PharmaSUG 2017 - Paper DS21

Programming Efficiency in the Creation of ADaM BDS Datasets

Ellen Lin, Amgen Inc, Thousand Oaks, CA

ABSTRACT

The ADaM Basic Data Structure (BDS) has become one of the most prominent and widely implemented dataset structures in the industry since the CDISC ADaM Implementation Guide V1.0 was published in 2009. The strictly vertical data design of the BDS brings two common challenges to statistical programming:

(1) BDS datasets often quickly grow very large, especially in larger clinical trials.

(2) Metadata for BDS datasets become more difficult to develop and understand due to the use of value-level metadata (VLM) for describing variables by PARAM/CD and the use of multiple BASETYPEs and other derived data records within the same dataset.

This paper will describe the programming challenges specific to BDS and illustrate with examples how to achieve better programming efficiency and quality. The approaches include: 1) designing streamlined programming steps to maximize data processing efficiency; 2) reading metadata (e.g., VLM and Controlled Terminology) directly into dataset creation to ensure consistency and avoid error-prone hardcoding; and 3) using modular macros to standardize common data computations and imputations.

1. INTRODUCTION

In 2009, the CDISC ADaM team published ADaM Implementation Guide V1.0, which defines the Basic Data Structure (BDS) as one of the standardized ADaM dataset structures. Since then, BDS has been widely implemented in the industry - typically for by-visit finding data and other special cases, such as time-to-event analysis and over-time summaries.

The ADaM BDS, especially for the by-visit finding data, has a strict vertical design. It requires that analysis parameters (PARAMCD), analysis visits (AVISIT) including assessment time points and summary time periods, imputations of missing assessments or average of multiple assessments (DTYPE), and baseline types (BASETYPE), when applicable, are all built into a vertical structure.

This BDS vertical structure has two inherent issues. First, the BDS datasets often quickly grow very large, especially in larger clinical trials. In addition to the sheer number of measurements collected over time, many analyses require additional derived / repeat data records to be added in the BDS for imputations, averages, different baseline types, etc.

In addition to the size, BDS datasets often are also quite complex. The addition of derived data records means that the data derivation is in two dimensions, variables and records. To further complicate matters, the current variable-based ADaM metadata are difficult to understand for complex BDS datasets because they are not effective for describing the relationship between variables and records, especially when datasets are involved with multiple BASETYPEs, summary time points (AVISIT), derived parameters (PARAMCD), and / or other various record derivations (DTYPE).

These two issues inherent to vertical structure, size and complexity, may create several issues in programming related to BDS, such as:

- Excessive program runtime on large datasets due to inefficient programming code
- Increased program development time and frequent programming errors due to the hard-to-understand specifications
- Inconsistent results across studies and low reusability of programs due to the inconsistent understanding of the specifications by different programmers

Facing these challenges in programming for BDS datasets, we identified several programming techniques to create programs with greater efficiency, maintainability, reusability, and understandability. In this paper, we discuss three specific techniques:

1) Designing streamlined program steps (discussed in Section 3)
2) Reading metadata directly into programming (discussed in Section 4)
3) Using modular macros to standardize common data computations and imputations (discussed in Section 5)

Section 2 provides a data example to facilitate the following discussion on these three techniques. Finally, Section 6 puts together all the pieces to illustrate how a real BDS dataset program can become easily maintainable, reusable, and understandable by applying these techniques.

2. A DATA EXAMPLE

Throughout this paper, we will use bone mineral density (BMD) data measured by dual-energy X-ray absorptiometry (DXA) to illustrate the data structures and the discussion of programming for by-visit BDS datasets. However, the methodologies summarized using this data example can be generalized to other similar by-visit data and analyses, such as laboratory tests, vital signs, quality of life (QOL), etc.

A hypothetical clinical study measures bone area (BAREA), bone mineral content (BMC), and BMD by DXA scan at a few planned time points - screening, month 3, 6, 12, 15, 18, 24, 30 and 36. Double measurements are required at screening and month 12. The study analyzes percent change in BMD at each visit relative to the study baseline and to month 12, respectively. It uses the average of the double scans at screening and month 12. When BMD is missing at any post baseline visit, the Last Observation Carried Forward (LOCF) method is used to impute the missing data.

