Paper SM12

Using the full power of SAS to resolve Algorithmic MedDRA SMQs

Steve Prust, elderbrook solutions GmbH,Biberach an der Riß, Germany

ABSTRACT

This paper presents a method to resolve MedDRA algorithmic SMQs . The method involves using a user-written SAS function that allows data operations that might normally involve several DATA steps or PROCs to be performed in a single macro call (and within a DATA step) as well as hash tables and a little-used SAS MACRO interface. The method has also been extended to include weighted SMQs and flexible contemporaneous requirements and is a comprehensive solution to current algorithmic SMQ definitions. The resultant code demonstrates how diverse elements of the SAS language can be combined to produce powerful solutions.

INTRODUCTION

This paper is about a generalised SAS ® Macro for solving algorithmic Standardised MedDRA Queries (SMQs) for adverse events (AEs). As part of this there is:

- A consideration how the SAS architecture restricts data access in programming and what techniques can be used to improve this.
- An examination of the requirements of SMQ handling (in particular a method of resolving Boolean expressions together with handling of a contemporaneous requirement when more than one AE is to be considered.
- A description of some advanced techniques to access data flexibly
- A SAS macro that uses these advanced techniques to solve an algorithmic SMQ

SAS ARCHITECTURE AND THE PHARMAVERSE

SAS goes back to 1970s and aspects of its architecture reflects computer considerations of that time, specifically relative lack of available memory and prevalence of sequential access methods to data. The background programming language for SAS was originally PL/1 (Programming Language / 1 developed for mainframe computers by IBM). Artefacts of the IBM architecture can be seen in various aspects of the current SAS environment (such as some formats and functions). SAS language constructs like the DATA step read data sequentially into a program data vector (PDV) and applied programming instructions in a sequential manner. Complexity of programming requirements could be handled by creating new datasets with sorting and merging information. This was a fairly common type of programming at the time.

SAS was very efficient for its time and became popular for many reasons, not least the quality of its statistics, its ability to process diverse data, the flexibility of the language, and the speed of processing.

One of the industries that was an early adopter of SAS was pharmaceuticals and much of the current Pharmaverse still resembles the environment of the 70's, 80's and 90's. For example, SAS has made database access much easier but we are dominated by sequential data access from SAS datasets with a high degree of data redundancy.
The consequence is that we constantly have to gather data together before processing. That gathering processing is tedious, and can require a high degree of manipulation

Since the earliest days of SAS PROC SQL has been introduced as well as hash tables and the SAS functions that allow direct data access (perhaps intended for use with the Screen Control Language [SCL] but usable in the datastep also. These all give better tools to flexibly access data. Although better they are not ideal:

- PROC SQL can be difficult to debug especially when data is unclean
- Hash tables reside in memory and can only be used for relatively small amounts of data
- the SAS functions that allow direct data access are cumbersome,

When constructing efficient solutions - efficient both in terms of resources and in terms of code maintenance - it is worthwhile to consider these flexible approaches.

ALGORITHMIC SMQs

Algorithmic SMQs are a means by which the identification of AEs of interest can be refined (especially when compared to a "broad" search).

All algorithmic SMQs have an algorithm in either the form of a boolean expression where letters A, B, C and so on represent different AE categories or for the SMQ "Systemic lupus erythematosus (SMQ)" it is "A or Sum(Category Term Weight)>6".

Here is a full list at the time of writing of all algorithms:

Anaphylactic reaction (SMQ)	A or (B and C) or (D and (B or C))
Acute pancreatitis (SMQ)	A or (B and C)
Neuroleptic malignant syndrome (SMQ)	A or (B and C and D)
Systemic lupus erythematosus (SMQ)	A or Sum(Category Term Weight)>6
Anticholinergic syndrome (SMQ)	A or (B and C and D)
Eosinophilic pneumonia (SMQ)	A or (B and C)
Hypotonic-hyporesponsive episode (SMQ)	A or (B and C and D)
Generalised convulsive seizures following immunisation (SMQ)	A or (B and C)
Tumour lysis syndrome (SMQ)	A or (B and C)
Drug reaction with eosinophilia and systemic symptoms syndrome (SMQ)	A or (B and C and D) or (B and C and E) or (B and D and E)

Note that in the "Term Categories" (TERMCAT) used in the MedDRA definitions:

- term category "A" is always a narrow search (TERMSCOP=2)
- all other term categories are always a broad search (TERMSCOP=1)

We therefore have definitions that can be either satisfied by a narrow search being satisfied (that is "A" is true) or some combination of the broad search terms.The resolution of the narrow search will be easy, the resolution of a combination of terms is more complex because of the possibility of qualifying AEs being non-contemporaneous. Clearly this could be resolved by a manual method but this paper deprecates such an approach and instead uses a fully-programmed method of ensuring contemporaneity.

(RE)SOLVING THE ALGORITHM

Resolving a boolean expression that is stored in a SAS variable is possible. However, it is not as simple as might be hoped. Code such as this

```
data _null_;
  length condition $14;
  input condition 1-14;
  if condition then put 'Condition << ' condition '>> is TRUE'; else put 'Condition
<< ' condition '>> is FALSE';
  cards;
0 or (0 and 1)
0 or (1 and 1)
1 or (0 and 1)
;;;;
run;
```

results in error messages and incorrect resolution of the conditions (log below is edited for readability):

```
NOTE: Invalid numeric data, CONDITION='0 or (0 and 1)' , at line 14337 column 6.
Condition << 0 or (0 and 1) >> is FALSE
CONDITION=0 or (0 and 1) _ERROR_=1 _N_=1
NOTE: Invalid numeric data, CONDITION='0 or (1 and 1)' , at line 14337 column 6.
Condition << 0 or (1 and 1) >> is FALSE
CONDITION=0 or (1 and 1) _ERROR_=1 _N_=2
NOTE: Invalid numeric data, CONDITION='1 or (0 and 1)' , at line 14337 column 6.
Condition << 1 or (0 and 1) >> is FALSE
CONDITION=1 or (0 and 1) _ERROR_=1 _N_=3
```

Potentially the %eval function can be used to resolve expression such as "0 or (1 and 1)" but within the desired macro it may be difficult or impossible to use this directly.

