CS05 Re-programming a many-to-many merge with Hash Objects

David J Garbutt,
GarbuttConsult, Basel, Switzerland

The hash object has been available in SAS for some time but does not yet seem to have reached the mainstream. This talk presents a case where its use is compelling but also revealing: The case of the many-to-many merge as exemplified by program to check CDISC vocabularies. This case illustrates the strengths and weaknesses of the hash object and also provides useful example code.

1. Introduction

The aim of this paper is two-fold, first to introduce the hash object in two situations which pose familiar SAS programming problems, and second to show how it widens the scope of the data step beyond where it has been for the last 20 years.

The hash object was made a production feature in SAS 9.1 (released in 2006[1].) six years ago and yet does not seem to be in widespread use. This is despite the fact that it is with PROC FCMP the most useful addition to the SAS language since data step functions (DOPEN, DFETCH etc.) and the macro language itself. It provides a new model for storage of data in memory while a data step is running and it communicates in a simple way with the Program Data Vector (PDV).

I first used hash objects in a SAS program in 2010 [Garbutt (2011)] and was intrigued and impressed, first, because the example code I used worked almost immediately, second because apart from setting up the hash object the code was a trivial two lines, and third because, for that problem, there was not a simpler conventional solution. For these reasons I included a section on hash objects for the common merge problem explained and solved with formats in [Banga & Garbutt (2011)]. A talk at PhUSE 2011 by Guido Wendland about checking CDISC controlled vocabularies impressed me with its clarity and neat solution of using macro variables to embed the vocabularies in an IN() test, but left me thinking the problem might be more easily solved using hash arrays and the CHECK() method, and as example 2 shows it turns out to be the case.

The two examples discussed in this paper are based on previous papers discussing many-to-many merge problems [Franklin (2011), Wendland (2011)]. These problems are familiar and frequent and make a good area to illustrate the power of the hash object. In [Franklin (2011)] the problem was solved using formats as a lookup table and in [Wendland (2011)] by moving data into macro variables. Both papers are recommended reading and if many-to-many merges are not familiar ground for you I suggest reading them before the example section in this paper.

No criticism is meant of these two papers but their clarity of exposition makes it easy to build a third solution on.

The primary sources for information about the Hash object is of course the SAS documentation [SAS Hash Objects, 2011], [Ray & Secosky (2008)] is useful because

[1]The hash array was experimental in 9.0, Production in 9.1 and enhanced with the multidata option in SAS 9.2.

it explains the new features added in 9.2. There are more papers coming out all the time on the power of hash objects, [Hinson & Shi (2012), DeVenezia (2008), Secosky (2012), Dorfman & Fridman (2010), Dorfman (2002), Dorfman, 2010, Blackwell (2010), Ray & Secosky (2008), Rashul (2010)] and it is an area to watch for new work. The SAS object teams page is also an area to monitor for developments [SAS Hash Object Support area

This paper will examine hash objects, what they are, how you can use them in SAS programs, how to program data driven data steps with them, why you should start using them, and the effect they can have on SAS programming.

1.1. Other implementations of the hash array

Hash arrays (or content-addressable arrays, associative arrays, or Multimaps) were first available in SNOBOL4 and, among scripting languages, in AWK (first released in 1977)[2]. Here is an AWK program[3] to read a list of filenames , extract the filetype (as diagnosed by the file command) and count how many occurrences there are of each filetype.

```
xargs file |
awk '{
$1=NULL;
t[$0]++;
}
END {
for (i in t) printf("%d\t%s\n", t[i], i);
}' | sort -nr   # Sort the result numerically, in reverse
```

The line `t[$0]++` increments the counter for each word and the second line (executed at the end of the input file) prints out the list of indexes (i) used and their counts `t[i]`. The array, as in shown in the sparsity of the code above knows what indexes it has. This might seem a world away from the normal array of variables and their indexing by an integer. And in one sense it is, it is backwards. But in fact the normal array is just a set of buckets (for holding strings or numerics), the index variable, and with a rule to calculate which bucket to use. A hash array is the same but the index variable is the value itself, and the rule uses that value to calculate which bucket to add 1 to.

One might think that such an exotic method for storing and accessing a datastore would be inefficient and slow. But in fact the hash array proper uses a hash function to calculate a storage bucket address for the value and this is generally a very fast function somewhat like a random number generator. The result is that the storage, retrieval, and query functions take a constant time regardless of the amount of data stored in the array. This is very different from searching an array of variables for a match. It is a method that scales and does not suddenly get slower when put into production.

Hash arrays are also implemented in most modern scripting languages, including Korn Shell 93, bash and Power Shell, as well as higher level languages[4], and now, SAS

1.2. Problems with Abstraction in SAS programming

Abstraction or the principle that systems of programs should not duplicate functionality has been widely touted since the late 70's[5] and goes along with encapsula-

[2] http://en.wikipedia.org/wiki/AWK

[3] from Unix Power tools script count_types.sh available from here http://examples.oreilly.com/9780596003302/

[4] See http://en.wikipedia.org/wiki/Associative_array

[5] http://en.wikipedia.org/wiki/Abstraction_principle_(computer_programming)

tion, and automation as a fundamental tool of software engineering. Another way to look at it is that abstraction allows you to avoid hard-coded hell, and encapsulation protects you from sharp tridents in the neighbouring fire-pit.

In SAS programming this reliance on hard coding was a big problem till version 5 when the current macro language was introduced. The macro language[Philp (2008)] works by allowing text substitution to happen before SAS compiles a step and provides a way to pass information between data steps and to conditionally execute SAS code. Large complex systems can be written with SAS macro, and are. But, it is difficult to write SAS programs that are completely data dependent in the way that can be done in dynamic languages like Lisp: in summary — it lacks the EXEC() function. There is the CALL EXECUTE function which executes SAS statements assembled as character strings and is powerful and easy to debug. However CALL EXECUTE does not execute the program statements right away, it stacks them until the current data step compiles and runs and executes them afterwards. This characteristic limits their flexibility for use in systems. Nevertheless, much that is done with macros could be as easily (perhaps more easily) done with CALL EXECUTE. There are also the dataset functions (OPEN, FETCH, FETCHOBS, CALL SET, GETVARN, GETVARC, etc.)[6] that can read and write datasets and query their attributes outside the context of a data step[Galbis-Rieg (2007)]. There is a stack for macro variables that are local to a macro execution but integration often relies on global variables and this can give problems when variable names are automatically generated.

Abstraction in SAS programs is a slippery concept because in PROCS there *is* a lot of abstraction — for example, the SORT procedure does not need to know if the keys are character or numeric, and the model definition languages in GENMOD and other procs are very high-level. But if we examine the data step programming language then we can see the problems. I think it is widely accepted that it is difficult to write high level *systems* in SAS that are generic and fast. And I think the sources of the problem are, in no particular order,

1. the separation of data steps and procs from each other and
2. the way macro language is implemented purely as a program text generator,
3. macro variables are simple, individual, variables. Other structures and collections have to be emulated using resolvable variable names,
4. the lack of other data types in the data step besides character and numeric scalars (they are scalars because of the way the PDV works). At least without using IML, and extra products cannot be relied on by the programmer.
5. The lack of truly encapsulated user-written functions within the data step itself.
6. lack of an EXEC() call that can execute any SAS code and return the result immediately to the macro context

Since version 9.2 the items on this list have now all been addressed by the recent additions to the SAS language (PROC FCMP, hash objects and the DOSUB and DOSUBL macros [Secosky (2012)]).