Table 1 displays a portion of the collected data in tabulation format for one subject. Note that the month 18 measurement is not done for this subject. This data has a structure of one or two records per subject per test (BMTESTCD) per bone location (BMLOC) per visit (BMDTC or VISIT) according to the protocol-specified data collection.

Table 1. DXA BMD Tabulation Data (SDTM BM Domain):

Row #	BMTEST	BMTESTCD	BMLOC	BMDTC	VISIT	BMSTRESN	BMSTRESU
T1	Bone Mineral Density	BMD	LUMBAR SPINE	2010-05-14	SCREEN	0.772	g/cm2
T2	Bone Mineral Density	BMD	LUMBAR SPINE	2010-05-14	SCREEN	0.786	g/cm2
T3	Bone Mineral Density	BMD	LUMBAR SPINE	2010-09-17	MONTH 3	0.825	g/cm2
T4	Bone Mineral Density	BMD	LUMBAR SPINE	2010-12-10	MONTH 6	0.837	g/cm2
T5	Bone Mineral Density	BMD	LUMBAR SPINE	2011-06-10	MONTH 12	0.840	g/cm2
T6	Bone Mineral Density	BMD	LUMBAR SPINE	2011-06-10	MONTH 12	0.855	g/cm2
T7	Bone Mineral Density	BMD	LUMBAR SPINE	2011-09-09	MONTH 15	0.886	g/cm2
T8	Bone Mineral Density	BMD	LUMBAR SPINE	2012-06-10	MONTH 24	0.912	g/cm2
T9	Bone Mineral Density	BMD	LUMBAR SPINE	2012-12-14	MONTH 30	0.910	g/cm2
T10	Bone Mineral Density	BMD	LUMBAR SPINE	2013-06-18	MONTH 36	0.880	g/cm2
…	…						

Table 2 illustrates what an ADaM-compliant BDS dataset may look like for the analysis. The BDS analysis dataset called ADBMD is created using the BM domain (Table 1) as the source data. Besides deriving the analysis variables, such as PARAM/CD, AVAL, ADT, AVISIT, BASE, CHG, PCHG, etc., the new data records for (1) the average of double scans, (2) LOCF imputation, and (3) BASETYPE “MONTH 12” must also be derived.

There are three kinds of data records in the ADBMD dataset by ADaM design – source, derived and repeated. Rows T1 to T10 are source data records from the SDTM BM domain. Rows D1 to D3 are derived records with various DTYPE values. Rows R1 to R8 repeat a subset of rows T1 to T10 and rows

D1 to D3 where AVISIT is “MONTH 12” or later, with BASE and PCHG relative to Month 12 instead of the study baseline (i.e. BASETYPE=MONTH 12).

Note that there could be a different implementation of BDS such as creating separate datasets for different BASETYPEs instead of including all BASETYPEs in one dataset. This paper focuses on the one-dataset approach so that the discussion is more complete for typical scenarios.

Table 2. DXA BMD Analysis Data (ADBMD Dataset):

