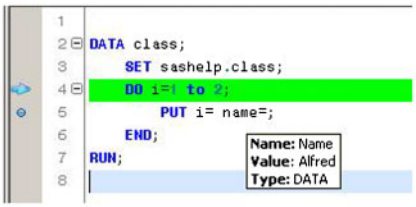

Figure 6 MACUMBA DATA Step debugger

handy way to debug SAS code that is generated by a macro call. If for example the SAS code that is generated by the following macro call is to be debugged:

```
%doStudyEvaluation(studyId=0815,
projectId=4711);
```

The following code would be executed:

```
FILENAME mprint
"%SYSFUNC(pathname(WORK))/resolved.
sas";
OPTIONS MPRINT MFILE;
%doStudyEvaluation(studyId=0815,
projectId=4711);
OPTIONS NOMPRINT NOMFILE;
FILENAME mprint;
```

In MACUMBA this could be achieved by applying an appropriate template and using the 'Run in Template' feature. The developer would still need to manually navigate to the temporary directory to open the file created.

MACUMBA provides a one-step solution in the form of the 'Resolve Code' execution mode. When using 'Resolve Code', all the manual steps are done automatically and additionally a source code formatter reformats the MPRINT output, to add correct indention, empty lines and so on, before it is opened. The result is a more readable and helpful output for use in code debugging.

Graphical DATA Step Debugger

The DATA Step debugger is one of the least well known features of SAS. It has been available since the release of SAS 6.11, it is very useful when debugging a complicated DATA Step.

For example the SAS DATA Step debugger provides the following features:

- single step/command code execution (GO, JUMP, STEP);
- breakpoints, when and watch expressions (BREAK, BREAK WHEN, WATCH);
- examination and update of variable values (EXAMINE, SET).

SAS implementation

To start the debugger in interactive SAS the DEBUG parameter has to be added to the DATA command:

```
DATA class / DEBUG;
    SET sashelp.class;
    DO i=1 TO 2;
        PUT i= name=;
    END;
RUN;
```

When executing this code in interactive SAS the Data Step debugger opens as a separate window (Fig. 5). When open, the DATA Step debugger suspends the DATA Step and waits for debug commands. If Batch SAS is used to execute this code an error message results (see below).

```
ERROR: Unable to initialize the DATA
STEP Debugger environment.
```

The SAS implementation is strictly command based. Commands are for example STEP, GO, JUMP, EXAMINE, BREAK …. These commands have to be typed each time a command is to be executed. Abbreviations are available, such as E can be used instead of the EXAMINE command, but the code 'E <VARIABLE>' has to be typed into the debugger for each variable that should be examined.

MACUMBA implementation

MACUMBA provides a custom implementation of the DATA Step debugger. Here, the debugger is fully integrated into the program editor, not in a separate window (Fig. 6), and is completely controlled by mouse actions and keyboard shortcuts.

Some benefits are provided by the MACUMBA integration:

- code does not have to be changed for debugging;
- breakpoints can be kept over multiple debug sessions (line numbers do not change);
- all features from the editor window (e.g. syntax highlight) are available;
- variable values are displayed as a tool tip text, when mouse rests over a variable name;
- jumping in the code can be done with the mouse through Drag'n'Drop;
- editing of long values is easier (edit instead of retype).

The current implementation does not include all of the features that the SAS DATA Step debugger provides. For example 'BREAK WHEN <*condition*>' is currently not implemented but this could be added in a future release.

The DEBUG parameter does not work, because MACUMBA only 'emulates' an interactive session by using a batch session in fact. For SAS batch, the LDEBUG parameter can be used instead:

```
DATA class / LDEBUG;
    SET sashelp.class;
```

```
    DO i=1 TO 2;
        PUT i= name=;
    END;
RUN;
```

All the rest of the implementation is the invocation of the correct debugger command at the right time and parsing of the SAS log for debugger output.

For example when defining a breakpoint at line 5, the command 'BREAK 5' is submitted and the SAS log is parsed for the line 'Breakpoint <*n*> set at line 5'.