Reading papers from PhUSE and other meetings leads me to the observation that meta-data driven reporting systems are the next wave and it is a good time now to explore other ways of programming that are purely data dependent, with programs

[6] SAS documentation for dataset functions at http://support.sas.com/documentation/cdl/en/lefunctionsref/63354/PDF/default/lefunctionsref.pdf, see the SAS file IO category.

that can, in principle, be compiled data steps. If this can be done then I believe the validation[7] and speed benefits are there.

1.3. Memory models in SAS

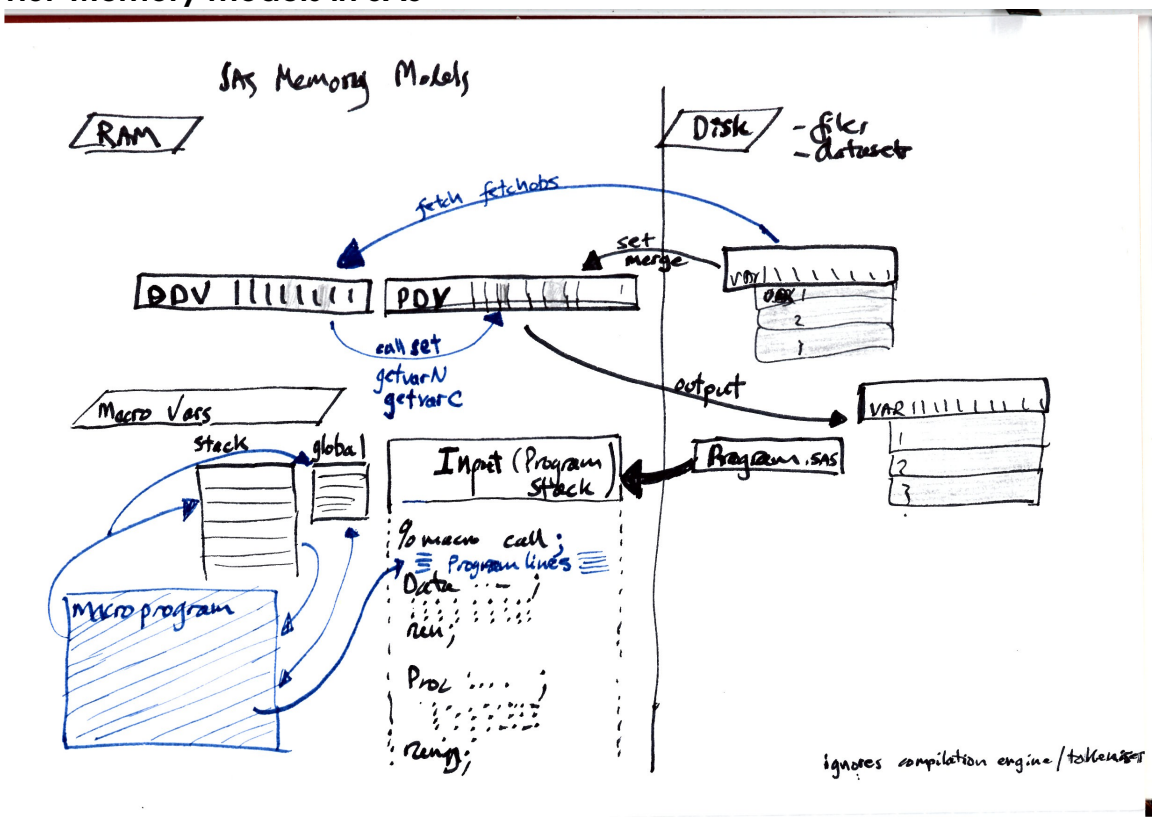

Figure 1: SAS memory model for data and macro variables

In most other programming languages the memory model is clear and simple — variables and objects the program knows about are in memory and are always available (*pace* scoping rules), External data are read from disk or perhaps from instruments and sensors. But with SAS the act of reading (and writing) datasets is integrated with programming. The classic SAS memory model is illustrated in Figure 1[8] By default the data step reads and writes to and from datasets stored on disk using the SET, MERGE and UPDATE statements. While this happens only the current observation is allocated space in memory. This area is called the program data vector (PDV) and is a contiguous area of memory holding the variables in the order they have been met in the program. A second area of memory called the dataset data vector (DDV) is used by the data set access functions[Galbis-Rieg (2007)] to get data in and out of datasets outside data steps and allow that information to be passed to macro variables. These functions are commonly used within macro language and are used to fetch metadata about datasets (to better set-up code generation by the macro) and also to fetch actual data for processing. When used for data processing their strength is also their weakness: using a separate area as a data buffer is good because it does not interfere with the PDV contents, but bad if you want to do a SET as in example 1. Previously the program needed to call FETCH or FETCHOBS and then call GETVARC or GETVARN explicitly for each variable to be copied. The CALL SET routine simplifies the copying of data from the DDV to the

[7]Validation is easier because fixed data steps can use the data step debugger and are fixed code with all the maintenance advantages that brings. In this kind of system a lot hangs on the quality of the data coming in but these are the kind of problems we have been working with for years.

[8]For those interested in learning more about this see the excellent paper [Lavery, 2007]

PDV but if a variable on the new dataset does not already exist in the PDV it is not created. This means that to do a merge type operation the dataset coming in to the PDV must be queried with data step functions first and then PDV space created for them. For problems with relatively few variables this is tedious although possible but for more general problems it makes using the data step functions painful.

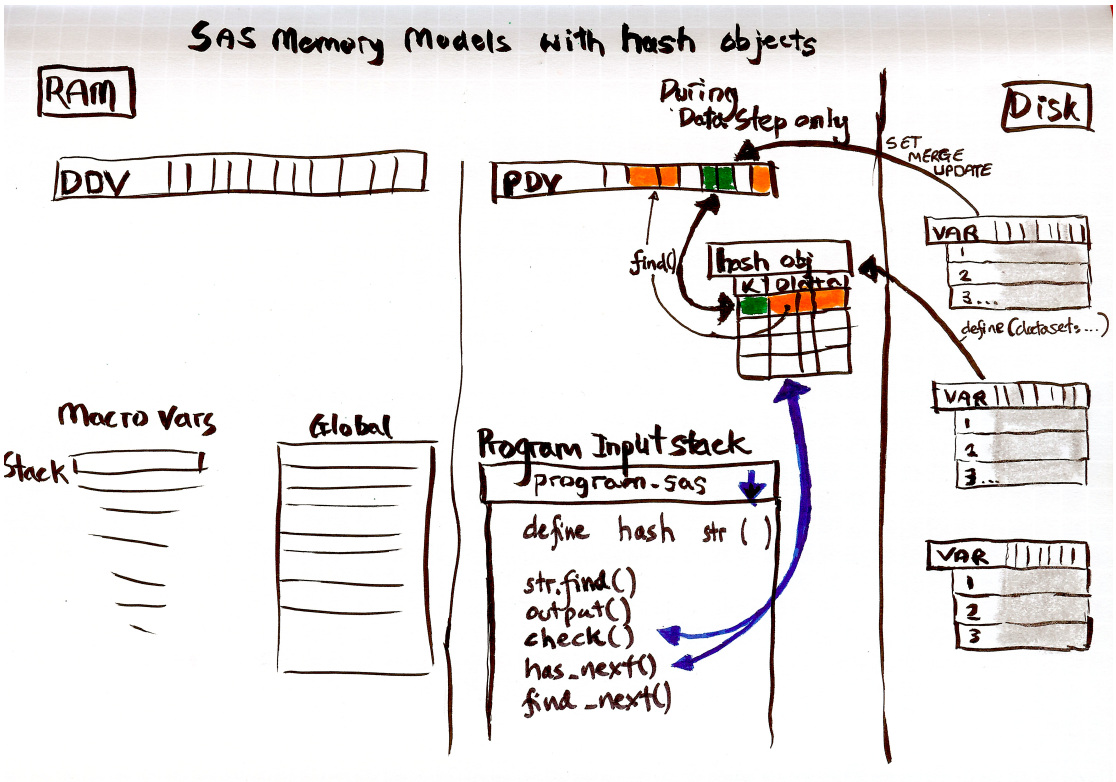

Figure 2: SAS Memory Model with added hash object and sample methods. Note The PDV and the hash object do not persist between data steps whereas macro variables and the DDV do.