A solution is the RESOLVE function (a SAS Data step function for macros - so similar in classification to SYMGET or SYMEXIST) together with %eval. For example:

```
data _null_;
  length condition $14 condition_prep $25;
  input condition 1-14;
  condition_prep = '%eval(' || trim(condition) || ')';
  if input(resolve(condition_prep),best.) then put 'Condition << ' condition '>> is
TRUE'; else put 'Condition << ' condition '>> is FALSE';
  cards;
0 or (0 and 1)
0 or (1 and 1)
1 or (0 and 1)
;;;;
run;
```

and the result is now error free and correctly calculated:

```
Condition << 0 or (0 and 1) >> is FALSE
Condition << 0 or (1 and 1) >> is TRUE
Condition << 1 or (0 and 1) >> is TRUE
```

With this technique it is possible to solve all the current MedDRA SMQ algorithms.

From the previous list of algorithms there are five distinct specifications:

- A or (B and C)
- A or (B and C and D)
- A or (B and C) or (D and (B or C))
- A or (B and C and D) or (B and C and E) or (B and D and E)
- A or Sum(Category Term Weight)>6

These are all directly solvable when the letters are replaced with "0"s and "1"s except for the item with "Sum(Category Term Weight)". The "Sum.." expression needs to be replaced with the actual category term letters. This code does that:

```
%* list of algorithmic smqs - full definitions;
proc sort data=dict (where=(smqalgo ne 'N' and termstat='A' and smqstat='A')) out=dict_smq;
  by mudaeccd mudaec smqalgo;
run;

%* unique list of algorithmic smqs - algorithms and codes only;
proc sort data=dict (where=(smqalgo ne 'N' and termstat='A' and smqstat='A'))
  out=smqalgo_u (keep=mudaeccd mudaec smqalgo) nodupkey;
  by mudaeccd mudaec smqalgo;
run;

%* replace Sum(Category Term Weight) with actual term letters ;
proc sql;
  create table smqalgo_u_termwght_cat as
   select * from
     (select distinct mudaeccd, termcat
       from dict_smq (where=(termwght>0 and index(smqalgo,'Sum(Category Term Weight)'))))
    order by mudaeccd, termcat;
quit;

proc transpose data=smqalgo_u_termwght_cat out=smqalgo_u_termwght_cat2
prefix=termwght_cat_;
  by mudaeccd;
  id termcat; var termcat;
run;

data smqalgo_u2;
  merge smqalgo_u smqalgo_u_termwght_cat2 (in=b keep=mudaeccd termwght_cat:);
  by mudaeccd;
  if b then do;
    array _cat[*] termwght_cat_:;
```

```
    length sumexp $50;
    do loopvar = 1 to dim(_cat);
      if not missing(_cat[loopvar]) then do;
        if missing(sumexp) then sumexp = '( '  || _cat[loopvar];
                           else sumexp = trim(sumexp) || ' + ' || _cat[loopvar];
      end;
    end;
    sumexp = '( ' || trim(sumexp) || ' )';
    put 'Old algorithm : ' smqalgo;
    smqalgo = trim( tranwrd(smqalgo,'Sum(Category Term Weight)',trim(sumexp)) ) || ' )';
    put 'New algorithm : ' smqalgo;
  end;
  else put 'Algorithm :  smqalgo;'
  drop loopvar termwght_cat_: sumexp;
run;
```

Here is the SAS log from the final step:

```
Algorithm : A or (B and C) or (D and (B or C))
Algorithm : A or (B and C)
Algorithm : A or (B and C and D)
Old algorithm : A or Sum(Category Term Weight)>6
New algorithm : A or ( ( B + C + D + E + F + G + H + I )>6 )
Algorithm : A or (B and C and D)
Algorithm : A or (B and C)
Algorithm : A or (B and C and D)
Algorithm : A or (B and C)
Algorithm : A or (B and C)
Algorithm : A or (B and C and D) or (B and C and E) or (B and D and E)
```

Now some code to show how the five different algorithms could be solved with variable data combinations:

```
data _null_;
  set smqalgo_u2 end=last;
  %* there a five unique algorithms ... these codes represent the five cases;
  where mudaeccd in ('20000021', '20000022', '20000044', '20000045', '20000225');
  call symput ('__algo'||compress(put(_n_,best.)),trim(mudaeccd));
  call symput ('__algoname'||compress(put(_n_,best.)),trim(mudaec));
  call symput ('__algodef'||compress(put(_n_,best.)),trim(smqalgo));
  if last then call symput ('__nalgo',compress(put(_n_,best.)));
run;

%* test data of algorithms and all potential AE profiles - except option where A is true;
data smq_testdata;
  length mudaeccd $10 a b c d e f g h i $1 ;
  input mudaeccd $10. a $1. b $1. c $1. d $1. e $1. f $1. g $1. h $1. i $1.;
cards;
20000021  0000
20000021  0001
20000021  0010
20000021  0011
20000021  0100
20000021  0101
20000021  0110
20000021  0111
20000022  000
20000022  001
20000022  010
20000022  011
20000044  0000
20000044  0001
20000044  0010
20000044  0011
20000044  0100
20000044  0101
20000044  0110
20000044  0111
20000045  010000000
20000045  010300000
20000045  002002021
20000045  011120300
20000225  00000
```

```
20000225  00001
20000225  00010
20000225  00011
20000225  00100
20000225  00101
20000225  00110
20000225  00111
20000225  01000
20000225  01001
20000225  01010
20000225  01011
20000225  01100
20000225  01101
20000225  01110
20000225  01111
;;;;
run;

%macro smqalgo_test1 ;
  %do i = 1 %to &__nalgo;
    %put Processing SMQ code: &&__algo&i, Algorithm: &&__algodef&i, SMQ name:
&&__algoname&i   ;
    data smq_testresult&i;
      set smq_testdata;
      where mudaeccd = "&&__algo&i";
      array termcat_res [9] a b c d e f g h i;
      length smqalgor smqalgo_eval $100 termcat $1;
      smqalgor = "&&__algodef&i";
      smqalgo_eval = '%eval( ' || trim(smqalgor) || ' )';
      do loopvar = 1 to 9;
        termcat = substr("ABCDEFGHI",loopvar,1);
        if index (smqalgo_eval,termcat) then do;
          smqalgo_eval = tranwrd(smqalgo_eval,termcat,termcat_res[loopvar]);
        end;
      end;
      drop termcat loopvar;
      %* resolving the Boolean expression using SAS macro function;
      smqalgo_result = input(resolve(smqalgo_eval),best.);
    run;
  %end;
%mend;
%smqalgo_test1;
```