Row #	PARAMCD	BASETYPE	ADT	AVISIT	ABLFL	ANL01FL	DTYPE	AVAL	BASE	PCHG
T1	DBMDLSPA	BASELINE	14May2010	BASELINE				0.772	0.779	
T2	DBMDLSPA	BASELINE	14May2010	BASELINE				0.786	0.779	
D1	DBMDLSPA	BASELINE	14May2010	BASELINE	Y	Y	AVERAGE	0.779	0.779	0.00
T3	DBMDLSPA	BASELINE	17Sep2010	MONTH 3		Y		0.825	0.779	5.91
T4	DBMDLSPA	BASELINE	10Dec2010	MONTH 6		Y		0.837	0.779	7.45
T5	DBMDLSPA	BASELINE	10Jun2011	MONTH 12				0.840	0.779	7.83
T6	DBMDLSPA	BASELINE	10Jun2011	MONTH 12				0.855	0.779	9.76
D2	DBMDLSPA	BASELINE	10Jun2011	MONTH 12		Y	AVERAGE	0.848	0.779	8.79
T7	DBMDLSPA	BASELINE	09Sep2011	MONTH 15		Y		0.886	0.779	13.74
D3	DBMDLSPA	BASELINE	09Sep2011	MONTH 18		Y	LOCF	0.886	0.779	13.74
T8	DBMDLSPA	BASELINE	10Jun2012	MONTH 24		Y		0.912	0.779	17.07
T9	DBMDLSPA	BASELINE	14Dec2012	MONTH 30		Y		0.910	0.779	16.82
T10	DBMDLSPA	BASELINE	18Jun2013	MONTH 36		Y		0.880	0.779	12.97
R1	DBMDLSPA	MONTH 12	10Jun2011	MONTH 12				0.840	0.848	
R2	DBMDLSPA	MONTH 12	10Jun2011	MONTH 12				0.855	0.848	
R3	DBMDLSPA	MONTH 12	10Jun2011	MONTH 12	Y	Y	AVERAGE	0.848	0.848	0.00
R4	DBMDLSPA	MONTH 12	09Sep2011	MONTH 15		Y		0.886	0.848	4.54
R5	DBMDLSPA	MONTH 12	09Sep2011	MONTH 18		Y	LOCF	0.886	0.848	4.54
R6	DBMDLSPA	MONTH 12	10Jun2012	MONTH 24		Y		0.912	0.848	7.61
R7	DBMDLSPA	MONTH 12	14Dec2012	MONTH 30		Y		0.910	0.848	7.31
R8	DBMDLSPA	MONTH 12	18Jun2013	MONTH 36		Y		0.880	0.848	3.77
…	…									

3. CAREFULLY DESIGN THE PROGRAMMING FLOW

An efficient programming flow not only minimizes data processing time, but also helps simplify code and ensures a consistent approach for all the programmers on the team, all of which shortens program development time and improves program maintainability and reusability.

Designing an efficient programming flow for BDS can be complex and challenging. A BDS usually contains multiple types of data records. As shown in Table 2, not all BDS records are directly from the source data. Some are newly derived records for various DTYPEs, and some are repeat records for a baseline type other than the study baseline. Creating a BDS dataset usually means two-dimensional data derivation, both adding new data records and deriving new variables. While the usual variable-based programming flow design works well for horizontal data structures, for the vertical data structure of BDS, both dimensions need to be considered.

Considering both dimensions can be done by first identifying the types of data records. BDS records can generally be split into three groups based on the dependency and the different data sources / derivations used – 1) source records, 2) derived records, and 3) repeat records. Source records come directly from the original data sources and generally need to have only variable-based programming design. Derived records are new records created following the derivation instructions as required for the analyses. Repeat records are copies of existing records (either source or derived) with a change in the derivation of one or more variables on the records. Each group of records is created in a separate programming step. These programming steps are ordered based on dependency to maximize data processing efficiency and code simplicity. In the ADBMD example as illustrated in Table 2, rows T1-T10 are source data records and therefore created in the first step. Row D1-D3 are derived records and depend on the source records. Thus, the D1-D3 are created after T1-T10 are ready. Records R1-R8 for BASETYPE “MONTH 12” depend on both T-T10 and D1-D3 and are therefore created in a third step. Record creation steps are numbered using odd numbers, 1, 3, 5, etc. because variables are derived between the steps or after the last record step.

Once the record derivation order has been determined, the timing to derive each analysis variable can be planned. The variable programming steps are numbered in even numbers, 2, 4, 6, etc. A variable is created before the next record step when (1) the creation of records in the next step depends on the variable or (2) the variable has different definitions between the two record groups and cannot be derived in a same sequence of data manipulations (e.g., in a same data step) after new records are added.

Table 3 summarizes the step-by-step programming flow for ADBMD. Corresponding to the three types of data records, three separate programming steps are identified for the record creation. Step 1 gets source data records. Step 3 derives data records for DTYPE. Step 5 repeats data records for BASETYPE “MONTH 12”. Then based on the variable definitions and data dependency, the variables are grouped into steps 2, 4, or 6.

This approach to the design of the programming flow may be used generally in all BDS datasets. In some complex data and analysis scenarios, there might be more than one step required for one or more of the three types of data records. However, the same criteria can be used to group record types and determine the timing of deriving new variables.