There is another point: just like the PDV it is designed for holding *one* observation at a time. In other words — it works like the SET statement in Example 1, it must be in a loop to process all of the observations in a dataset. In this light we can see how the hash object differs so much — it holds a *whole* data table in memory while the data step executes.

In Figure 2 we see how the hash object changes this memory model. Now two areas have values only during their data step: the PDV and any hash objects. To query a single value in a hash object we do not search the whole table. Read operations take an execution time independent of the number of observations in the hash. Hash objects can use multiple variables as keys and can return multiple variables as data. These variables can be data or character. The hash object is also integrated with the PDV in a smart way — it uses key variables as read-only parameters and only writes to the data variables. This makes it unlike a SET statement because the SET statements just sets its variables to missing, then reads in all the values overwriting any that already have values. This behaviour of SET means variables need to be re-named as in example 1.

Hash objects can be output to datasets and built up in memory. So they are not limited to look-up but can also be used for real processing as well. In addition any data step can create new hash objects to be saved and re-used.

Hash objects only exist during a data step and this is different from macro variables and would seem to be less useful. But because a hash object can be initialised

directly from a dataset at creation this is problem is theoretical rather than practical. Every data step in a system can easily have access to a standard set of hash objects containing meta data, either by repeating code or by writing an initialising macro. The information persists as datasets between data steps and is as available as a macro variable. The consequence of this is that any programs using large numbers of macro variables to store metadata or control data can almost certainly be rewritten to use hash objects instead.

2. Example 1 - A many-to-many merge

It is time to see the hash object in action.

Proc SQL and set point=

Performing a many-to-many merge (an inner join) with a data step does not normally give the expected result and generally using PROC SQL is the best option but sometimes we need another method and there are other ways as discussed in [Franklin (2011)]. The example used in that paper is to merge adverse events data with concomitant medications and the key here is that there may be more medications than AEs or vice versa. Franklin uses the the following data to illustrate this method.

```
/* this program from
   http://www.theprogrammerscabin.com/PS11TU05.pdf */
data ae; ** Adverse Event Data;
   infile cards;
   input ptnum $ 1-3 @5 date date9. event $ 15-35;
   format date date9.;
cards;
001 16NOV2009 Nausea
002 16NOV2009 Heartburn
002 16NOV2009 Acid Indigestion
002 18NOV2009 Nausea
003 17NOV2009 Fever
003 18NOV2009 Fever
005 17NOV2009 Fever
;
run;
data cm; ** Concomitant Medication Data;
   infile cards;
   input ptnum $ 1-3 @5 date date9. medication $ 15-35;
   format date date9.;
cards;
001 16NOV2009 Dopamine
002 16NOV2009 Antacid
002 16NOV2009 Sodium bicarbonate
002 18NOV2009 Dopamine
003 18NOV2009 Asprin
004 19NOV2009 Asprin
005 17NOV2009 Asprin
;
run;
```

The PROC SQL code is on the wiki page and here is the correct output.

```
** SAS Output
Merge using SQL -- most common way this is seen done
*/
Obs ptnum date event medication
1 001 16NOV2009 Nausea Dopamine
2 002 16NOV2009 Heartburn Antacid
```

```
3 002 16NOV2009 Heartburn Sodium bicarbonate
4 002 16NOV2009 Acid Indigestion Antacid
5 002 16NOV2009 Acid Indigestion Sodium bicarbonate
6 002 18NOV2009 Nausea Dopamine
7 003 18NOV2009 Fever Asprin
8 005 17NOV2009 Fever Asprin
*/
```

The data step solution uses a second SET statement inside a loop to check through all the CM observations and execute an output if the keys match. The program is straight forward and reminds us that we can use many SET statements inside a data step.

```
** SAS Code;
data all1;
  set ae;
       ** Our loop within a loop -- output if match;
    do i=1 to xnobs;

       set cm (rename=(ptnum=_ptnum date=_date))
              nobs=xnobs
              point=i    ;
       if ptnum=_ptnum and date=_date then output;
end;
  drop _: ; ** Drop temporary variables;
run;
```

Note that in the second SET statement we must rename the PTNUM and DATE variables to avoid the values from the first set being overwritten. We can see looking at the memory model diagram in Figure 1 that have to have both values (from CM and the current value from the outer set) to see if there is a match. To make a comparison we have to copy the values into the PDV which is what the inner SET does. We could stop searching CM after a match except for the fact that we cannot assume the CM dataset is ordered and for each PTNUM and DATE pair there can be many medications. It is not possible to add a where clause to the SET statement like

```
set cm (rename=(ptnum=_ptnum date=_date)
         where=(ptnum = _ptnum and date = _date )
               nobs=xnobs
               point=i;
```

which would subset each query exactly as needed. The consequence of this is that the SET statement in inner loop reads the entire CM dataset for every row of the AE dataset. The total number of reads will be the number of obs in cm x the number of obs in AE.

Another method is the use of an indexed dataset and SET with KEY= and an excellent guide to how this works is to be found in [Lavery, 2007].

Using a hash object

It makes sense to use a hash object for this problem because we can make a simple look-up without reading all of CM and without overwriting the PDV.

To use a hash object in a data step it has first to be declared and then created. A hash can be initialised with a data set or built up an item at a time. A hash object called CM can be created and initialised in one statement like this:

```
declare hash cm(dataset: "work.cm",
                hashexp:16) ;
```

This creates the hash object and says it will be initialised with data from the data set WORK.CM. The HASHEXP: 16 parameter sets the default size to be 2^{16}, the hash will grow as needed but if you can make a good guess then do so. The name of the hash object must be unique — you cannot use a (variable) name that is also present in any datasets you will be loading. It is also worth knowing that you can specify dataset options to the initialising dataset using the obvious extension to the syntax:

```
dataset: "work.cm (where = (ptnum = 3))",
```

The hash object has next to be told which are the key variables and which are the extra 'looked up' variables that will be loaded into the PDV when a key value is found. To do this we call the DEFINEKEY method :

```
cm.defineKey('PTNUM', 'DATE');
```

The key variables should exist in the current data step (i.e. type and length are known to the SAS compiler and are present in the PDV) if they are not known then warnings will be issued because the SAS compiler cannot see anything within the parentheses () and so it cannot tell what type or length the variables to be loaded have. A simple way to do that is to add

```
if 0 then set cm (keep= ptnum,date, medication) ;
```

to the code. This does nothing but makes the SAS compiler aware of the variables needed. Next we define the data variables (and normally repeat the key variables:

```
cm.defineData('PTNUM','DATE','medication');
```

The hash is now sufficiently defined to be usable so we call the DEFINEDONE method:

```
cm.defineDone() ;
```

Now the variables are known to the data step compiler it will check to ensure they are initialised and so adding a call missing for each variable will prevent that warning message from being issued. This method works for any list of variables, and the variables can be character or numeric:

```
call missing(ptnum, date, medication) ;
```

Note that in these examples we supply the variables names to the hash methods as strings but they could be SAS character variables or expressions.

