Implementation of Define.XML Generators Using Hash Objects and User-Defined Functions

Lei Zhang, Celgene Corp., Summit, NJ

ABSTRACT

Define.xml, as a data definition file for clinical study data, is one of key documents required for electronic submissions of CDISC standard datasets, such as SDTM and ADaM. However, generating a define.xml file is not an ordinary task, it poses a new challenge to many SAS programmers. In this paper, we explore a new way to develop a define.xml generator by leveraging new programming concepts and facilities in SAS®9.

The paper is organized as follows. It first reviews define.xml elements and the challenges of creating a define.xml file, and then proposes a new model for the define.xml metadata repository using object-relational (OR) data modeling concepts. Next, it introduces a lightweight framework called CCAT for the development of define.xml generators. The CCAT framework is implemented with the hash objects and user-defined functions in SAS®9. This paper provides comprehensive examples to illustrate how to create define.xml v2.0 elements and form a define.xml file with the CCAT utilities.

This paper will be very helpful for SAS programmers who want to learn how to design a define.xml generator and other similar applications, and how to implement these applications with advanced programming concepts and the CCAT framework.

INTRODUCTION

CDISC define.xml file is one of the key documents required by FDA for electronic submissions of CDISC datasets, such as SDTM and ADaM. As a road map to guide regulatory reviewers in their exploration of clinical study data submitted, the define.xml file is required to clearly describe the structures, contents and relationships between various components of CDISC datasets in XML format, and help reviewers to easily access and understand the clinical data they are reviewing. This poses a lot of challenges to many SAS programmers. In order to overcome many difficulties that they may encounter during the creation of define.xml files, this paper presents a new model for define.xml metadata repository and a new framework called CCAT for the development of a define.xml generator. The CCAT framework is implemented with hash objects and user-defined functions and subroutines in SAS® 9. Because of space limitation, it is assumed that readers have a good understanding of Define-XML specifications and related CDISC standards.

CHALLENGES IN IMPLEMENTING A DEFINE.XML GENERATOR

A define.xml file provides metadata for domains, datasets, variables, values and associated information in a clinical study. It contains the following elements according to CDISC Define-XML 2.0 specification:

- XML header, including the ODM root, Study and MetaDataVersion elements.
- Dataset and variable definitions.
- Value definitions with associated Where clause definitions
- Controlled Terminology definitions and their associations with external coding dictionaries
- Computational method and comment definitions
- ADaM analysis results metadata (under development)
- Hyperlinks and associations among the above elements and with external documents, such as links to annotated Case Report Forms in SDTM and supplemental data definitions.

For the detailed information about define.xml structures and their explanations, please see [1] [2][3]. Although the first version of Define-XML specification was published as early as in 2004, implementing a properly functioning define.xml generator still faces many challenges. This is because

1. A define.xml file is a structured document. It has to be constructed in XML format according to the CDISC Define-XML specification.
2. A define.xml file involves a large amount of comprehensive metadata about a clinical study. It not only needs attribute metadata such as variable type, length and label, that can be automatically extracted from SAS Dictionary views, but also demands the metadata about the linkages and relationships among the various data components that user have to provide manually.