Here is the SAS log and screenshots of some of the resultant datasets showing the code working correctly.

```
Processing SMQ code: 20000021, Algorithm: A or (B and C) or (D and (B or C)), SMQ name:
Anaphylactic reaction (SMQ)

NOTE: There were 8 observations read from the data set WORK.SMQ_TESTDATA.
      WHERE mudaeccd='20000021';
NOTE: The data set WORK.SMQ_TESTRESULT1 has 8 observations and 13 variables.

Processing SMQ code: 20000022, Algorithm: A or (B and C), SMQ name: Acute pancreatitis
(SMQ)

NOTE: There were 4 observations read from the data set WORK.SMQ_TESTDATA.
      WHERE mudaeccd='20000022';
NOTE: The data set WORK.SMQ_TESTRESULT2 has 4 observations and 13 variables.

Processing SMQ code: 20000044, Algorithm: A or (B and C and D), SMQ name: Neuroleptic
malignant syndrome (SMQ)

NOTE: There were 8 observations read from the data set WORK.SMQ_TESTDATA.
      WHERE mudaeccd='20000044';
NOTE: The data set WORK.SMQ_TESTRESULT3 has 8 observations and 13 variables.

Processing SMQ code: 20000045, Algorithm: A or ( ( B + C + D + E + F + G + H + I )>6 ), SMQ
name: Systemic lupus erythematosus (SMQ)

NOTE: There were 4 observations read from the data set WORK.SMQ_TESTDATA.
      WHERE mudaeccd='20000045';
NOTE: The data set WORK.SMQ_TESTRESULT4 has 4 observations and 13 variables.
```

```
Processing SMQ code: 20000225, Algorithm: A or (B and C and D) or (B and C and E) or (B and
D and E), SMQ name: Drug reaction with
eosinophilia and systemic symptoms syndrome (SMQ)

NOTE: There were 16 observations read from the data set WORK.SMQ_TESTDATA.
      WHERE mudaeccd='20000225';
NOTE: The data set WORK.SMQ_TESTRESULT5 has 16 observations and 13 variables.
```

VIEWTABLE: Work.Smq_testresult1

	MUDAECCD	A	B	C	D	E	F	G	H	I	SMQALGOR	SMQALGO_EVAL	SMQALGO_RESULT
1	20000021	0	0	0	0						A or (B and C) or (D and (B or C))	%eval(0 or (0 and 0) or (0 and (0 or 0)))	0
2	20000021	0	0	0	1						A or (B and C) or (D and (B or C))	%eval(0 or (0 and 0) or (1 and (0 or 0)))	0
3	20000021	0	0	1	0						A or (B and C) or (D and (B or C))	%eval(0 or (0 and 1) or (0 and (0 or 1)))	0
4	20000021	0	0	1	1						A or (B and C) or (D and (B or C))	%eval(0 or (0 and 1) or (1 and (0 or 1)))	1
5	20000021	0	1	0	0						A or (B and C) or (D and (B or C))	%eval(0 or (1 and 0) or (0 and (1 or 0)))	0
6	20000021	0	1	0	1						A or (B and C) or (D and (B or C))	%eval(0 or (1 and 0) or (1 and (1 or 0)))	1
7	20000021	0	1	1	0						A or (B and C) or (D and (B or C))	%eval(0 or (1 and 1) or (0 and (1 or 1)))	1
8	20000021	0	1	1	1						A or (B and C) or (D and (B or C))	%eval(0 or (1 and 1) or (1 and (1 or 1)))	1

VIEWTABLE: Work.Smq_testresult4

	MUDAECCD	A	B	C	D	E	F	G	H	I	SMQALGOR	SMQALGO_EVAL	SMQALGO_RESULT
1	20000045	0	1	0	0	0	0	0	0	0	A or ((B + C + D + E + F + G + H + I)>6)	%eval(0 or ((1 + 0 + 0 + 0 + 0 + 0 + 0 + 0)>6))	0
2	20000045	0	1	0	3	0	0	0	0	0	A or ((B + C + D + E + F + G + H + I)>6)	%eval(0 or ((1 + 0 + 3 + 0 + 0 + 0 + 0 + 0)>6))	0
3	20000045	0	0	2	0	0	2	0	2	1	A or ((B + C + D + E + F + G + H + I)>6)	%eval(0 or ((0 + 2 + 0 + 0 + 2 + 0 + 2 + 1)>6))	1
4	20000045	0	1	1	1	2	0	3	0	0	A or ((B + C + D + E + F + G + H + I)>6)	%eval(0 or ((1 + 1 + 1 + 2 + 0 + 3 + 0 + 0)>6))	1

At this point we have a method of solving any SMQ algorithm (stored in a SAS variable) when we know the values of A, B, C and so on.

The next stage is to process the data.

PROCESSING THE ADVERSE EVENTS

By definition there are always at least two AEs that need to exist for a patient in order for an algorithmic SMQ to be satisfied. An obvious approach is having found an AE that satisfies part of the SMQ then the other AEs for the patient need to be checked to see if they satisfy the remaining element(s) of the SMQ

This pseudocode for a macro shows how this might be accomplished.

BEGIN
- Identify the category letters for this SMQ (for example A, B, and C)
- Populate an array for each category letter with the AE preferred term codes ("category letter code array")
- **FOR** each subject
  - populate a variable with the algorithm ("SMQ algorithm considered")
  - initialise a result array, one item per category letter, to 0 ("category result array")
  - initialise a category term weights array, one item per category letter, to 0 ("category weights array")
  - **FOR** each AE for this subject
    - IF the AE code corresponds to any item in the "category letter code array"
    - THEN set corresponding category result array to 1
  - **ENDFOR** each AE for this subject
  - **IF** the algorithm uses a sum-of-term-weights calculation
  - **THEN** calculate the sum of weights and substitute into "SMQ algorithm considered" variable
  - **ELSE** substitute the category result array values into "SMQ algorithm considered" variable
  - resolve the "SMQ algorithm considered" variable
  - output an observation containing relevant variables and the result
- **ENDFOR** each subject

END

Some notes about the processing:

1. The population of the "category letter code array" could be done either by getting a list of codes and transposing them or by using hash tables. Hash tables were preferred for two reasons:

    a. The transposing method would need several pre-steps several differently sized arrays per category letter - this was considered awkward to code.
    b. The hash table method does not need much storage (a prime requirement for using hash tables effectively) and produces concise, relatively-readable code. Also, the array size issue mentioned in a) above becomes irrelevant if a matrix is used.
2. The macro can easily to adapted to cover all SMQ algorithms by enclosing the above in a extra macro loop (note: the code for the macro is given in Appendix 1 and does include this extra loop)
3. There is no provision to check if events are close to each other in time.