The above code only needs to be executed once so it is normally wrapped in an

```
if _n_ = 1 then do;

end;
```

block.

There are various operations we can do on a hash object and perhaps FIND(); is the most useful. In its simplest form we can write :

```
cm.find();
```

and the key values (of PTNUM and DATE in the PDV loaded by the SET statement) are searched for in the hash and if found the data values (MEDICATION) are copied out into the PDV. That is actually all it does.

All the hash object methods act like call statements and deliver a return code. A zero return code means the operation (FIND for example) was successful. A non-zero code means failure and for FIND() of course that means the key values are not contained in the hash. The code so far has been simplified by leaving the return codes unchecked, but in practice you should always check them and inform accordingly[9]. So let us add it:

```
rc = cm.find();
      /* Got CM data that matches key? */
if rc = 0 then output ;
```

If non-zero then there was no corresponding value in CM.

The full program with output is on the wiki page with the result, which is not good. Incorrect, in fact. If we look in the data more carefully we see the reason is that the key pairs are not unique in the CM dataset. By default when duplicate keys are found the first is kept and the others are ignored and no message is given in the log. Since SAS 9.2 this can be changed by adding

```
duplicate: 'error'
```

into the definition of the hash object. The program above would then have stopped when the first duplicate was found.

In this example we should keep all duplicates and since SAS 9.2 we can add

```
multidata: 'YES'
```

to the hash declaration. But how does this help with the merge? It actually allows us to use code with the same logic as used by the SET= method discussed above — if there is a match we can then loop through the duplicated keys and call OUTPUT for each occcurrence. To do this we cannot use CM.FIND() we need operators to allow us to step through all the stored values and these are HAS_NEXT() and FIND_NEXT(). HAS_NEXT() tests if there is another entry and returns zero if there is. We can then write a DO WHILE loop controlled by a variable set using the HAS_NEXT() method. The variable to be set cannot be assigned to since the method returns only the return code. Therefore we name a result variable in the method call using the RESULT: keyword. The whole is protected by an outer test so it is only tried if there is at least one value (i.e. the first CM.FIND() returns a zero).

```
rc = cm.find();
more = 0 ;
if rc = 0 then do;
   output ;
   cm.has_next(result: more) ;
    *-- test for more than one value
        count of values left goes in 'more' --;
   do while ( more ) ;
  rc = cm.find_next() ;
  if rc  = 0 then output ;
  cm.has_next(result: more) ;
   end;
end;
```

This amended version is correct and gives the same results as the other methods. The code is more compact than the set version but the initialisation of the hash object takes several lines. There is no need for renaming data step variables with the hash program. This is because the SET statement is used to read successive

[9] If you do not assign the return code to a variable and it is non-zero SAS will put an error message to the log.

observations regardless of their value and then *after* the observation is copied to the PDV the value is checked for equality and skipped if not equal. The operation of the hash is designed so that key variables are *only* used to look up and data variables are only overwritten *if* a match is found.

For datasets with small sizes there is little performance difference between these methods but with large CM data sets we could expect the set method to take longer because all values are checked for a match. With the hash method all searches take constant time no matter how many items are in the hash object. This should mean that the normal advice when merging datasets of widely different sizes (loop through the biggest data set with the main loop and put the smaller in the inner loop) should be reversed for best performance when using hashes[10].

3. Example 2 - A validation problem in CDISC Vocabularies

The problem and a way to approach it is detailed in [Wendland (2011)]. The problem can be seen as a many-to-many merge followed by a look up. In the published solution the controlled vocabularies are manipulated into a data set and then the text value substituted in to a (macro-generated) clause somewhat like this:

```
data violations (keep=variable error) ;

set ae ;

if aeacn not in
('DOSE INCREASED','DOSE NOT CHANGED','DOSE REDUCED',...)
then do;
   variable = 'AEACN' ;
   error = 'value of AEACN'||strip(aeacn)||' not found in dictionary' ;

output ;
end;
/* next test follows ...*/
run;
```

Much of the work is in constructing the datasets and is explained very clearly with animations in the presentation cited.

This solution works by making a text comparison for the variable and generates a program with all the text comparisons needed for all the variables with a controlled vocabulary.

What if we could just do a fast look up in the controlled vocabulary? That would be simpler. We already met the FIND() method working on the hash object, but this pulls in the data variables which is not necessary in this case. Fortunately there is also there is the CHECK() method. It searches for the key and returns 0 (zero) if the key was found. So we should be able to use code (assuming the hash with the code lists is called CODELST) something like:

```
notfound = codelst.check(key: CODELIST_CODE, Key: AEACN );
if notfound then do;
   variable = 'AEACN' ;
   error = 'value of AEACN'||strip(aeacn)||' not found in dictionary' ;

output ;
/* next variable... */
```

Notice that unlike most of the hash object examples in the SAS documentation we have not enclosed the keys in quotes. This is because many of the examples

[10] Of course this only applies as long as the larger dataset fits into memory

given in the SAS documentation just use text values for simplicity, but in fact the arguments can be variables or expressions. In this case we want want to check CODELIST for the content of AECN, not for the string 'AECN'. But what about the data sets, preparing them? Surely there is still a lot of work to do?

For this demonstration I have taken the data from the NCI website[SDTM Terms, NCI] where they provide an XML file. This XML file can be read using a custom XML map and code in the Appendix A. An Excel spreadsheet is also supplied which can be read with PROC IMPORT using the code given in the Appendix.

Whichever method is used there is the same problem with the data. There are two kinds of rows mixed in the table — first rows with data about the different code lists and second rows with the actual coded values. This means the contents of the CDISC_SUBMISSION_VALUE holds either the CODELIST value or the code id number of the code. In the tables of values the CODELIST is a unique id for each row and the CODELIST name is held with the code for ACN from the row for the table

Code	Code List Code	Codelist Name	CDISC Submission Value
C66767		Action Taken with Study Treatment	ACN
C49503	C66767	Action Taken with Study Treatment	DOSE INCREASED
C49504	C66767	Action Taken with Study Treatment	DOSE NOT CHANGED
C49505	C66767	Action Taken with Study Treatment	DOSE REDUCED
C49501	C66767	Action Taken with Study Treatment	DRUG INTERRUPTED
C49502	C66767	Action Taken with Study Treatment	DRUG WITHDRAWN
C48660	C66767	Action Taken with Study Treatment	NOT APPLICABLE

Because of this it makes sense to split the data into two tables, which is done by the data step that follows.

```
data
    cs05l.codelist_md (
         keep = CDISC_SUBMISSION_VALUE CODELIST_CODE
         RENAME=(CDISC_SUBMISSION_VALUE=CODELIST)
         )
    cs05l.codelist (
       keep=CDISC_SUBMISSION_VALUE CODELIST_CODE
       RENAME=(CDISC_SUBMISSION_VALUE=CODELIST_value)
         ) ;
set cs05l.sdtm_code_xl ;
if codelist_code = ' ' then do;
   codelist_code = code ;
   output cs05l.codelist_md ;
   end;
else
   output cs05l.codelist ;
run;
```

We can later use the CODELIST_CODE to look up the values.