Another example is to process the next step the command 'STEP 1' is submitted and the SAS log is parsed for the line 'Stepped to line <*n*> column <*m*>'.

MACUMBA implementation issues

One of the main issues in implementing the SAS DATA Step debugger in the MACUMBA application was the determination of the start and the end of a DATA Step.

An easy solution could be to define a rule like 'It starts at the line that is begins with DATA and goes up to the line that contains RUN<SEMICOLON>.' This is true for code like the following:

```
DATA tmp;
    a = 1;
    PUT a=;
RUN;
```

For a SAS developer this rule is very weak and does not hold true as the code becomes more complex. The following section will demonstrate how difficult it can be to determine the end of a DATA Step, but the rules can generally be applied to the determination of the start of a program as well.

The initial rule implies that a new command has to start on a new line or vice versa that a new line starts a new command. This might be true for some other programming languages, but it's not true for SAS and so the rule does not hold true for code like:

```
DATA tmp;
    a = step +
        run;
    PUT a=;
RUN;

DATA tmp;
    a = step + run;
PUT a=;RUN;
```

A simple solution could be to ignore whitespaces and use the semicolon as the command delimiter. The rule to find the end could be adapted as <SEMICOLON>RUN<SEMICOLON> with whitespaces ignored. This will work correctly for both of the above DATA Steps, but still does not hold true in all cases. The following two DATA Steps show valid SAS code, but the previously defined rule will not find the correct end of the program.

```
DATA tmp;
    SET sashelp.class;
    /* simple copy of sashelp.class*/
RUN;

DATA tmp;
    SET sashelp.class;
    %calc_macro(p1=abc, p2=def)
RUN;
```

The next step would be to ignore not only whitespaces, but also comments and macro calls. This way all previously mentioned DATA Steps will correctly be determined, but it still does not hold true in all cases. The first point to be missed is that a DATA Step does not have to be closed by a RUN.

```
DATA class;
    SET sashelp.class;
    DO i=1 to 2;
        PUT i= name=;
    END;
PROC SORT DATA=class;
    BY sex age name;
RUN;
```

The rule does need to contain also PROC and QUIT as end commands. And even this is still not enough. As soon as macro quoting comes into account, a simple text search can't do the job anymore as in the next code:

```
DATA class;
    SET sashelp.class;
    DO i=1 to 2;
        PUT i= name=;
    END;
    %PUT This is an example
    %STR(%);RUN;);
RUN;
```

This is still valid SAS code and does what it should do. It outputs a strange message into the SAS log. For a text parser, this implies that it will not be possible to only search for a command and check some code before and after.

A complete SAS source code parser was implemented in MACUMBA.

First the parser is used to parse the SAS code and break it up into commands. Afterwards these commands are grouped into command groups. Such a group is for example a macro definition or an RSUBMIT block. DATA and PROC Steps are also groups and these groups can simply be used to determine the beginning and the end of a DATA Step.

Graphical Macro Debugger

A single step debugger for SAS macros is something really missing in a standard SAS development