DEVELOPING THE MACRO FURTHER - CONTEMPORANEOUS EVENT REQUIREMENT

The initial example assumed that in the case of a term such as "B or C" that it did not matter when B and C might occur in relation to each other. Clearly, if there were years between an event of category B and another of category C then there could be a scientific decision to make as to whether the algorithm applies. For example, a 30 day maximum gap between the applicable events might need to be applied.

The consequence for the initial macro is that now at each event that qualifies for anything less than a simple term we need different processing. In the case of "A and (B or C)" then "A" is "simple" because satisfying events do not need to be contemporaneous with any other event; whereas "(B and C)" is non-simple because they need to be contemporaneous.

In developing a general solution a different approach is taken: for any event that satisfies the first part of a non-simple term (for example we will use "B or C" to illustrate below) then are there any other events within that term for this patient and within a time period defined by the original event start date less 30 days to end date plus 30 days?

This sounds difficult but a technique involving user written functions and the macro facility can make this happen. It relies on a solution given in a paper to the SAS Global Forum in 2012 by the author Mike Rhoads:

> "*what do you do if you need a macro that can be embedded within a SAS statement, but the underlying task requires the execution of one or more DATA or PROC steps?"*

USER WRITTEN FUNCTION TO GET AN ANSWER FROM AN SQL QUERY

Here is a simple example of a user written function to get an "answer" from an SQL query. It consists of

1. A user written macro that will firstly call an SQL query via a macro (see 2.) and then return a numeric value using the macro variable count.
2. A macro that calls an SQL and places the count of a named variable into a macro variable called "count"

```
proc fcmp outlib=sasuser.mysubs.utility;
 function GetSQLans(query $, var $) ;
 length query $ 32000  var $ 32 count $ 32;
 query_arg = dequote(query);
 var_arg   = dequote(var);
 rc = run_macro('GetSQLans_Inner', query_arg, var_arg, count);
 if rc = 0 then return(input(count,best.));
 else return(-1);
 endsub;
run;

%MACRO GetSQLans_Inner;
 %local query ;
 %let query = %superq(query_arg);
 %let query = %sysfunc(dequote(&query));

 %local var ;
 %let var = %superq(var_arg);
 %let var = %sysfunc(dequote(&var));

 %let count = 0;
  proc sql noprint;
    select count(&var) into : count from &query;
  quit;

%MEND GetSQLans_Inner;
```

The next step demonstrates how the function can be used. On the line below "count = GetSQLans (query,"name");" will execute a query that has been defined in the line above - the code is able to say for each observation "how many other observations are like (in terms of sex and age) this observation". We are now reading from the whole dataset at the same time as processing individual observations.

```
options cmplib=sasuser.mysubs;
data example;
 set sashelp.class;
 length query $32000;
 query = "sashelp.class (where=(sex = '" || sex || "' and age = " || put(age,2.) || "))";
 count = GetSQLans (query, "name");
 drop query;
run;
proc print data=example;
run;
```

In the results the COUNT column thus contains how many other members of this dataset are there with this age and sex.

Obs	NAME	SEX	AGE	HEIGHT	WEIGHT	COUNT
1	Alfred	M	14	69.0	112.5	2
2	Alice	F	13	56.5	84.0	2
3	Barbara	F	13	65.3	98.0	2
4	Carol	F	14	62.8	102.5	2
5	Henry	M	14	63.5	102.5	2
6	James	M	12	57.3	83.0	3
7	Jane	F	12	59.8	84.5	2
8	Janet	F	15	62.5	112.5	2
9	Jeffrey	M	13	62.5	84.0	1
10	John	M	12	59.0	99.5	3
11	Joyce	F	11	51.3	50.5	1
12	Judy	F	14	64.3	90.0	2
13	Louise	F	12	56.3	77.0	2
14	Mary	F	15	66.5	112.0	2
15	Philip	M	16	72.0	150.0	1
16	Robert	M	12	64.8	128.0	3
17	Ronald	M	15	67.0	133.0	2
18	Thomas	M	11	57.5	85.0	1
19	William	M	15	66.5	112.0	2

This is a simple example of what is a powerful technique. Also, it is what we need to solve the contemporaneous event requirement:

APPLYING THE USER WRITTEN FUNCTION TO THE CONTEMPORANEOUS EVENT REQUIREMENT

The first stage identify simple and complex elements of the algorithms and separate these into "clauses". For example this macro

```
%macro clause;

%do i = 1 %to &__nalgo;
  %put Processing SMQ code: &&__algo&i, Algorithm: &&__algodef&i, SMQ name: &&__algoname&i   ;
  %put ================================================================================;
  options nonotes;
  data _null_;
    length clause clausetype $50 smqalgo $200;
    smqalgo = tranwrd("&&__algodef&i", '(' , ' ( ' );
    smqalgo = tranwrd(smqalgo, ')' , ' ) ' );

    clausecount = 0;
    i = 1 ;
    code = 0;
    level = 0;
    length canditem $20;
    canditem = scan(smqalgo,i," ");
    clause = ' ';
    do while (canditem ne ' ');
      if canditem = '(' then level = level + 1;
      if canditem = ')' then level = level - 1;
      if level = 0 and canditem = 'or' then do;
        link addlevel;
      end;
      else clause = trim(left(clause)) || ' ' || canditem;

      i=i+1;
      canditem = scan(smqalgo,i," ");
    end;
```

```
    link addlevel;

    call symput ("__nclause",compress(put(clausecount,best.)));
  return;

  addlevel:
        clausecount = clausecount + 1;
        clause = left(clause);
        if substr(clause,1,1) = '(' and substr(clause,length(clause),1) = ')'
          then clause = left(substr(clause,2,length(clause)-2));
        call symput ("__clause" || compress(put(clausecount,best.)) , trim(clause) );
        clause = ' ';
  return;
  run;
  options notes;

  %put Number of distinct clauses: &__nclause;
  %do j = 1 %to &__nclause;
    %put clause &j: &&__clause&j ;
  %end;
%end;
%mend;
%clause;
```