Not all variables in the AE dataset have controlled vocabularies and on a particular run we might wish to use a subset (perhaps for testing) and so we use a control data set called MAPLIST, as used by Wendland.

```
data cs05l.maplist ;
  length ruleID $6 variable $8 Codelist $411 ;
   input ruleid @10 variable @20 codelist $10. ;
datalines ;
CT0001    AEACN      ACN
CT0002    AESEV      AESEV
CT0009    DOMAIN     DOMAIN
```

```
CT0027    AEOUT      OUT
CT0037    AEBODSYS   SOC
CT0064    AESER      NY
;
```

This dataset uses the name of the lookup table (ACN) not the code which is on each row (C66767), so therefore for each observation of AE we need to first look up for AEACN the name of the code list, and then look up the corresponding CODELIST code. Then we can look up the variable value to see if it in the hash. Note that these tests run using the complete CDISC terms with over 6000 rows.

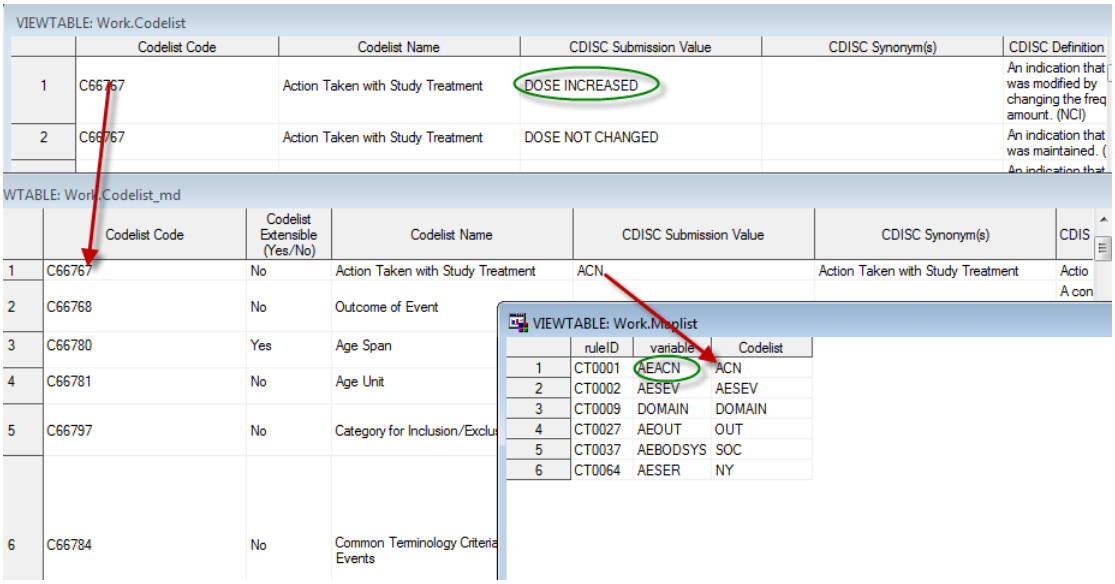

Figure 3: Contents of the three datasets to be imported as hash objects. Arrows trace the path from Vocabulary value (DOSE INCREASED), via CODELIST CODE, and CODELIST NAME to variable name for checking (AEACN in this example). Note that the variable CDISC_SUBMISSION_VALUE holds different information in the CODELIST and CODELIST_MD data sets. These variables are therefore renamed to avoid problems with overwriting.

We declare hashes for all three datasets (the mapping list, the CODELIST, and the CODELIST_MD). By performing the find methods in the right order we can use values retrieved by the previous find as keys for the next one.

```
varleft = vars_to_check.first();
    /* uses key=ACN to get codelist=ACN */

rc = codelist_md.find();
    /* uses key=ACN to get codelist_code=C66767

notfound = codelst.check(key: CODELIST_CODE, Key: AEACN );
/* searches for combination of  C66767 and  AEACN   */
```

That is the core of the solution with a set of AE and the lookup, and then for failed lookup the steps to construct the reporting variables and an output statement. Table 1 shows the sequence of changes to values in the PDV as the above sequence of finds is executed. IT shows the extended version where the variables in MAPLIST are all searched for each observation in AE. A discussion is below in the next section.

The full solution in Appendix B is just one data step with a SET statement, three hash objects and not a single macro variable in sight. A data driven program.

Source	Dataset :AE	Maplist	Maplist	Maplist	Codelst	Codelst	PDV
				Codelist_md	Codelist_md		
Variable	AEACN	RuleID	Variable	Codelist	Codelist_code	codelist_value	RC
Sample Value	Treatment	CT00001	AEACN	ACN	C66797	INCREASED	0
set ae ;	Treatment						.
variable = 'AEACN';	Treatment		AEACN				.
rc = maplist.find();	Treatment	CT00001	**AEACN**	*ACN*			0
rc = codelist_md.find();	Treatment	CT00001	AEACN	**ACN**	*C66797*		0
codelist_value= vvaluex(variable);	Treatment	CT00001	AEACN	ACN	C66797	Treatment	0
rc = codelst.check() ;	Treatment	CT00001	AEACN	ACN	**C66797**	**Treatment**	2365
rc not zero means text not found so generate error							

Table 1: Program flow for Example 2 following changes to PDV as the hash objects are queried. **Red bold** text is acting as a key for that line, *Blue Italic* text is acting as data, that is, it is returned by the FIND() method

3.1. Some more details

Some complications do arise on the way, for example, the CODELST hash object (it turns out a hash cannot be named the same as a pre-existing variable in the data step) should be keyed on the text and the CODELIST_ID (in case values repeat in other codes). But when using CHECK() to find values it is not necessary to use the FIND_NEXT() or iterators to search explicitly for duplicates.

Iterating through the dataset variables

This would be a classic case for storing the variable values from the MAPLIST dataset into indexed macro variables and then building a macro loop to duplicate code separately for each variable. But there is another way.

We can read the MAPLIST dataset into a hash object and iterate through it. To enable iteration we must declare a hash iterator after the normal declaration:

```
/* Maplist matches variable and codelist (num id) */
declare hash maplist(dataset: "cs05l.maplist"
       , hashexp:4
       , ordered: 'yes');
rc + maplist.DEFINEKEY('VARIABLE')  ;
rc + maplist.DEFINEDATA('VARIABLE','RULEID','CODELIST');
rc + maplist.DEFINEDONE();
call missing(VARIABLE, ruleid, codelist) ;
    if rc then
       putlog 'ERRO' 'R: problem creating hash obj maplist' ;

/* define iterator for maplist to check a list of variables */
    declare hiter vars_to_check('maplist') ;
```

The syntax is plain and simple and there are no complications with that. To retrieve values [11] (sorted in this case because we specified ORDERED: 'YES') we use the FIRST() and NEXT() methods. NEXT() returns a return code of zero when there are no items left. So we can construct a DO WHILE type of loop using the FIRST() method and the NEXT() methods to iterate through the values stored in the hash.

```
 varleft = vars_to_check.first();
 do while (varleft = 0 ) ;
 RC = 0;
 notfound = 1 ;

    call missing(ruleid, codelist
                , codelist_value, codelist_code) ;
    varleft = vars_to_check.next() ;
end;
```

Now that we loop through the variables we cannot write just

```
notfound = codelst.check(key: CODELIST_CODE, Key: AEACN);
```

Because the value we need to look up is different each time we go round the loop. Luckily there is a SAS function (VVALUEX) to return the value of a variable whose name is kept in a character variable (confusingly called VARIABLE in the MAPLIST dataset).

```
valtotest = vvaluex(variable) ;

notfound = codelst.check(key: CODELIST_CODE, Key: valtotest );
```

There are two complications left. The first is that we need to add an if test in case the first look up of table number does not find a vocabulary corresponding to the code typed in in maplist.

```
rc = codelist_md.find();
if rc ne 0 then do;

   source = 'CS05-testdata' ;

   value=vvaluex(variable) ;
   keyinfo = catx(' ','The codelist requested for'
               ,codelist,'('
               ,codelist_code
                   ,') was not found in the SDTM vocab list'
               );
  output;
 end;
 else do;
        /* check the value that should conform */
    valtotest = vvaluex(variable) ;
```

The second extra point is that, as is common with do loops, we should take care to reset values that would not be initialised when we start the next variable iteration and might therefore persist through to later iterations when they should not.

```
call missing( ruleid, codelist,
              codelist_value, codelist_code) ;
```

[11] SAS version 9.2 and later only

Looping across datasets

The problem here is that there seems to be no way to put a variable name in the SET statement. There are several ways to tackle this — including, of course, a macro loop. One way would be to extend the mapping list to include the domain name, or perhaps calculate it from the variable name and then to execute a conditional set inside a SELECT (DOMAIN) ; statement.