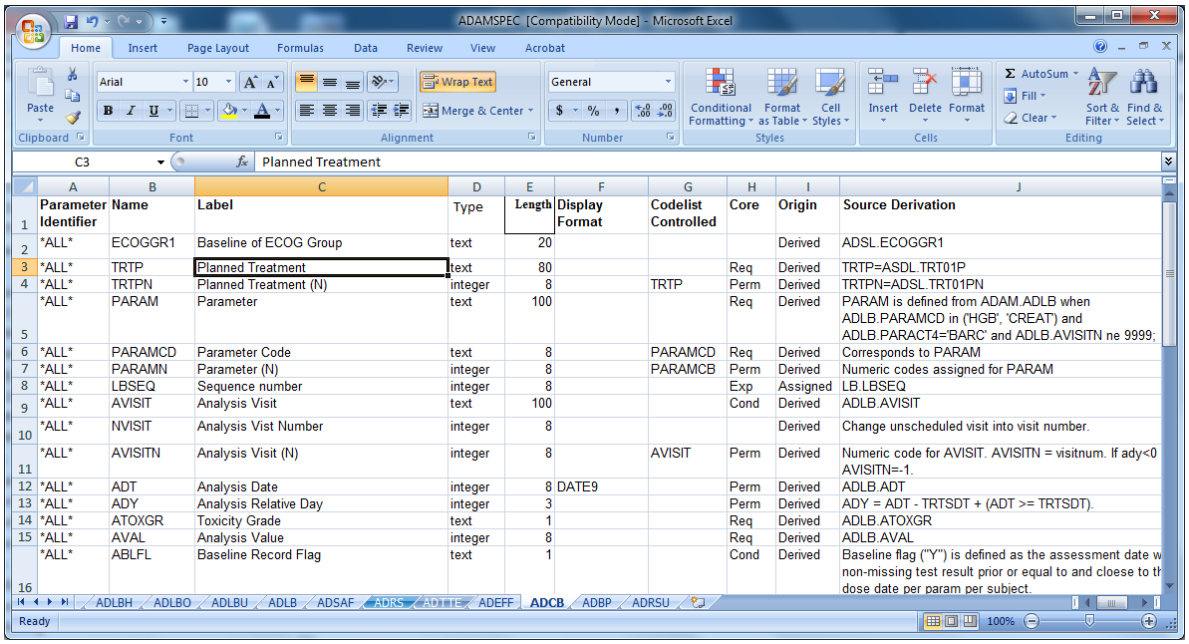
3. The raw metadata that users create for the define.xml generation are hardly normalized, and quite difficult to be normalized according to relational database principle. They even contain complex values that have to be further processed for proper use.

4. Different define.xml files are needed for the clinical data with different purposes. For example, a define.xml file for SDTM datasets is different from the one for the ADaM datasets, although both of them use the same Define-XML specification.

5. CDISC Define-XML specification is still evolving. A newer version may introduce new elements at the same time modify and depreciate existing ones.

AN OBJECT-RELATIONAL DATA MODEL FOR DEFINE.XML

In order to implement a define.xml generator, one of the key steps is to define the way that raw metadata have to be collected and prepared. Although SAS Dictionary views contain a significant amount of information about the attributes of datasets and their variables, they are far from enough for a generator to create a complete define.xml file. A common approach for the raw metadata collection is to create a set of Excel spreadsheets based on SAS Dictionary views initially and then let users to manually add additional metadata needed for the define.xml generator [4]. This set of Excel spreadsheets are often referred to as metadata sheets or specifications. Below is a snapshot of ADaM metadata sheets.



	A	B	C	D	E	F	G	H	I	J
1	Parameter Identifier	Name	Label	Type	Length	Display Format	Codelist Controlled	Core	Origin	Source Derivation
2	*ALL*	ECOGGR1	Baseline of ECOG Group	text	20				Derived	ADSL.ECOGGR1
3	*ALL*	TRTP	Planned Treatment	text	80			Req	Derived	TRTP=ASDL.TRT01P
4	*ALL*	TRTPN	Planned Treatment (N)	integer	8		TRTP	Perm	Derived	TRTPN=ADSL.TRT01PN
5	*ALL*	PARAM	Parameter	text	100			Req	Derived	PARAM is defined from ADAM.ADLB when ADLB.PARAMCD in ('HGB', 'CREAT') and ADLB.PARACT4='BARC' and ADLB.AVISITN ne 9999;
6	*ALL*	PARAMCD	Parameter Code	text	8		PARAMCD	Req	Derived	Corresponds to PARAM
7	*ALL*	PARAMN	Parameter (N)	integer	8		PARAMCB	Perm	Derived	Numeric codes assigned for PARAM
8	*ALL*	LBSEQ	Sequence number	integer	8			Exp	Assigned	LB.LBSEQ
9	*ALL*	AVISIT	Analysis Visit	text	100			Cond	Derived	ADLB.AVISIT
10	*ALL*	NVISIT	Analysis Vist Number	integer	8				Derived	Change unscheduled visit into visit number.
11	*ALL*	AVISITN	Analysis Visit (N)	integer	8		AVISIT	Perm	Derived	Numeric code for AVISIT. AVISITN = visitnum. If ady<0 AVISITN=-1.
12	*ALL*	ADT	Analysis Date	integer	8	DATE9		Perm	Derived	ADLB.ADT
13	*ALL*	ADY	Analysis Relative Day	integer	3			Perm	Derived	ADY = ADT - TRTSDT + (ADT >= TRTSDT).
14	*ALL*	ATOXGR	Toxicity Grade	text	1			Req	Derived	ADLB.ATOXGR
15	*ALL*	AVAL	Analysis Value	integer	8			Req	Derived	ADLB.AVAL
16	*ALL*	ABLFL	Baseline Record Flag	text	1			Cond	Derived	Baseline flag ("Y") is defined as the assessment date w non-missing test result prior or equal to and cloese to th dose date per param per subject.