Table 3. Programming Flow for ADBMD, Variables vs. Records:

Step	Description	Detail
1	Read in source data records from SDTM BM domain. (Creates source records)	This step obtains the data records needed from the source data. Merge SDTM BM and SUPPQUAL domain to get source data records with a data structure shown in Table 1. Subset data if necessary (e.g., subset data where BMMETHOD=”DXA SCAN” for DXA BMD). Records are described by only SDTM variables at this step.
2	Derive parameter, result, and timing variables (Add variables to source records)	The new data records in the next step depend on parameter, result, and timing variables, so they need to be derived at this step (PARAM/CD, AVAL, ADT, ADY, & AVISIT/N). When this step is completed, rows T1-T10 in Table 2 will have been created with the new analysis variables listed above.
3	Add data records for DTYPE “AVERAGE” and “LOCF” (Create derived records)	This step adds new records for DTYPE “LOCF” and “AVERAGE”. This step also populates all existing variables (PARAM/CD, AVAL, ADT, ADY, & AVISIT/N) for the new records (since they were derived in the previous step) When this step is completed, rows D1-D3 in Table 2 will be present in addition to rows T1-T10.
4	Derive the record-level analysis flag (Add variables to source and derived records)	This step derives the record-level analysis flag, ANL01FL, because the records for other BASETYPEs (step 5) use the values directly from this step. When this step is completed, rows T1-T10 and D1-D3 in Table 2 are present – same as in the step above.
5	Add data records for BASETYPE (Create repeat records)	This step adds new records for BASETYPE “MONTH 12”. It also populates all existing variables (PARAM/CD, AVAL, ADT, ADY, & AVISIT/N, DTYPE, and ANL01FL) for the new records since they were derived in previous steps. When this step is completed, all rows T1-T10, D1-D3, and R1-R8 in Table 2 will be present.

Step	Description	Detail
6	Derive change and percent change by BASETYPE (Add variables to source, derived, and repeat records)	This last step of data derivation adds variables ABLFL, BASE, CHG, & PCHG. In this example, ABLFL is defined differently for each BASETYPE but it can be handled in one data step here (see program code in Section 6). BASE, CHG, and PCHG have the same algorithms regardless of BASETYPE. When this step is completed, all variables and records needed for the analysis are created.

4. INCORPORATE METADATA INTO THE PROGRAMMING

ADaM metadata are specifications for ADaM datasets and are available before the start of programming. Some metadata are well structured, such as a list of possible codes and decodes in a paired CT set or a list of PARAMCD values in the VLM for the variable AVAL. Such well-structured metadata are ideal for reading into programming. Using metadata directly eliminates manual coding errors, ensures consistency between ADaM datasets and the metadata, helps with consistency within and between studies, and encourages a consistent programming approach from different programmers - all of which improves program maintainability and reusability.

Within ADaM, the use of VLM is specific to BDS datasets. Consistently using VLM to describe AVAL / AVALC in BDS gives better specifications. In addition, when VLM is available, it may become useful as a programming technique. This is especially true for those PARAMCDs that are a 1-1 map to a set of variables from the source data. In the ADBMD example, each PARAMCD corresponds to a combination of source data variables BMTESTCD and BMLOC so that PARAMCD can uniquely describe AVAL. In Table 4, the first three columns, A, B, and C, are from the existing VLM for variable AVAL by PARAMCD. By expanding the VLM table to add columns D and E, a lookup table is created for mapping PARAMCD from BMTESTCD and BMLOC (using columns A, D, and E). Further expanding the table to add column F and G establishes a lookup table to map PARAM and PARCAT1 as well. In the end, we are using the VLM directly to define PARAMCD, PARAM, and PARCAT1 for all records from the source data (the T1-T10 rows in Table 2).