A third route would be to compile the data step we already have and then use a data step reading the directory for the datasets to create a list of datasets that we could process with a single data step that would generate statements with call execute calling our compiled data step.

```
data violations /PGM=work.violations ;
   [redacted]
run;
   *--- generate list of members in vmembers --;
data _null_ ;
   set vmembers ;
   call execute ('data PGM=work.violations ;' ) ;
   call execute('execute ;') ;
   call execute('redirect INPUT AE='||strip(memname)||';') ;
   call execute('redirect output  violations =viol_"
                ||strip(memname)
                ||'; run;');
run;
```

A fourth route (and the oldest) would be to generate the code and outputting it to a temporary file (the filename TEMP type does this nicely) and then %INCLUDE the file. This technique is clearly explained by [Fraeman, (2010)]. Another and somewhat new route is described below.

4. Discussion

4.1. How hash objects and PROC FCMP change the data step

Away from hardcode hell and towards a single data step

Hash objects in contrast do not persist at the data step boundary after they were created they vanish. This would seem to limit their usefulness and in some ways it does, however hash objects can be loaded from datasets and also written out to datasets and this means they can persist between data steps. In the work with hash objects I have done loading them seems almost free. If this is a general rule then hash objects can be an effective way to communicate between data steps and to control the processing they do. This opens a new route away from hard coded hell.

There are several major advantage of using hash objects to pass (control, or metadata) between data steps over using macros variables:

1. access to the values does not depend on the size of the hash
2. a hash object can have many key variables and they can be numeric or character
3. any number of variables can be returned by the hash key,
4. a hash object is a set of structured records with a single name, and methods to extract, query, and iterate through it.

These kind of structures can only be simulated with macro variables by using macro variables in names, but, how can I put it? they do not lead to programs that are readable or easy to change.

In summary the new hash object addresses issue 1–4 mention in section 1.2. both the storage limitations of macro variables and the lack of any table like structures outside the PDV. What of the other issues of real functions and subroutines?

The introduction of PROC FCMP is SAS Version 9.2 makes true functions and call routines available, and is the second part of the programming revolution in SAS 9. Functions written with this tool are full citizens in the SAS world - they can be called anywhere a standard function can be called, in data steps, PROC SQL, compute blocks in PROC REPORT. They can be passed arrays, and the call routines can be passed values by reference, presumably including hashes.

A recent paper by Secosky[Secosky (2012)] has announced the DOSUB and DOSUBL functions. These function execute SAS code in memory and the data step waits until they finish.

```
data _null_ ;
  rc = dosubl('proc sql ;
       create table work.regions as
       select distinct region from sashelp.shoes; ');
run;
```

The RUN_MACRO function run a SAS macro and also waits until it returns and can access the macro variables created and pass their values back to the originating data step. It must be called from a PROC FCMP function. This supplies the functionality of EXEC() as mentioned in section 1.1. The example function in the above paper that opens datasets and uses data access functions is also very instructive. This allows a proc step to be called from within a data step and allow it to pass values (via a macro) back to the original data step. It appears that this mechanism must provide a nested environment within the one executing the DOSUBL code.

This work changes the landscape of SAS programming completely and is another step towards making the data step and not the macro the centre of SAS system programming.

Hash objects as drivers of processing

We have seen an example of how the new possibilities of hash objects allow us to replace a longish macro (generating hundreds of lines of code) with a single data step to create a truly data driven program.

The ability to pull any table into a hash within a data step means that control data and looping can be explicitly done within any, or every data step. If all needed commands can be given using text operations and passed to call execute (or another way to include code) after the data step has finished generating code then I believe this way of programming systems can be made to work.

Some suggested next steps

1. Extend the method to access data in the SAS Dictionary tables or views which currently must be stored in an unstructured way in macro variable names (with the names holding the keys, or indices).

2. Explore these techniques in meta data driven systems and find what benefits there might be.

3. Benchmark this approach vs others with more realistic sizes of data

4. Benchmark vs a large macro driven system, paying attention to disk use and time needed for program compilation.

5. Use PROC FCMP to encapsulate some of the program, such as the declaration of a hash object.

6. Use a hash object with PROC FCMP? The documentation mentions that hash objects are compatible with PROC FCMP but does not explain in what way. [12] For example can they passed as a parameter? Are they treated as global so they can be regarded as global? Some experimentation is called for here.

7. Using Hash arrays should work well for the derivation order problem in big systems used for calculating derived data such as ADaM datasets. It should be possible to implement Tarjan's algorithm and calculate the order calculation directly in one data step. This method would also be useful for calculating running order in systems running SAS report jobs either on one server or if parallelising to run many SAS jobs simultaneously on a cluster of servers.

8. Free the hash object! Ask for hash objects to be allowed outside data steps in the next SAS Ballot!

This paper is available at the PhUSE Wiki at http://www.phusewiki.org/wiki/index.php?title=Reprogramming_a_many-to-many_merge_with_Hash_Objects The full code is also there for viewing and will also be made available bitbucket or Git for co-operative editing and download. Join in the discussion on the Wiki page.

5. Conclusions

There is only so much time and effort we can spend on our work and when we have a new tool or work at a higher level of abstraction then we save time, but more importantly we also increase the range of problems that can be tackled successfully.

When we add the capabilities of the hash object as a bearer of metadata to the new possibilities in PROC FCMP to call arbitrary programs from a data step it is clear a new age is dawning for SAS programming.

It's the 21st century and SAS programming is fun again!

6. Acknowledgements

I would like to thank Guido Wendland and David Franklin for their gracious permission to build on their previous work. Tanks are also due to Rohit Banga for his comments on the manuscript and for drawing my attention to the paper on the SET= method by Russ Lavery. I would also like to thank my family for putting up with my absences from their company while writing this paper, but also for letting me know I was being missed.