The metadata sheets users create usually have following characteristics:

- The metadata in the sheets are loosely organized by putting the closely related columns in the same sheet, and less or unrelated ones in different sheets in order to best suit users to enter metadata. They are hardly normalized, and in some cases difficult to be normalized with the relational database principle.
- Most of metadata in the sheets contain simple numeric and character values, but some of them do contain complex values that contain subcomponents needed in a define.xml file. For example, computational method column may contain the description of an algorithm as well as the links to the related documents. However, the links have to be extracted and treated as separate elements in the define.xml file.
- Rows within a group of columns in a metadata sheet may even have different meanings, or interpretations, dependent on the values of the other columns. This is reflected in the Parameter Value List element of the Define-XML 2.0 specification.

In order to convert a user-defined metadata specification into a define.xml file, a data model, which consists of a set of datasets, has to be created to serve as a repository for the metadata sheets. Choosing an appropriate data model for the repository is very important for a define.xml generator, because it determines the structure and behavior of a define.xml generator in a large degree.

As mentioned before, metadata sheets are often non-normalized and may contains complex data with sub-structure needed by a define.xml file, we therefore proposes an object-relational (OR) data model for a define.xml generator, because we think the OR model incorporates the object-oriented programming features into the relational database, and can have following advantages:

- It allows user-defined types. Therefore, it is easy for the data modeling and the representation is more close to reality and user perception.
- The logical structure it presents is natural enough to be understood by both developers and users
- It makes a define.xml generator easy to develop, maintain and extend.

For more information about the OR data model, please see [5].

The major difference between an object-relational database and relational database is that the object-relational database allows user-defined data types and methods. Although the OR model is a good choice to represent define.xml metadata, but the SAS system is not an object-relational database system. In order to solve this problem, we suggest using numeric and character variables with the associated user-defined functions (UDFs) to simulate the user-defined types in the OR model. Below is an OR model we've created for the ADaM define.xml metadata sheets according to the CDISC Define-XML 2.0 specification. Due to the space limitation, we mainly focus on the model variables (or columns) that need to be treated like a user-defined variable with associated user-defined functions, and drop the routine business rules and constraints involved.

1. HDR, which supports the generation of XML elements for the XML header, study, global variables, metadata version and supplement document links. It is designed as a vertical structure as follows.

```
CREATE TABLE ADAM.HDR
   ( HDRKEY     char(32)    /* Key                                  */
     HDRVALUE   char(256)   /* Value                                */
 );
```

   Note that, while most of key/value pairs in the HDR dataset are plain text, the key/value pair that stores the supplement document references is treated as a user-defined variable. It will contain a semicolon-delimited list of IDs referencing to external documents for the creation of supplemental document elements.

2. DOMAIN, which contains the domain level metadata, supports the generation of ItemGroupDef elements, and associated MethodDef, and def:CommentDef elements. It has following structure:

```
CREATE TABLE ADAM.DOMAIN

   ( DOMNAME    char(32)   /* Domain, or dataset name                  */
    ,DOMDESC    char(256)  /* Domain description                       */
    ,DOMSTRUCT  char(128)  /* Domain structure                         */
    ,DOMKEYS    char(1024) /* Domain keys                              */
    ,DOMCLASS   char(16)   /* Domain class                             */
    ,DOMREP     char(3)    /* Repeating records per subject? Yes, or No */
    ,DOMISREF   char(3)    /* Containing reference data? yes,or no     */
    ,DOMCMNT    char(8192) /* Comment on domain                        */
    ,DOMRANK    num        /* Domain rank                              */
);
```

   Note that, two variables in this dataset are identified as user-defined variables: `DOMKEYS` keeps an ordered list of variable names as the dataset keys; `DOMCMNT`, contains comments on domain, optionally followed by one or multiple links to the internal elements, or external documents.