produces this analysis (in the SAS log):

```
Processing SMQ code: 20000021, Algorithm: A or (B and C) or (D and (B or C)), SMQ name:
Anaphylactic reaction (SMQ)
================================================================================
Number of distinct clauses: 3
clause 1: A
clause 2: B and C
clause 3: D and ( B or C )

Processing SMQ code: 20000022, Algorithm: A or (B and C), SMQ name: Acute pancreatitis (SMQ)
================================================================================
Number of distinct clauses: 2
clause 1: A
clause 2: B and C

Processing SMQ code: 20000044, Algorithm: A or (B and C and D), SMQ name: Neuroleptic malignant
syndrome (SMQ)
================================================================================
Number of distinct clauses: 2
clause 1: A
clause 2: B and C and D

Processing SMQ code: 20000045, Algorithm: A or Sum(Category Term Weight)>6, SMQ name: Systemic
lupus erythematosus (SMQ)
================================================================================
Number of distinct clauses: 2
clause 1: A
clause 2: Sum ( Category Term Weight ) >6

Processing SMQ code: 20000048, Algorithm: A or (B and C and D), SMQ name: Anticholinergic
syndrome (SMQ)
================================================================================
Number of distinct clauses: 2
clause 1: A
clause 2: B and C and D

Processing SMQ code: 20000157, Algorithm: A or (B and C), SMQ name: Eosinophilic pneumonia (SMQ)
================================================================================
Number of distinct clauses: 2
clause 1: A
clause 2: B and C

Processing SMQ code: 20000211, Algorithm: A or (B and C and D), SMQ name:
Hypotonic-hyporesponsive episode (SMQ)
================================================================================
Number of distinct clauses: 2
clause 1: A
clause 2: B and C and D

Processing SMQ code: 20000212, Algorithm: A or (B and C), SMQ name: Generalised convulsive
seizures following immunisation (SMQ)
================================================================================
```

```
Number of distinct clauses: 2
clause 1: A
clause 2: B and C

Processing SMQ code: 20000219, Algorithm: A or (B and C), SMQ name: Tumour lysis syndrome (SMQ)
================================================================================
Number of distinct clauses: 2
clause 1: A
clause 2: B and C

Processing SMQ code: 20000225, Algorithm: A or (B and C and D) or (B and C and E) or (B and D
and E), SMQ name: Drug reaction with eosinophilia and
systemic symptoms syndrome (SMQ)
================================================================================
Number of distinct clauses: 4
clause 1: A
clause 2: B and C and D
clause 3: B and C and E
clause 4: B and D and E
```

The initial macro smqalgo_loop now needs adapting to use this. The macro then needs the clauses to be analysed separately and any complex clauses to have the contemporaneous requirement applied.

It is noted that the algorithm complex clauses are always in the form "<category-letter> AND ….". This means that if the first category letter is not satisfied we know that this clause result is false (or zero). Conversely, if the first category letter is satisfied then it is necessary to check the other categories using the data range requirement. This will be a call using the function defined above in the form:

```
count = GetSQLans (query, "usubjid");
```

where the variable "query" is constructed as

```
"<<AE dataset>> (where=(usubjid='" || trim(usubjid) || "' and "
|| "(('"
|| put(<<start date>>,date9.) ||"'d<=astdt<='" || put(<<end date>>,date9.)||"'d)"
|| " or ( '"
|| put(<<start date>>,date9.) ||"'d<=aendt<='" || put(<<end date>>,date9.)||"'d)"
|| " or (astdt<'" || put(<<start date>>,date9.) || "'d and aendt>'" || put(<<end
date>>,date9.) || "'d ) )"
|| " and aeptcd in ( " <<list of terms for the relevant category>> || ) ))"
```

Adjusting the macro to include the clauses and GetSQLAns function means a redesign of the macro (see Appendix 2). It is longer and less immediately readable but it fulfills its design requirement.

A REVIEW OF TECHNIQUES USED TO SOLVE AN ALGORITHM

There are a few points to highlight from the above generalised code:

- The RESOLVE function is an effective way in SAS to take an algorithm defined in a text field and find the boolean result in such a way that program execution can be controlled on an observation level.
- Hash table functionality and response times can be problematic when loading large amounts of data into memory; it would have been a solution to the contemporaneous requirement to use a hash table to read all the AEs for a patient but this was rejected for those potential problems..
- The use of hash tables is thus restricted to reading the MedDRA information, this is considered an appropriate use of hash tables and unlikely to cause memory issues
- The user-written function presented here and in other papers is a powerful method of accessing data. Anything that can be put in an SQL query can be used on an observation-by-observation basis in the SAS data step (see paper AD11 from PHUSE 2023 "*Creating applications with mash-ups: user written functions, stored sql queries, and interfaces*" for some more examples of this).
- The current attributes of the algorithmic SMQ definitions have been exploited when programming these generalised solutions.

Perhaps it would be possible to reproduce the same analysis by other methods but it is considered to be less efficient, less maintainable and harder to code.

CONCLUSION

By synthesising our programming techniques we can produce solutions to complex problems. While this paper includes only SAS techniques there are many other platforms that can be used in such ways (for example R and Python). The conclusion of this paper is that synthesis of tools and ideas is one of the best ways in which to improve.

.

REFERENCES

Use the Full Power of SAS in Your Function-Style Macros, Mike Rhoads, Paper 004-2012, SAS Global Forum 2012
Creating applications with mash-ups: user written functions, stored sql queries, and interfaces, Steve Prust, Paper AD11, PHUSE EU Connect 2023
https://admin.meddra.org/sites/default/files/guidance/file/SMQ_intguide_27_0_English.pdf

ACKNOWLEDGEMENT

The author would like to acknowledge Corentin Le Scanff for his initial idea, his tenacity, innovation and collaboration.