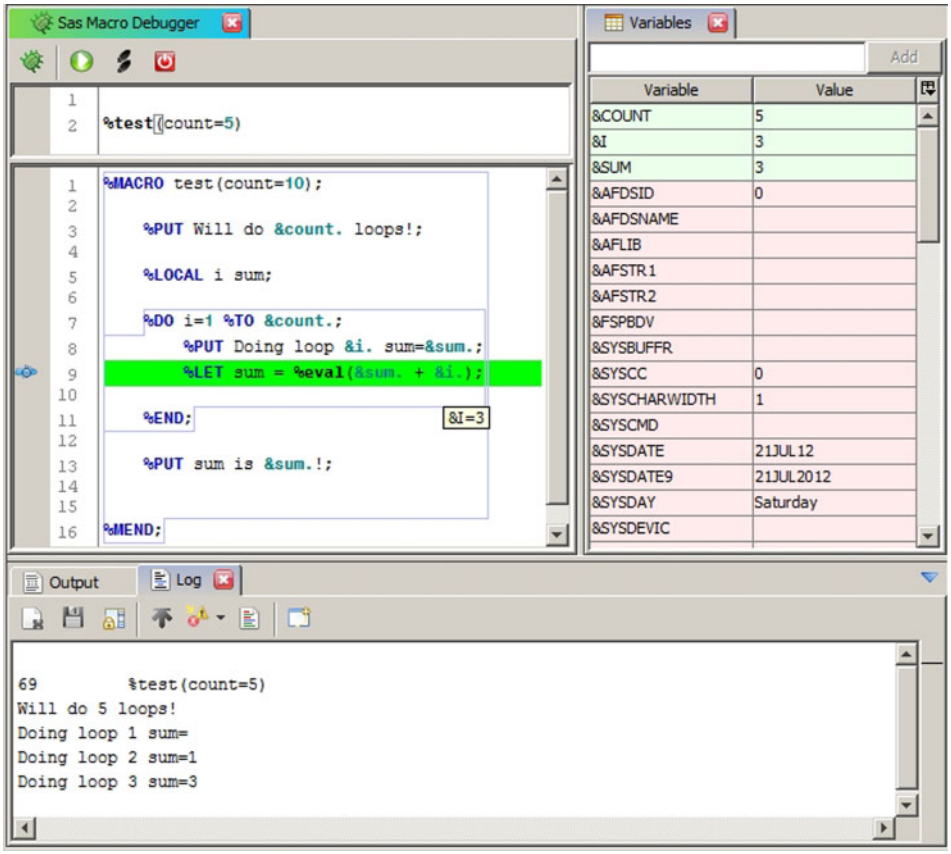

Figure 7 MACUMBA SAS macro debugger

environment. Tracking down a bug in a long and complex macro can become a real challenge.

There are two possible solutions in interactive SAS:

One is adding extra code (e.g. %PUT statements) to strategic positions like:

```
%MACRO test(value=6);
    %LOCAL out;
    %PUT DEBUG - SYSFUNC(mod(&value.,
    2))=%SYSFUNC(mod(&value., 2));
    %IF %SYSFUNC(mod(&value., 2)) EQ 0
    %THEN %DO;
        %PUT DEBUG - true;
        %LET out = %eval(&value. / 2);
    %END;
    %ELSE %DO;
        %PUT DEBUG - false;
        %LET out = %eval(&value. * 2);
    %END;
    %PUT out=&out.;
%MEND;
```

And the other one is using SAS system options like SYMBOLGEN and MLOGIC and check the SAS log:

```
MLOGIC(TEST): Beginning execution.
MLOGIC(TEST): Parameter VALUE has
value 6
MLOGIC(TEST): %LOCAL OUT
SYMBOLGEN: Macro variable VALUE
resolves to 6
MLOGIC(TEST): %IF condition
%SYSFUNC(mod(&value., 2)) EQ 0 is TRUE
MLOGIC(TEST): %LET (variable name is
OUT)
SYMBOLGEN: Macro variable VALUE
resolves to 6
MLOGIC(TEST): %PUT out=&out.
SYMBOLGEN: Macro variable OUT resolves
to 3
out=3
MLOGIC(TEST): Ending execution.
```

Both of these provide a way to get the job done, but neither is ideal.

MACUMBA contains a special view that allows interactive single step debugging in SAS macros (Fig. 7).

SAS macro debugger usage

Currently the SAS macro debugger is implemented as a special view. This is done, because the invocation of a macro differs from the definition of a macro. In most cases a macro is defined in one program and used in many other programs. The debugger window provides two text fields. One field will contain the macro call and another contains the macro definition.

For macros that are accessible through the 'autocall' macro facility, only the macro call has to be given and the macro code will be fetched automatically. For macros that are not accessible though the 'autocall macro facility the macro code has to be copied into the code field. For compiled macros, the source code can be retrieved from the compiled macro (i.e., decompiled), but this would not include

macro or block comments, as these are removed on compilation, and is therefore not recommended.