3. DOMVARS, which contains variable level metadata for each of domains, supports the generation of ItemDef elements, and assocated def:WhereClauseDef, MethodDef, and def:CommentDef elements. It has following structure:

```
CREATE TABLE ADAM.DOMVARS

   ( DOMNAME    char(32)   /* Domain, or dataset name                  */
    ,VARNAME    char(32)   /* Variable name                            */
    ,VARLABEL   char(256)  /* Variable label                           */
    ,VARTYPE    char(32)   /* Variable type                            */
    ,VARLEN     num        /* Variable length                          */
    ,VARFMT     char(32)   /* Variable format                          */
    ,VARORDER   char(3)    /* Variable order                           */
    ,VARCT      char(128)  /* Controlled terminology for variable      */
    ,VARORIGIN  char(32)   /* Variable or                              */
    ,VARCMNT    char(8192) /* Comment on variable                      */
    ,VARREQD    char(3)    /* Is the variable required? yes, or no     */
```

```
);
```

Note that, VARCMNT will be treated as a user-defined variable similar to DOMCMNT above.

4. PVL, which contains metadata for the generation of def:ValueListDef and def:WhereClauseDef elements, has following structure

```
CREATE TABLE ADAM.PVL

   ( DOMNAME     char(32)   /* Domain, or dataset name                */
    ,VARNAME     char(32)   /* Variable name                          */
    ,PVLID       char(32)   /* Parameter value list identifier (ID)   */
    ,PVLWC       char(1024) /* Where clause                           */
    ,PVLORIGIN   char(16)   /* Origin for the parameter value list    */
    ,PVLCMNT     char(1024) /* Comment for the parameter value list   */
    ,PVLREQD     char(3)    /* Is the parameter value list? yes,or no */
);
```

Note that, PVLCMNT will be treated as a user-defined variable similar to DOMCMNT too. Another user-defined variable is PVLWC, which will contain a set of range checks for the Where clause elements.

5. CT, which contains controlled terminology metadata, supports the generation of codelist elements . It has following primary structure

```
CREATE TABLE ADAM.CT

   ( CTID        char(32)   /* Controlled terminology (CT) identifier */
    ,CTNAME      char(32)   /* CT name                                */
    ,CTTYPE      char(8)    /* CT type                                */
    ,CTCODE      char(256)  /* CT code                                */
    ,CTDECODE    char(256)  /* CT decode                              */
    ,CTRANK      num        /* CT rank                                */
);
```

Note that, CT dataset can have association with NCI/CDISC Controlled Terminology dictionary, or other external dictionaries if any. The structure forNCI/CDISC Controlled Terminology dictionary in this implementation has following structure:

```
CREATE TABLE ADAM.SDTM_NCI_TERMS
   (
     CDISC_Definition Char(512)       /* CDISC Definition               */
     ,CDISC_Submission_Value Char(64) /* CDISC submission value         */
     ,CDISC_Synonyms Char(128)        /* CDISC synonyms                 */
     ,NCI_Preferred_Term Char(512)    /* NCI preferred term             */
     ,TERM_Name Char(64)              /* Term name                      */
     ,TERM_Code Char(6)               /* Term code                      */
     ,TERM_GroupCode Char(6)          /* Term group code                */
     ,TERM_Extensible Char(3)         /* Is the tem extensible          */
);
```

6. LINK, which contains metadata for external links, supports the generation of def:leaf elements. It has following structure.

```
CREATE TABLE ADAM.LINK
   (
     LeafID Char(32)         /* Link ID                                */
     ,LeafTITLE Char(256)    /* Link title                             */
     ,LeafHREF Char(1024)    /* Link definition                        */
);
```

If you want to develop a define.xml generator that supports ADaM analysis results metadata elements, you can add another dataset to the above data model with required variables, with either vertical or horizontal structure.

The variables with complex values noted in the above model will be simulated with user-defined functions, or a segment of code. We will show you how to implement them in the next sections.

A LIGHTWEIGHT FRAMEWORK For DEFINE.XML GENERATORS

Since SAS® 9, two important programming constructs have been introduced in the data steps. One is user-defined function or UDF, the other is hash object. UDF, implemented via PROC FCMP facility, lets users create their own functions, or subroutines to manipulate one or multiple variables. It is a very useful facility to improve the reusability, modularity, and extensibility of a SAS program or macro, it can be roughly regarded as a type extension mechanism for the SAS programs. Hash Objects let users directly bring the datasets into memory and access them at any points

of the data step execution without any prior data sorting and merging. The direct data-addressing mechanism it brings about can greatly facilitate data retrieval. For more information on how to use them, please see [6] [7] respectively.