Table 4. Expanding VLM for ADBMD to A Lookup Table:

Existing VLM			Added to Facilitate Programming			
A: PARAMCD	**B: Variable**	**C: Derivation**	**D: BMTESTCD**	**E: BMLOC**	**F: PARAM**	**G: PARCAT1**
DBMDLSPA	AVAL	BM.BMSTRESN when BMTESTCD="BMD" and BMLOC="LUMBAR SPINE"	BMD	LUMBAR SPINE	DXA BMD Lumbar Spine (g/cm2)	BONE MINERAL DENSITY
DARELSPA	AVAL	BM.BMSTRESN when BMTESTCD="BAREA" and BMLOC="LUMBAR SPINE"	BAREA	LUMBAR SPINE	DXA AREA Lumbar Spine (cm2)	BONE AREA
DBMCLSPA	AVAL	BM.BMSTRESN when BMTESTCD="BMC" and BMLOC="LUMBAR SPINE"	BMC	LUMBAR SPINE	DXA BMC Lumbar Spine (g)	BONE MINERAL CONTENTS
DBMDTHIP	AVAL	BM.BMSTRESN when BMTESTCD="BMD" and BMLOC="TOTAL HIP"	BMD	TOTAL HIP	DXA BMD Total Hip (g/cm2)	BONE MINERAL DENSITY
DARETHIP	AVAL	BM.BMSTRESN when BMTESTCD="BAREA" and BMLOC="TOTAL HIP"	BAREA	TOTAL HIP	DXA AREA Total Hip (cm2)	BONE AREA
DBMCTHIP	AVAL	BM.BMSTRESN when BMTESTCD="BMC" and BMLOC="TOTAL HIP"	BMC	TOTAL HIP	DXA BMC Total Hip (g)	BONE MINERAL CONTENTS

Existing VLM			Added to Facilitate Programming			
A: PARAMCD	**B: Variable**	**C: Derivation**	**D: BMTESTCD**	**E: BMLOC**	**F: PARAM**	**G: PARCAT1**
…	…					

The following code illustrates two potential programming approaches to add PARAMCD, PARAM, and PARCAT1, after reading in source data (i.e., step 2 in Table 3). The first example uses the VLM lookup table as discussed here. The second one uses the more traditional if-then statements with manual coding. Obviously, the first approach is more efficient, requires less code maintenance, introduces much less chance of typographical or copy-paste errors resulting in incorrect or missing derived values, and has a significantly lower QC burden:

```
/* Approach #1: merge BM domain with the VLM lookup table to get parameter variables;
work.bm is the BMD data after merging the SDTM BM domain and suppqual domain, see Table 1 for
the data structure;
vlm.adbmd is the VLM lookup table dataset for ADBMD, see Table 4 for the data structure */
     proc Sql;
          create table work.bm2 as
          select    a.*,
                    b.PARAMCD,
                    b.PARAM,
                    b.PARCAT1
          from work.bm a left join vlm.adbmd b
          on a.BMTESTCD=b.BMTESTCD and a.BMLOC=b.BMLOC;
     quit;
/* Approach #2: use a data step and if-then statements to create parameter variables */
     data work.bm2;
          set work.bm;
          if BMTESTCD=”BMD” then do;
               PARCAT1=”BONE MINERAL DENSITY”;
               if BMLOC=”LUMBAR SPINE” then do;
                    PARAMCD=”DBMDLSPA”;
                    PARAM=”DXA BMD Lumbar Spine (g/cm2)”;
               end;
               <repeat for other bone locations, e.g., BMLOC=”TOTAL HIP”>
          end;
          <repeat for BMTESTCD=”BMC”, “BAREA”, etc.>
     run;
```

Similarly, the existing metadata in the CT can be used to eliminate if/then loops and manual coding. A paired CT can be used as a lookup table or converted into a SAS® format / informat, with either form used directly in the programming. Table 5 lists an example paired CT for AVISITN and AVISIT. One side of the paired CT must be derived based on algorithms specified in the Statistical Analysis Plan (SAP; in this example AVISIT is derived as BASELINE, MONTH 3, etc.). Then the other side of the paired CT (in this example AVISITN) can have values assigned using a SAS format / informat created directly from the existing metadata. A data step for assigning AVISITN using the CT looks like the following:

```
     data bmd_tmp2;

         set bmd_tmp1;

         <code to derive AVISIT based on SAP algorithms>;

         AVISITN=input(AVISIT, avisit.); /*avisit is an informat created from the paired CT */
```

```
    run;
```