CONTACT INFORMATION

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

Stephen Prust
elderbrook solutions GmbH
Prinz-Eugen-Weg 24
D-88400 Biberach an der Riß
Work Phone:+49(0) 7351 52 96 49 2
Email:stephen.prust@ext.elderbrooksolutions.com

APPENDIX 1

Code for macro to resolve Algorithmic SMQ (no contemporaneous requirement)

```
%* list of algorithmic smqs - note that dataset <<dict>> is derived from the MEDDRA SMQ dataset;
proc sort data=dict (where=(smqalgo ne 'N' and termstat='A' and smqstat='A')) out=smqalgo_u (keep=mudaeccd mudaec
smqalgo) nodupkey;
  by mudaeccd mudaec smqalgo;
run;

%* load SMQ information into macr variables;
data _null_;
  set smqalgo_u end=last;
  call symput ('__algo'||compress(put(_n_,best.)),trim(mudaeccd));
  call symput ('__algoname'||compress(put(_n_,best.)),trim(mudaec));
  call symput ('__algodef'||compress(put(_n_,best.)),trim(smqalgo));
  if last then call symput ('__nalgo',compress(put(_n_,best.)));
run;

%* a loop to check macro variables are correctly loaded;
%macro test_loop;
  %do i = 1 %to &__nalgo;
    %put Processing SMQ code: &&__algo&i, Algorithm: &&__algodef&i, SMQ name: &&__algoname&i   ;
  %end;
%mend;
%test_loop;

%* macro to resolve SMQ algorithms for an input dataset <<AE_RAW>>;
%* output datasets will be <<AE_SMQ_i>> where i is the relative SMQ number;
%* output dataset contains subject id and result of algorithm - if more details are needed
  (such as qualifying AE details) then adapt the macro. HINT see where verbose switch is used below;
%macro smqalgo_loop (
   runcode=0 /* expected values: 0/1 - set to 1 to run check                  */
 , verbose=0 /* expected values: 0/1 - set to 1 to get informational messages  */
);

  %* outer loop through all algorithmic SMQs;
  %do i = 1 %to &__nalgo;
    %put Processing SMQ code: &&__algo&i, Algorithm: &&__algodef&i, SMQ name: &&__algoname&i   ;
    proc sort data=dict out=smqalgo&i (keep=termscop termcat termwght smqlevel mudaeccd mudaec mlevel mtm mtmcd);
      by mudaeccd mudaec smqalgo;
      where smqalgo ne 'N' and termstat='A' and smqstat='A' and mudaeccd="&&__algo&i";
    run;

    %local __cats&i;
    proc sql noprint;
      %* macro variable __cats<ampersand>i contains the category letters used in an SMQ;
      select distinct termcat into :__cats&i separated by "" from smqalgo&i order by termcat;
      %* macro variable __LIM contains the maximum number of terms in any of the category letters;
      select countcd into :__lim from
          (select termcat, count(mtmcd) as countcd from smqalgo&i group termcat)
        having countcd=max(countcd) order by termcat ;
    quit;

    %let ncats = %length(&&__cats&i);
    %put &=ncats __cats&i=&&__cats&i &=__lim;
    %* macro variable for the maximum size of the matrix to hold the codes for each category;
    %let ncatslim = %eval(&ncats*&__lim);

    %if &runcode %then %do;

      data ae_smq_&i;
        attrib
          usubjid         length=$11         label="Unique Subject Identifier"
          smqalgor        length=$60         label="SMQ algorithm considered"
          cats            length=$20         label="SMQ category letters"
          smqalgo_eval    length=$100        label="SMQ algorithm evaluation"
          sumwgt          length=8           label="SMQ algorithm weight sum"
          smqresult       length=8           label="SMQ algorithm result"
          termcat         length=$1          label="Term category"
          termwght        length=8           label="Term weight"
          smqalgo         length=$60         label="SMQ algorithm"
          mudaeccd        length=$10         label="MedDRA user defined AE category code"
          mudaec          length=$200        label="MedDRA user defined AE category"
          mlevel          length=$10         label="MedDRA level"
          mtmcd           length=$10         label="MedDRA term code"
          mtm             length=$200        label="MedDRA term"
        ;
        set ae_raw ;
        by usubjid;
        /* codes that belong to each category - "category code letter array" */
        length term1-term&ncatslim $10;
        retain term1-term&ncatslim ;
        array __term [&ncats,&__lim] term1-term&ncatslim;

        /* number of terms in each category, 0 indicates there are no items to check */
        length termdef1-termdef&ncats 8.;
        retain termdef1-termdef&ncats;
        array __termdef [&ncats] termdef1-termdef&ncats;
```

```
        /* result of search for terms in a category */
        length termres1-termres&ncats 8.;
        retain termres1-termres&ncats;
        array __termres [&ncats] termres1-termres&ncats;

        /* term weights for each category - applies if sum of category term weights is used */
        length termwgt1-termwgt&ncats 8.;
        retain termwgt1-termwgt&ncats;
        array __termwgt [&ncats] termwgt1-termwgt&ncats;

        %* initialise the matrix of codes by category, the result for each term category, the term weights for each
category;
        if _n_ = 1 then do;
          call missing(mudaeccd, mudaec, smqalgo, termcat, termwght, mtm, mtmcd, mlevel);

          /* hash table for SMQ def */
          declare hash dictdef(dataset:"dict", multidata: 'y');
          dictdef.defineKey("mudaeccd");
          dictdef.defineData("mudaec", "smqalgo");
          dictdef.defineDone();

          mudaeccd = "&&__algo&i";

          rc = dictdef.find();
          if rc=0 then do;
            put mudaec= mudaeccd= smqalgo= ;
            smqalgor = smqalgo;
            retain smqalgor;
            /* hash table for SMQ components */
            declare hash dict(dataset:"dict", multidata: 'y');
            dict.defineKey("mudaeccd", "termcat");
            dict.defineData("mtm", "mtmcd", "mlevel", "termwght");
            dict.defineDone();

            do i = 1 to &ncats;
              termcat = substr("&&__cats&i",i,1);
              __termdef[i] = 0;
              n = 0;
              rc2 = dict.find();
              do while (rc2=0);
                n = n+1;
                __term[i,n]  = mtmcd;
                __termdef[i] = n;
                __termwgt[i] = termwght;
                rc2 = dict.find_next();
              end;
              put termcat= __termdef[i]=  __termwgt[i]=;
            end;

          end;
          else do;
            put "%str(ERR)OR: Key not found";
            goto leave;
          end;

        end;

        %* for each patient initialise the term result as 0 (false);
        if first.usubjid then do;
          do i = 1 to &ncats;      __termres[i]=0;    end;
        end;

        %* analyse each AE as whether it contributes to the SMQ being considered;
        do i = 1 to &ncats;
          do j = 1 to __termdef[i];
            if aeptcd = input(__term[i,j],best.) then do;
              __termres[i] = 1;
              %if &verbose %then %do; put 'hit ' usubjid i= aeptcd= aedecod; %end;
              /* NOTE: there is not a test here for events to be contemporaneous
*/
              /* IF this term has to be at the same time as other terms then the question is: are there other AEs in
the correct timeframe ?                */
              /* need to invoke a method of checking this: there AEs, for this patient, within this time frame, from
this list of terms ==> function call */
            end;
          end;
        end;

        %* at the last observation for a patient output the results;
        if last.usubjid then do;
          %* construct a string to perform a SAS macro evaluate via a RESOLVE function call;
          smqalgo_eval = '%eval( ' || trim(smqalgor) || ' )';
          if index(smqalgor,'Sum') then do;
            sumwgt = 0;
            do i = 1 to 9;
              if __termwgt[i] > 0 and __termres[i] > 0 then sumwgt = sumwgt + __termwgt[i];
            end;
            smqalgo_eval = tranwrd(smqalgo_eval,'Sum(Category Term Weight)',compress(put(sumwgt,best.)));
          end;
          do i = 1 to &ncats;
```

```
            termcat = substr("&&__cats&i",i,1);
            if index(smqalgo_eval,trim(termcat)) then smqalgo_eval = tranwrd(smqalgo_eval,trim(termcat),
compress(put(__termres[i],best.)) );
          end;
          smqresult = input(resolve(smqalgo_eval),best.);
          length cats $20;
          cats = "&&__cats&i";
          output;
        end;
        keep usubjid cats smqalgor smqalgo_eval termres1-termres&ncats termwgt1-termwgt&ncats sumwgt smqresult
        ;

        leave:
      run;
    %end;

  %end;

%mend;
%smqalgo_loop (runcode=1, verbose=1);
```