Contact Information

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

[12] http://listserv.uga.edu/cgi-bin/wa?A2=ind1204e&L=sas-l&P=281

Author Name	David J Garbutt
Company	GarbuttConsult
Web	Http://www.garbuttconsult.ch/
Address	Ingelsteinweg 4d
PLZ/City	CH - 4341, Dornach, Switzerland
Phone:	+41 78 734 437 6
Email:	mailto:david.j.garbutt@gmail.com

References

[CDISC/NCI, 2012] CDISC/NCI controlled vocabulary published here http://evs.nci.nih.gov/ftp1/CDISC/SDTM/

[Banga & Garbutt (2011)] Banga,R & Garbutt D, Seven Sharp Surgery tips for Clinical Programmers, 2011, PhUSE, http://www.lexjansen.com/phuse/2011/cc/CC07.pdf 1

[Blackwell (2010)] Blackwell, J, Find() the power of Hash - How, Why and When to use the SAS® Hash Object, NESUG 2010, http://www.nesug.org/Proceedings/nesug10/bb/bb01.pdf 2

[DeVenezia (2008)] DeVenezia, R, 2008, Using HASH to find a sum over a transactional path, NESUG 2008, http://www.nesug.org/Proceedings/nesug09/cc/cc22.pdf 2

[Dorfman (2002)] Dorfman, P, The Magnificent DO. SESUG, 2002. http://www.devenezia.com/papers/other-authors/sesug-2002/TheMagnificentDO.pdf. 2

[Dorfman, 2010] Dorfman, P, HoW to DOW, NESUG 2010, http://www.nesug.org/Proceedings/nesug10/hw/hw03.pdf 2

[Dorfman & Fridman (2010)] Dorfman, P & Fridman, M, Black Belt Hashigana, NESUG 2010 http://www.nesug.org/Proceedings/nesug10/bb/bb03.pdfcode: 2

[Fraeman, (2010)] Fraeman, K H, Let SAS® Write and Execute Your Data-Driven SAS Code, NESUG 2010, http://www.nesug.org/Proceedings/nesug10/cc/cc12.pdf 15

[Franklin (2011)] Franklin, D, A many-to-many merge, without SQL?, PharmaSUG 2011, available from http://www.theprogrammerscabin.com/PS11TU05.pdf 1, 6

[Garbutt (2011)] Garbutt, D, Rebuilding Define.pdf by reading Word and XFDF files , PhUSE 2011, http://www.lexjansen.com/phuse/2011/pp/PP13.pdf 1

[Galbis-Rieg (2007)] Galbis-Reig, Felix, Data Without (Step) Boundaries: Using Data Access Functions, NESUG 2007. Available at http://www.nesug.org/proceedings/nesug07/cc/cc15.pdf 3, 4

[Hinson & Shi (2012)] Hinson J , Changhong Shi, Transposing Tables from Long to Wide: A Novel Approach Using Hash Objects, PharmaSUG 2012, http://www.pharmasug.org/proceedings/2012/PO/PharmaSUG-2012-PO14.pdf 2

[Lavery, 2007] Lavery, R, 2007, An Animated Guide: The Map of the SAS Macro Facility, PhUSE 2007, http://www.lexjansen.com/phuse/2007/is/is01.pdf 4

[Lavery, 2007] Lavery, R, 2008, An Animated Guide: Speed Merges with Key merging and the _IORC_ Variable, PharmaSUG 2008, http://www.lexjansen.com/pharmasug/2008/tt/tt17.pdf 7

[Philp (2008)] Philp, S, SAS® MACRO: Beyond the Basics, SAS Global Forum 2008, http://www2.sas.com/proceedings/forum2008/045-2008.pdf 3

[Rashul (2010)] Rashul, A Side of Hash for You To Dig Into, NESUG 2010, 2

[Ray & Secosky (2008)] Ray, R & Secosky, J, Better Hashing in SAS® 9.2 , SAS Global Forum 2008, available from SAS Support site http://support.sas.com/rnd/base/datastep/dot/better-hashing-sas92.pdf 1, 2

[SDTM Terms, NCI] The XML file used in these programs is here: http://evs.nci.nih.gov/ftp1/CDISC/SDTM/SDTM/Terminology.odm.xml 11

[Secosky (2012)] Secosky, J, 2012, Executing a PROC from a DATA step, SAS Global Forum 2012, http://support.sas.com/resources/papers/proceedings12/227-2012.pdf 2, 3, 16

[SAS Hash Objects, 2011] SAS Institute, SAS Component Object online Manual, http://support.sas.com/documentation/cdl/en/lecompobjref/63327/PDF/default/lecompobjref.pdf 1

[SAS Hash Object Support area] SAS Institute, Component Object Support area http://support.sas.com/rnd/base/datastep/dot/ 2

[Wendland (2011)] Wendland, G, A SAS macro to check SDTM domains against controlled terminology, PhUSE 2011, http://www.lexjansen.com/phuse/2011/cs/CS02_ppt.ppt 1, 10

Appendix - Programs and Data

The code and example code is too long to include all in this paper so it has been added to the paper's wiki page at http://www.phusewiki.org/wiki/index.php?title=Reprogramming_a_many-to-many_merge_with_Hash_Objects. The most essential pieces are included here for ease of reading.

A. Example Data

A.1. Example II

The SDTM files ware freely available from NCI website download the XML or XLS files from

http://evs.nci.nih.gov/ftp1/CDISC/SDTM/

It can be read with the following programs.

Read Excel file

```
libname cs05l "C:\Users\garbuda4\projects\CS05-djg" ;
PROC IMPORT OUT= cs05l.sdtm_code_xl
DATAFILE= "H:\DATA\Projects\CS05-djg\SDTM Terminology.xls"
DBMS=EXCEL REPLACE;
     RANGE="'SDTM Terminology 2012-08-03$'";
     GETNAMES=YES;
     MIXED=NO;
     SCANTEXT=YES;
     USEDATE=YES;
     SCANTIME=YES;
     TEXTSIZE=32000;
RUN;
```

Read ODM XML file

```
/* Or use this program using an XML map for the CDISC SDTM controlled terminology list
The XMLMap is in the the next section */
data cs05l.codelist_md (
          keep =  CDISC_SUBMISSION_VALUE CODELIST_CODE
          RENAME=(CDISC_SUBMISSION_VALUE=CODELIST))
     cs05l.codelist (
          keep=CDISC_SUBMISSION_VALUE CODELIST_CODE
          RENAME=(CDISC_SUBMISSION_VALUE=CODELIST_value)) ;
   set cs05l.sdtm_code_xl ;
   if codelist_code = ' ' then do;
      codelist_code = code ;
      output cs05l.codelist_md ;
   end;
   else
      output cs05l.codelist ;
run;
```

XML Map to CDISC terms

```
filename SDTMCT 'H:\garbutt\DATA\Projects\CS05-djg\SDTM_CT.xml.xml';
filename SXLEMAP 'H:\garbutt\DATA\Projects\CS05-djg\CDISC_terms.map';
```

```
libname SDTMCT xml xmlmap=SXLEMAP access=READONLY;
data codeList_M;
   SET SDTMCT.CodeList_M;
run;
data codelist;
   set sdtmct.codelist;
   codelist_name= scan(oid,3,'.') ;
run;
```

Read maplist dataset

```
data cs05l.maplist ;
  length ruleID $6 variable $8 Codelist $411 ;
input ruleid @10 variable @20 codelist $10. ;
datalines ;
CT0001    AEACN      ACN
CT0002    AESEV      AESEV
CT0009    DOMAIN     DOMAIN
CT0027    AEOUT      OUT
CT0037    AEBODSYS   SOC
CT0064    AESER      NY
;
```