By leveraging the user-defined functions and hash objects, we've implemented a lightweight generic framework called CCAT for define.xml generators. The CCAT framework consists of four primary macros: %CCAT, %XOUT, %DefineHash, and %ToDEFXML. They are built on the user-defined functions and subroutines and can be conveniently used to create XML elements through data steps.

1. %CCAT – Create a formatted string with CAT functions

%CCAT is a data step macro. It has two arguments,

```
%CCAT(template, args)
```

Where the first parameter defines a template, or format string, and the second parameter is a list of character variables, or expressions separated by spaces. The template is simply a mini-document with "placeholders" in it. The "placeholder" is expressed with the meta-character @ that tells the macro where to insert the data generated from the corresponding variable, or expression of the second parameter. Below is a simple example.

```
dsname="  adqsadas  "; fileext="xpt";
xml=%CCAT("<def:title>@.@</def:title>", dsname fileext);
put xml;
```

Output from the data step: `<def:title>  adqsadas.xpt</def:title>`

The %CCAT is simply implemented through a user-defined function called CCAT() and %Sysfunc().

```
%Macro  CCAT(
     template         /*  Template string    */
     ,argument_list   /*  a space-delimited list of variables, and/or expressions */
);
%sysfunc(ccat(%qsysfunc(dequote(&template)), &argument_list))
%Mend;
```

Please note, although the leading space returned from a variable or expression will be kept, %CCAT trims the trailing spaces.

2. %XOUT - Assign a formatted string to a variable and output it to a dataset

%XOUT is also a data step macro. As an extension from %CCAT(), it creates an indented text line for a specified variable and then output the text variable to a dataset. It has five arguments:

```
%XOUT(indent, template, args, linevar=LINE, dsname=)
```

Where the first parameter uses the following indentation instruction:

```
Tn  -   The number of tabs
Sn  -   The number of spaces
```

Where $n >= 0$.

The second and third parameters are same as these defined in the %CCAT(). The fourth parameter is the name of a character variable that is used to store the text output. By default, the default name is LINE. The fifth parameter is optional, you can use it to assign a dataset to be outputted to. Here is a simple example:

```
dsname="adqsadas"; fileext="xpt";
%XOUT(T2, "<def:title>@.@</def:title>", dsname fileext);
put xml;
```

The above code , once executed in a data step, will create a text line like previous example but with two tab characters at the beginning.

%XOUT is implemented with following code.

```
%MACRO XOUT(
   INDENT          /*  Indentation mini command                          */
  ,TEMPLATE        /*  Text expression to be assigned , and outputted.   */
  ,ARGS            /*  A list of variable or expressions separated by spaces */
  ,OUTVAR=LINE     /*  Output Text variable to be assigned. Default is LINE. */
  ,OUTDSN=         /*  Output dataset name                               */
);
```

```
%local indentcmd ch;

%if %length(&TEMPLATE) %then %do;
  %let TEMPLATE=%qsysfunc(dequote(&TEMPLATE));
%end;

%let indentcmd=;
%if %length(&INDENT) %then %do;
  %let ch=%upcase(%substr(&INDENT,1,1));
  %if &ch=T %then %do;
    %let indentcmd=repeat(Byte(9),%eval(%substr(&Indent,2) -1));
  %end; %else %if &ch=S %then %do;
    %let indentcmd=repeat(' ',%eval(%substr(&Indent,2) -1));
  %end;
%end;

%if %length(&TEMPLATE) > 0 %then %do;
  %if %length(&indentcmd) > 0 %then %do;
    &OUTVAR=&indentcmd||%sysfunc(CCAT(%superq(TEMPLATE),&ARGS));OUTPUT &OUTDSN;
  %end; %else %do;
    &OUTVAR=%sysfunc(CCAT(%superq(TEMPLATE),&ARGS));OUTPUT &OUTDSN;
  %end;
%end;%else %do;
  &OUTVAR="";OUTPUT &OUTDSN;
%end;
%MEND;
```

3. %DefineHash – Define a hash object based on a dataset

%DefineHash, as a data step macro, provides a convenient way for a user to define a hash object and its associated iterator. It has three parameters, and is implemented with user-defined function DefineHash() and %sysfunc. Below is its definition:

```
%Macro DefineHash(
   HasHexp     /*  Hassh[:Hash-iterator] (required)                    */
  ,DSNEXP      /*  DSN[key1 key2 . . . /var1 var2 . . .] (required)    */
  ,HashOpts    /*  Hash options such as multidata:'yes'                */
);
%sysfunc(DefineHash(&HashExp,&DSNEXP,&HashOpts));
%Mend;
```

Two data step code patterns can be used with %DefineHash when using hash objects.