Note that a key element of this programming technique is assessing which one of the paired variables should be programmed and which should be assigned directly from CT. Consideration should be given in particular to code maintainability and reusability. In this example, AVISITN is assigned from the CT, i.e., deriving AVISIT first and then getting AVISITN from the SAS informat, rather than the other way around, for two reasons. First, code values alone without decodes are meaningless; e.g., a statement "if <*conditions*> then AVISITN=3003;" is hard to interpret and validate because '3003' could mean DAY 3, WEEK 3, MONTH 3, YEAR 3, etc. Using the AVISIT text values makes the program code easier to follow for both future programmers and potential regulatory reviewers. Second, in this particular case, the code values are the ones likely to be updated (e.g., changes to use a 2000-series set of codes instead of the current 3000 series for visits in the monthly unit) during the development of programs. When re-coding occurs, it only requires a simple rerun of the programs to update the data; the program code remains unchanged because it is not manually coded or derived based on any assumptions.

Table 5. A Paired CT for AVISITN and AVISIT:

AVISITN	AVISIT
1000	BASELINE
3003	MONTH 3
3006	MONTH 6
3012	MONTH 12
3015	MONTH 15
3018	MONTH 18
3024	MONTH 24
3030	MONTH 30
3036	MONTH 36

The above programming technique of using metadata directly, such as VLM and paired CT, improves programming efficiency by eliminating the need to write 'if-then' statements to manually code data values or maintain the programming code individually. In addition, data consistency is ensured across studies, and the quality risks due to typos or copy-paste accidents are greatly reduced. Certainly more opportunities of using metadata in programming exist besides the two examples discussed here.

5. USE MODULAR MACROS

Modular programming is a technique that identifies and creates clearly separated modules of code that can be easily maintained and re-used across different projects. Use of macros is one way of doing this. Modular macros are widely used in the statistical programming of clinical trial study data. Macros help standardize program code, ensure consistency across datasets and studies, achieve maximum efficiency and quality, and reduce development time because the same steps do not have to be developed and re-developed by different programmers. Macros may be created and maintained at a study level, a product level, or a department level, depending on the similarity or standardization of the programming algorithms.

BDS are a class of ADaM datasets that provide many opportunities for using modular macros. Several criteria are helpful for determining when a modular approach is a good idea, such as:

1) Derivations that are identical or similar across multiple domains or studies (baseline flag, change, percent change, etc.)

2) Derivations that are complex (imputations, derivation of summary scores, etc.)

Below, we list some examples where modular macros may be used for BDS:

- **Macro for the study baseline**

  Many data share a similar definition for the study baseline (e.g., last observation prior to the first dose) which makes it an ideal candidate for modular programming.

- **Macro for analysis visit**

  Analysis visits are generally derived in multiple datasets within a study and across studies. The definitions are usually structured similarly (e.g., using analysis visit windows defined in SAP along with consistent rules around handling multiple data points within the same window) and can be standardized in a modular program.

- **Macro for the record-level analysis flags**

  When the use of flag variables, ANLzzFL, is standardized, a macro for one, or a set, of flags is possible.

- **Macro for change and percent change from baseline**

  Calculating change and percent change from baseline is common with identical algorithms in many BDS datasets, which makes the derivation ideal for modular programming.

- **Macro for identifying missing visits**

  The process to identify the missing visits is generally the same regardless of the imputation method that will follow, thus making it a good candidate for modular programming as an input into subsequent imputation algorithms.

- **Macro for each type of derived records (e.g. LOCF, AVERAGE, etc.)**

  Data derivation for a given method (DTYPE) is usually identical across endpoints and studies. Since each DTYPE is defined and derived differently, it is most practical to have a separate macro for each of them.