Using the SAS macro debugger is very similar to the DATA Step debugger, once the macro code has been fetched using one of the methods described above. Debugging can be started through a toolbar button or a keyboard shortcut. The same is true for the GO and the STEP actions. Breakpoints can be added and removed by mouse clicks and the instruction pointer can be moved with the mouse as well. Values of macro variables can be seen in the 'Variables' window or as a mouse over 'tool tip text'.

In case the macro can't be invoked on a 'standalone' basis, the code in the macro call text field can be adapted. This allows invoking a macro that depends on a DATA Step or invoking an in-line macro within a %PUT statement (see examples below).

```
%MACRO triangle(n);
    %LOCAL res i;
    %LET res = 0;
    %DO i = 1 %TO &n.;
        %LET res = %eval(&res. + &i);
    %END;
    &res.
%MEND;
%MACRO initVars(vars);
    %DO i = 1 %TO
    %SYSFUNC(countw(&vars.));
        ATTRIB %SCAN(&vars, &i, %STR( ))
        LENGTH=$200
        FORMAT=CHAR200.;
    %END;
%MEND;
```

These macros could not be called independently. The first one does return a value that is not correct SAS code and the second requires an active DATA Step. Instead, they would be used in code as given in the example below.

```
data abc;
    %initVars(var1 var2 var3)
    var1 = "This is a test";
    var2 = "Test for 100:";
    var3 = "Result=%triangle(100)";
run;
```

For debugging of those macros the complete code would go into the macro call field.

One other example where this separation between call and definition is very helpful is the possibility to debug one macro while this is invoked from another macro. For example to debug the macro triangle from above, while it is invoked by another macro as in the example below.

```
%MACRO sumover(n);
    %LOCAL res i;
    %LET res = 0;
    %DO i = 1 %TO &n.;
        %LET res = %eval(&res. +
        %triangle(&i));
    %END;
    &res.
%MEND;
```

In this example, it is enough to put the code from the triangle macro in the code field and to put the call to the sumover macro in the call field.

Additionally, it is possible to debug multiple macros at the same time. Therefore, the macro code of all macros is to be copied into the macro code field. While macro debugging breakpoints can be set to any of the macros and in single step mode a macro call will enter debugging of the called macro.

SAS macro debugger features

Currently the following features are implemented:

- interrupt code execution (e.g. on breakpoints) (BREAK);
- perform single step code execution (STEP);
- examine values of local macro variables (EXAMINE);
- move execution pointer (JUMP);
- execute external code inside of a macro while execution is interrupted;
- debug of multiple macros (e.g. one macro is invoking another macro).

FUTURE outlook

Features currently in development are:

- support for macro code modification while debugging;
- automatically add sub macros to be debugged on invocation if requested;
- special Code Coverage mode for validation purposes (record how often a command is reached).

Conclusion

The SAS IOM technology is a very powerful feature of SAS. Through SAS IOM[1] it was possible to create a SAS development environment that perfectly fits our needs and provides state of the art features for SAS program development and debugging.

SAS Development is easier when using MACUMBA than when using interactive SAS. Apart from the debugging functionalities that are discussed in this paper, many other features are available to provide a better overview regarding the current state of the SAS session, the available data and the way SAS programs are linked together.

The implementation of the single step macro debugger was a special challenge that would not have been possible without the well documented SAS API.[2,3]

References

1 Available from: http://support.sas.com [Access 9 October 2012].
2 S. D. Riba: 'How to use the data step debugger', JADE Tech, Inc., Clearwater, FL, USA. Available from: http://www2.sas.com/proceedings/sugi25/25/btu/25p052.pdf (SUGI paper 52-25) [Access 9 October 2012].
3 SAS 'SAS® 9.2 Integration Technologies: Java Client Developer's Guide', SAS Institute Inc., Cary, NC, USA, Available from: http://support.sas.com/documentation/cdl/en/itechjcdg/61499/PDF/default/itechjcdg.pdf [Access 9 October 2012].