APPENDIX 2

Code for macro to resolve Algorithmic SMQ with contemporaneous requirement

```
proc sort data=dict (where=(smqalgo ne 'N' and termstat='A' and smqstat='A')) out=smqalgo_u2
(keep=mudaeccd mudaec smqalgo) nodupkey;
  by mudaeccd mudaec smqalgo;
run;
data _null_;
  set smqalgo_u end=last;
  where smqalgo ne 'A or Sum(Category Term Weight)>6' ;
  call symput ('__algo'||compress(put(_n_,best.)),trim(mudaeccd));
  call symput ('__algoname'||compress(put(_n_,best.)),trim(mudaec));
  call symput ('__algodef'||compress(put(_n_,best.)),trim(smqalgo));
  if last then call symput ('__nalgo',compress(put(_n_,best.)));
run;

%macro smqalgo_loopc (runcode = 1, margin=30);

%do i = 1 %to &__nalgo;
  %put Processing SMQ code: &&__algo&i, Algorithm: &&__algodef&i, SMQ name: &&__algoname&i
;
  options nonotes;
  data _null_;
    length clause clausetype $50 smqalgo $200;
    smqalgo = tranwrd("&&__algodef&i", '(' , ' ( ' );
    smqalgo = tranwrd(smqalgo, ')' , ' ) ' );

    clausecount = 0;
    i = 1 ;
    code = 0;
    level = 0;
    length canditem $20;
    canditem = scan(smqalgo,i," ");
    clause = ' ';
    do while (canditem ne ' ');
      if canditem = '(' then level = level + 1;
      if canditem = ')' then level = level - 1;
      if level = 0 and canditem = 'or' then do;
        link addlevel;
      end;
      else clause = trim(left(clause)) || ' ' || canditem;

      i=i+1;
      canditem = scan(smqalgo,i," ");
    end;
    link addlevel;

    call symput ("__nclause",compress(put(clausecount,best.)));
  return;

  addlevel:
        clausecount = clausecount + 1;
        clause = left(clause);
        if substr(clause,1,1) = '(' and substr(clause,length(clause),1) = ')' then clause =
left(substr(clause,2,length(clause)-2));
        call symput ("__clause" || compress(put(clausecount,best.)) , trim(clause) );
        clause = ' ';
  return;
  run;
  options notes;

  %put Number of distinct clauses: &__nclause;
  %do j = 1 %to &__nclause;
    %put clause &j: &&__clause&j ;
  %end;

  proc sort data=dict out=smqalgo&i (keep=termscop termcat termwght smqlevel mudaeccd mudaec
mlevel mtm mtmcd);
    by mudaeccd mudaec smqalgo;
    where smqalgo ne 'N' and termstat='A' and smqstat='A' and mudaeccd="&&__algo&i";
  run;
  %local __cats&i;
  proc sql noprint;
    select distinct termcat into :__cats&i separated by "" from smqalgo&i order by termcat;
    select countcd into :lim from
        (select termcat, count(mtmcd) as countcd from smqalgo&i group termcat)
      having countcd=max(countcd) order by termcat ;
```

```
  quit;

  %let ncats = %length(&&__cats&i);
  %put &=ncats __cats&i=&&__cats&i &=lim;
  %let ncatslim = %eval(&ncats*&lim);

  %if &runcode %then %do;

  data ae_smq_&i;
    attrib
      termcat         length=$1    format=$1.     label="Term category"
      termwght        length=8     format=8.      label="Term weight"
      smqalgo         length=$60   format=$60.    label="SMQ algorithm"
      mudaeccd        length=$10   format=$10.    label="MedDRA user defined AE category
code"
      mudaec          length=$200  format=$200.   label="MedDRA user defined AE category"
      mlevel          length=$10   format=$10.    label="MedDRA level"
      mtmcd           length=$10   format=$10.    label="MedDRA term code"
      mtm             length=$200  format=$200.   label="MedDRA term"
    ;
    set ae_raw ;
    by usubjid;
    /* clauses in algorithm */
    length clause1-clause&__nclause $50;
    retain clause1-clause&__nclause ;
    array __clause [&__nclause] clause1-clause&__nclause;

    /* location to analyses clauses in algorithm for each observation */
    length clauseanal1-clauseanal&__nclause $50;
    retain clauseanal1-clauseanal&__nclause ;
    array __clauseanal [&__nclause] clauseanal1-clauseanal&__nclause;

    /* clauses result */
    length clauseres1-clauseres&__nclause 8.;
    retain clauseres1-clauseres&__nclause ;
    array __clauseres [&__nclause] clauseres1-clauseres&__nclause;

    /* codes that belong to each category */
    length term1-term&ncatslim $10;
    retain term1-term&ncatslim ;
    array __term [&ncats,&lim] term1-term&ncatslim;

    /* number of terms in each category, 0 indicates there are no items to check */
    length termdef1-termdef&ncats 8.;
    retain termdef1-termdef&ncats;
    array __termdef [&ncats] termdef1-termdef&ncats;

    /* term weights for each category - applies if sum of category term weights is used */
    length termwgt1-termwgt&ncats 8.;
    retain termwgt1-termwgt&ncats;
    array __termwgt [&ncats] termwgt1-termwgt&ncats;

    if _n_ = 1 then do;

      %do j = 1 %to &__nclause;
        __clause[&j] = "&&__clause&j";
      %end;

      call missing(mudaeccd, mudaec, smqalgo, termcat, termwght, mtm, mtmcd, mlevel);

      /* hash table for SMQ def */
      declare hash dictdef(dataset:"dict", multidata: 'y');
      dictdef.defineKey("mudaeccd");