Pattern 1: Traversal over all key/value pairs in a single hash object

```
%DefineHash(h1:iter1,  …);
. . .

RC1=Iter1.first()
DO While (rc1 =0);
  /* Data  manipulation */
  . . .
  RC1=ITER1.NEXT();
END;
```

Pattern 2: Traversal over all the key/value pairs in a parent hash object and its child hash objects

```
%DefineHash(parent:parent_iter,  …);
%DefineHash(child:child_iter,  …);
. . .

RC1=parent_iter.first()
DO While (RC1 =0);
       /* Manipulating variables in the master dataset */
    . . .

    RC2=child.find();
    DO WHILE(RC2 = 0);
       /* Manipulating varaibles in the slave dataset */
      . . .
      RC2 = child.find_next();
    END;
     . . .

    RC1=parent_iter.next();
END;
```

Below is a complete example that uses the first pattern to creates a XML dataset for the def:leaf element.

```
DATA DEFLIB.leaf(keep=LINE);
  LENGTH LINE $8192;
  %DefineHash(H1:ITER1,MODEL.LINK[leafID], multidata:'n');

  RC1=ITER1.FIRST();
  DO WHILE (RC1=0);
    If lengthn(LeafTitle)=0 then LeafTitle=LeafID;
    %XOUT(T1, "<def:leaf ID=@ xlink:href=@>", Quote(CATX(".","LF",LeafID)) QStrip(lowcase(LeafHref)))
    %XOUT(T2, "<def:title>@</def:title>", ESCS(STRIP(LeafTITLE)))
    %XOUT(T1, "</def:leaf>")
    RC1=ITER1.NEXT();
  END;
Run;
```

For more examples of using %CCAT, %XOUT and %DefineHash, please see the next section.

4. %ToDEFXML - Create a define.xml file from a list of XML datasets

%ToDEFXML help you create the final define.xml file from a list of XML datasets created with %CCAT, %XOUT, and %DEFINEHASH. It is defined as follows.

```
%MACRO TODEFXML(
  DSNLST               /* A list of XML datasets (required)                  */
  ,OUTDIR=             /* Output directory for the define.xml file (required) */
  ,OUTFILE=DEFINE.XML  /* Define file name, default is DEFINE.XML (required)  */
  ,LINESIZE=8192       /* Line size. Default is 8192                         */
);

DATA _NULL_;
  LENGTH LINE $&LINESIZE _len_  8.  _ERRMSG_ $1024;
  file "&OUTDIR\&OUTFILE" notitles lrecl=&LINESIZE;
  set &DSNLST end=last;
  _len_=lengthn(trimn(LINE));
  if _len_ > &LINESIZE then do;
    _ERRMSG_=%CCAT("ER%str(ROR): the length of XML statement (=@)is longer than &LINESIZE.", _len_);
    Put _ERRMSG_;
    _len_=&LINESIZE;
  end;
  put line $varying&LINESIZE.. _len_;
  if last then do;
      line=repeat(byte(9),2)||"</MetaDataVersion>";_len_=lengthn(LINE);put line $varying&LINESIZE.. _len_;
      line=repeat(byte(9),1)||"</Study>";_len_=lengthn(LINE);put line $varying&LINESIZE.. _len_;
      line="</ODM>";_len_=lengthn(LINE);put line $varying&LINESIZE.. _len_;
  end;
Run;
%MEND;
```

The CCAT framework also includes a set of user-defined text manipulation utilities such as escaping the XML control characters, and quoting trimmed strings. Please see their definitions and usages in the appendix.