XMLMap for CDISC data

```
<?xml version="1.0" encoding="windows-1252"?>
<SXLEMAP
     description="XML Map to extract vocabulary and table of codelist properties"
     name="CDISC_Vocab" version="1.2">
    <!-- ############################################################ -->
    <TABLE name="CodeList_M">
        <TABLE-PATH syntax="XPath">
           /ODM/Study/MetaDataVersion/CodeList
        </TABLE-PATH>
        <COLUMN name="CodelistID">
            <PATH syntax="XPath">
             /ODM/Study/MetaDataVersion/CodeList/@nciodm:ExtCodeID
             </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>7</LENGTH>
        </COLUMN>
        <COLUMN name="Name">
            <PATH syntax="XPath">
              /ODM/Study/MetaDataVersion/CodeList/@Name
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>53</LENGTH>
        </COLUMN>
        <COLUMN name="codelist_name">
            <PATH syntax="XPath">
              /ODM/Study/MetaDataVersion/CodeList/nciodm:CDISCSubmissionValue
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>8</LENGTH>
        </COLUMN>
        <COLUMN name="Synonym">
            <PATH syntax="XPath">
               /ODM/Study/MetaDataVersion/CodeList/nciodm:CDISCSynonym
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>53</LENGTH>
        </COLUMN>
        <COLUMN name="PreferredTerm">
            <PATH syntax="XPath">
            /ODM/Study/MetaDataVersion/CodeList/nciodm:PreferredTerm
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>90</LENGTH>
        </COLUMN>
```

```
    </TABLE>
    <!-- ############################################################ -->
    <TABLE name="Codelist">
        <TABLE-PATH syntax="XPath">
          /ODM/Study/MetaDataVersion/CodeList/EnumeratedItem/nciodm:CDISCDefinition
        </TABLE-PATH>
        <COLUMN name="OID" retain="YES">
            <PATH syntax="XPath">
              /ODM/Study/MetaDataVersion/CodeList/@OID
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>19</LENGTH>
        </COLUMN>
        <COLUMN name="codeListID" retain="YES">
            <PATH syntax="XPath">
               /ODM/Study/MetaDataVersion/CodeList/@nciodm:ExtCodeID
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>7</LENGTH>
        </COLUMN>
        <COLUMN name="_Codelist_ID" ordinal="YES">
            <INCREMENT-PATH beginend="BEGIN" syntax="XPath">
               /ODM/Study/MetaDataVersion/CodeList/EnumeratedItem/nciodm:CDISCDefinition
            </INCREMENT-PATH>
            <RESET-PATH beginend="BEGIN" syntax="XPath">
                /ODM/Study/MetaDataVersion/CodeList
            </RESET-PATH>
            <TYPE>numeric</TYPE>
            <DATATYPE>integer</DATATYPE>
        </COLUMN>
        <COLUMN name="codelist_name" retain="YES">
            <PATH syntax="XPath">
               /ODM/Study/MetaDataVersion/CodeList/nciodm:CDISCSubmissionValue
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>8</LENGTH>
        </COLUMN>
        <COLUMN name="CodedValue" retain="YES">
            <PATH syntax="XPath">
               /ODM/Study/MetaDataVersion/CodeList/EnumeratedItem/@CodedValue
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>137</LENGTH>
        </COLUMN>
        <COLUMN name="Synonym">
            <PATH syntax="XPath">
                /ODM/Study/MetaDataVersion/CodeList/EnumeratedItem/nciodm:CDISCSynonym
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>86</LENGTH>
        </COLUMN>
        <COLUMN name="Definition">
            <PATH syntax="XPath">
               /ODM/Study/MetaDataVersion/CodeList/EnumeratedItem/nciodm:CDISCDefinition
            </PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>1041</LENGTH>
        </COLUMN>
    </TABLE>
</SXLEMAP>
```

B. Full SAS code for Example II

Check all variables in AE for errors

```
/* check AE vars for issues using BAD data*/
*--Original program fragments at
   http://www.lexjansen.com/phuse/2011/cs/CS02_ppt.ppt ;
```

```
libname cs05 '~/cs05' ;
data cs05l.violations
    (keep=keyinfo source domain ruleid variable value) ;
length source $13 variable domain $8 ruleid $40
       value $200 keyinfo valtotest $1000 ;
call missing(source,variable, domain, ruleid, value, keyinfo );
if _n_ =1 then
 do;
    if 0 then do;
   set cs05l.maplist;
   set cs05l.codelist ;
   set cs05l.codelist_md ;
   end;
    rc = 0 ;
/* Maplist matches variable and codelist (num id) */
declare hash maplist(
            dataset: "cs05l.maplist"
          , hashexp:4
          , ordered: 'YES'
          , duplicate: 'E');
rc + maplist.DEFINEKEY('VARIABLE')  ;
rc + maplist.DEFINEDATA('VARIABLE','RULEID','CODELIST');
rc + maplist.DEFINEDONE();
call missing(VARIABLE, ruleid, codelist) ;
    if rc then
       putlog 'ERRO' 'R: problem creating hash obj maplist' ;
       /* define iterator for maplist to check a
          list of variables */
    declare hiter vars_to_check('maplist') ;

rc = 0 ;
    declare hash codelist_md(
            dataset: "cs05l.codelist_md ",
            hashexp:16,
            duplicate: 'E' );
rc + codelist_md.DEFINEKEY('CODELIST')  ;
rc + codelist_md.DEFINEDATA('CODELIST', 'CODELIST_CODE');
rc + codelist_md.DEFINEDONE();
call missing(CODELIST, CODELIST_CODE) ;
if rc then
      putlog 'ERRO' 'R: problem creating hash obj codelist_md' ;
/* codelist matches codelist num id with allowed
       values in CDISC_SUBMISION_VALUE duplicate keys
       must be retained!*/
rc = 0 ;
    declare hash codelst(dataset: 'cs05l.codelist',
                         hashexp:16,
                         duplicate: 'E' );
rc + codelst.DEFINEKEY('CODELIST_CODE', 'CODELIST_VALUE')  ;
rc + codelst.DEFINEDATA('CODELIST_CODE','CODELIST_VALUE');
rc + codelst.DEFINEDONE();
call missing(codelist, ruleid, codelist_value) ;
if rc then
       putlog 'ERRO' 'R: problem creating hash obj codelist' ;
  end;
 set cs05l.aewrong ;
      /* loop over variables in maplist to be checked */
 varleft = vars_to_check.first();
 do while (varleft = 0 ) ;

 RC = 0;
 notfound = 1 ;
      /* assemble the linked information */

 rc = codelist_md.find();
 if rc ne 0 then do;
source = 'CS05-testdata' ;
            /* copy vars to the query dataset variables */
    domain='AE';
    value=vvaluex(variable) ;
    keyinfo = catx(' '
                   ,'The codelist requested for'
                ,codelist
                   ,'('
                ,codelist_code
                   ,') was not found in the SDTM vocab list'
```

```
              );
 output;
end;
else do;
/* Copy value of current variable into valtotest
   put no quotes on values because we get values
       from those variables and not from
   not constants like most examples in SAS manual
     */

       valtotest = vvaluex(variable) ;

   notfound = codelst.check(key: CODELIST_CODE,
                                 key: valtotest );
/* use two keys because text might match a ok decode -
   but from another codelist */

if notfound then do;

   source = 'CS05-test' ;
 /* construct error variables and output */
   domain='AE';
   value=vvaluex(variable) ;
   keyinfo = catx(' ',
            'Value of',
            quote(strip(vvaluex(variable)))
            ,'is not found in the controlled vocabulary '
       ,codelist
       ,'('
           ,codelist_code
           ,') defined for'
       ,quote(strip(variable))
           );
   output;
 end;

else
   putlog 'rc missing or neg' notfound= rc=
            variable= codelist= codelist_code= ;
end;
call missing( ruleid, codelist,
                 codelist_value, codelist_code) ;
varleft = vars_to_check.next() ;
end;
     *-- of loop over variables in AE listed in maplist
         to check for conformance --;
drop rc ;
run;
```