6. WHAT DOES THE FINAL ADBMD PROGRAM LOOK LIKE?

Combining all programming techniques discussed in the previous sections, the program below illustrates the creation of the BDS dataset ADBMD. Steps 1-6 in the program follow the programming flow discussed in Section 3 and correspond to the structure detailed in Table 4.

```
/****************************

Step 1: Get source data records from SDTM;

A product-level macro to initialize the program;

A department-level modular macro is used to merge SUPPQUAL domain into the standard domain BM

*****************************/

        %include “init.sas”;

        %m_sdtm_merge (indsn_=sdtm.bm, outdsn_=dxabmd1, subset_=%str(bmmethod=”DXA SCAN” and
        bmstresn^=.));

/****************************

Step 2: Create analysis variables for the source records, PARAM/CD, PARCAT1, AVAL, ADT, ADY, &
AVISIT/N.

Use VLM as a lookup table to get PARAM/CD and PARCAT1, which is done by a department-level
modular macro %m_get_param.

Use a paired CT as a lookup table to define AVISITN, which is done by a product-level modular macro
%m_avisit.

A department-level modular macro %m_isodate is used to convert the ISO dates to numeric dates.
```

```
*****************************/
        %m_get_param (indsn_=dxabmd1, outdsn_=temp21, vlm_=vlm.adbmd, incol_=%str(BMTESTCD / BMLOC),
                       outcol_=%str(PARAMCD / PARAM / PARCAT1));
        data temp22;
           set temp21;
           %m_isodate(indate_=bmdtc, outdate_=adt);
           ady=bmdy;
           aval=bmstresn;
        run;
        %m_avisit (indsn_=temp22, outdsn_=dxabmd2, lut_=lookup.avisit, endp_=BMD); /* assuming lookup.avisit
        is a lookup table for analysis windows */
/****************************
Step 3: Create derived records for LOCF and AVERAGE; add variable DTYPE.
Product-level modular macros are used for these two derivations separately.
Note AVERAGE records derived before LOCF records due to the dependency.
*****************************/
        %m_records_average (indsn_=dxabmd2, outdsn_=temp31, avisit_=%(BASELINE / MONTH 12));
        %m_records_locf (indsn_=temp31, outdsn_= dxabmd3, lut_=lookup.avisit,
                         sort_=%str(usubjid paramcd avisitn adt dtype));
/****************************
Step 4: Define variable ANL01FL using a product-level modular macro
*****************************/
        %m_anl01fl (indsn_= dxabmd3, outdsn_=dxabmd4, average_=yes, locf_=yes);
/****************************
Step 5: Create repeat records for BASETYPE=MONTH 12; add variable BASETYPE
*****************************/
        data dxabmd5;
          set dxabmd4;
          length basetype $20;
          basetype=”BASELINE”;
          output;
          if avisitn >= input (“MONTH 12”, avisit.) then do;
              basetype=”MONTH 12”;
              output;
          end;
        run;
```

```
/*****************************
Step 6: Derive variables, ABLFL, BASE, CHG, and PCHG
ABLFL is defined differently for two BASETYPEs, but can be done in one data step, so created together
here (a key to data processing efficiency); use a product-level macro to derive BASE, CHG, and PCHG
*****************************/
        data temp61;
           set dxabmd5;
           if anl01fl="Y" and ((basetype="BASELINE" and avisit="BASELINE") or
                              (basetype="MONTH 12" and avisit="MONTH 12")) then ablfl="Y";
        run;
        %m_chg (indsn_=temp61, outdsn_= dxabmd6, sort_=%str(usubjid paramcd basetype avisitn adt ablfl));
/*****************************
Final step: save output dataset to a permanent one;
Use a department-level macro that contains a few common processes for all ADaM datasets:
get core variables; assign variable attributes; run data quality checks, etc.
*****************************/
        %m_adam_outputer (indsn_= dxabmd6, outdsn_=adam.adbmd, metadata_=mddt.adbmd,
        study_=study0001);
```

7. CONCLUSION

ADaM BDS datasets can grow large and complex due to the vertical designs. Although there are some programming challenges associated with BDS, BDS datasets can still be programmed with high efficiency and quality through forward thinking and planning of programming ahead of time. Several useful techniques can help achieve these goals and save programming cost. Careful design of program flow, maximizing the use of metadata, and using modular macros are important programming techniques for BDS.

REFERENCES

CDISC ADaM Implementation Guide version 1.0

Define.xml version 2.0

ACKNOWLEDGMENTS

The author wishes to thank Erin Christen for spending significant time to help review and edit the paper. The author also appreciates the thoughtful review comments from Michiel Hagendoorn. Many ideas, considerations, and examples mentioned in the paper are based on real-life ADaM implementation carried out by the Amgen bone programming team; thank you team.

CONTACT INFORMATION

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

Ellen Lin

E-mail: zlin@amgen.com

Phone: 805 447 1018