      dictdef.defineData("mudaec", "smqalgo");
      dictdef.defineDone();

      mudaeccd = "&&__algo&i";

      rc = dictdef.find();
      if rc=0 then do;
        put mudaec= mudaeccd= smqalgo= ;
        /* hash table for SMQ components */
        declare hash dict(dataset:"dict", multidata: 'y');
        dict.defineKey("mudaeccd", "termcat");
        dict.defineData("mtm", "mtmcd", "mlevel", "termwght");
        dict.defineDone();

        do i = 1 to &ncats;
          termcat = substr("&&__cats&i",i,1);
```

```
          __termdef[i] = 0;
          n = 0;
          rc2 = dict.find();
          do while (rc2=0);
            n = n+1;
            __term[i,n]  = mtmcd;
            __termdef[i] = n;
            __termwgt[i] = termwght;
            rc2 = dict.find_next();
          end;
          put termcat= __termdef[i]=  __termwgt[i]=;
        end;

      end;
      else do;
        put "%str(ERR)OR: Key not found";
        goto leave;
      end;

    end;

    /* at the start prepare the clause analysis area and set all clause results to 0 [false]
*/
    do i = 1 to &__nclause;
      __clauseanal[i] = __clause[i];
      __clauseres[i] = 0;
    end;

    /* for each clause assess perform the following steps
*/
    /* 1. does the obs under consideration relate to the first category letter in clause ?
*/
    /*    if not then clause result is 0 and move onto next item, if so move onto step 2.
*/
    /* 2. for all the following category letters in the clause check for each letter :
*/
    /*     - if this patient had an AE in this category within the date range (plus margin)
*/
    /*       of the AE found in step 1
*/
    /*     - substitute the result of check into the clause in place of the category letter
*/
    do i = 1 to &__nclause;

      /* loop through all the categories in a term */
      catcount = 0;
      firstcatres = 0;
      firstcatst = .;
      firstcaten = .;
      format firstcatst firstcaten date9.;
      length nextcat $1;
      nextcatpos = indexc(__clauseanal[i],'ABCDEFGH');
      do while (nextcatpos);
        catcount = catcount + 1;
        nextcat = substr(__clauseanal[i],nextcatpos,1);
        nextcatres = '0';
        nextcatidx = index("&&__cats&i",nextcat);   /* the array index with PTs for category
*/

        if catcount = 1 then do;  /* if first loop iteration check terms for this category
*/

          do j = 1 to __termdef[nextcatidx];
            if aeptcd = input(__term[nextcatidx,j],best.) then do;
              /* a term is matched - store result and calculate contemporaneous date range
*/
              firstcat    = nextcat;
              firstcatres = 1;
              firstcatst  = astdt - &margin;
              if missing(aendt) then
                firstcaten = today();
              else
                firstcaten = min(aendt + &margin, today());
            end;
          end;
          if firstcatres then nextcatres = '1';
        end;
        else do;  /* second/ subsequent iteration of loop --> check other clause categories
*/
```

```
          if firstcatres then do;
            /* construct SQL query        */
            length query $10000;
            query = "ae_raw (where=(usubjid='" || trim(usubjid) || "' and "
                 || "(   ( '" || put(firstcatst,date9.) || "'d <= astdt <= '" ||
put(firstcaten,date9.) || "'d ) "   /* date element */
                 || " or ( '" || put(firstcatst,date9.) || "'d <= aendt <= '" ||
put(firstcaten,date9.) || "'d ) "
                 || " or ( astdt < '" || put(firstcatst,date9.) || "'d and aendt > '" ||
put(firstcaten,date9.) || "'d ) )"
                 || " and aeptcd in ( ";
              do j = 1 to __termdef[nextcatidx];    /* add the applicable terms to query */
                if j > 1 then query = trim(query) || ",";
                query = trim(query) || " " || trim(__term[nextcatidx,j]) ;
              end;
            query = trim(query) || ") ))";
            count = GetSQLans (query, "usubjid"); /* run query - returns how many match */
            if count > 0 then nextcatres = '1';

          end;
        end;

        if firstcatres then put __clauseanal[i]= nextcatpos= nextcat= nextcatres=;
        __clauseanal[i] = tranwrd(__clauseanal[i],nextcat,nextcatres);
        nextcatpos = indexc(__clauseanal[i],'ABCDEFGH');
      end;
      if firstcatres then do;
        length smqalgo_eval $100;
        smqalgo_eval = '%eval( ' || trim(__clauseanal[i]) || ' )';
        __clauseres[i] = input(resolve(smqalgo_eval),best.);   /* resolve the algorithm */
        if __clauseres[i] then do;  /* if this clause is satisfied then output */
          clause_satisfied = __clause[i];
          output;
        end;

      end;
    end;  /* of loop through all clauses */
    keep usubjid aeptcd astdt aendt firstcat: clause_satisfied query smqalgo_eval;

    leave:
    return;

  run;
  %end;

%end;

%mend;

%smqalgo_loopc;
```