The data step macros and user-defined functions in the CCAT framework are designed to be used inside SAS data steps like a built-in functions. They are non-intrusive, do not require users to create specific data structures in order to use it, and have no external dependencies. The idea behind them is to separate the specification of the business logic and computation required to generate xml elements from the specification of how these chunks of xml text is presented. Because of this, the CCAT framework brings following extra benefits:

- It simplifies the development of a define.xml generator.
- It encourages modular programming through user-defined functions or subroutines.
- It separates XML templates from business logics.
- It encourages the development of reusable XML templates for a family of define.xml generators.
- It lets XML templates serve as clear documentation for the implementation of define.xml elements.

With the CCAT framework, a define.xml generator can be developed with following steps:

1. Convert user-defined metadata sheets (usually in Excel) into OR model datasets via PROC IMPORT facility.
2. Convert study datasets into .XPT files with XPORT library engine and PROC COPY.

3. Create user-defined functions and subroutines via PROC FCMP facility for the variables with compound or complex values.

4. Create individual XML datasets using %CCAT, %XOUT and %DefineHash, together with the user-defined functions associated with complex or compound variables.

5. Call %ToDEFXML to combine all the XML datasets generated and create the final DEFINE.XML file.

In the next section, we will show you how the elements defined in the Define-XML specifications v2.0 can be created using this framework.

IMPLEMENTATION OF DEFINE-XML V2.0 ELEMENTS

In this section, we will use CCAT utilities to create individual Define-XML v2.0 elements based on the OR model proposed in Section 3. Because of space limitation, please see their XML structures and examples in the corresponding sections of the Define-XML Specification v2.0 published in March 2013.

Generating XML Header Elements

The Define-XML specification is an extension to the CDISC ODM standard, so as an instance, a define.xml file should have XML header, stylesheet reference, and ODM element like any ODM file. Below is a sample implementation of the XML header with CCAT utilities and the HDR dataset.

```
DATA DEFLIB.HDR(keep=LINE);
  LENGTH LINE $8192;
  LENGTH  _SUPPID_ $64 _IDX_ 8.;
  %DefineHash(H1,MODEL.HDR[HDRKEY], multidata:'n');

  %XOUT(T1, "<?xml version=""1.0"" encoding=""ISO-8859-1""?>")
  %XOUT(T1, "<?xml-stylesheet type=""text/xsl"" href=""define-adam-draft-06.xsl""?>")
  %XOUT(T1, "<ODM xmlns=""http://www.cdisc.org/ns/odm/v1.2""")
  %XOUT(T2, "xmlns:xsi=""http://www.w3.org/2001/XMLSchema-instance""")
  %XOUT(T2, "xmlns:xlink=""http://www.w3.org/1999/xlink""")
  %XOUT(T2, "xmlns:def=""http://www.cdisc.org/ns/def/v1.0""")
  %XOUT(T2, "xmlns:adamref=""http://www.cdisc.org/ns/ADaMRes/DRAFT""")
  %XOUT(T2, "xsi:schemaLocation=""http://www.cdisc.org/ns/odm/v1.2 util/adamres-draft2.xsd""")
  HDRKEY="FILEOID";HDRVALUE="";H1.find();
  %XOUT(T2, "FileOID=@", QSTRIP(HDRVALUE));
  %XOUT(T2, "ODMVersion=""1.2""")
  %XOUT(T2, "FileType=""Snapshot""")

  %XOUT(T2, "CreationDateTime=@>",qstrip(put(datetime(),is8601dt.)))
  HDRKEY="STUDYOID";HDRVALUE="";H1.find();
  %XOUT(T3, "<Study OID=@>", QSTRIP(HDRVALUE))
  %XOUT(T4, "<GlobalVariables>")
  HDRKEY="STUDYNAME";HDRVALUE="";H1.find();
  %XOUT(T4, "<StudyName>@</StudyName>", HDRVALUE);

  HDRKEY="STUDYDESC";HDRVALUE="";H1.find();
  %XOUT(T4, "<StudyDescription>@</StudyDescription>", HDRVALUE)
  HDRKEY="STUDYPROTOCOL";HDRVALUE="";H1.find();
  %XOUT(T4, "<ProtocolName>@</ProtocolName>", HDRVALUE)
  %XOUT(T4, "</GlobalVariables>")
  %XOUT(T4, "<MetaDataVersion OID=""CDISC.ADaM.2.1""")
  HDRKEY="DEFINENAME";HDRVALUE="";H1.find();
  %XOUT(T5, "Name=@", QSTRIP(HDRVALUE))
  HDRKEY="DEFINEDESC";HDRVALUE="";H1.find();
  %XOUT(T5, "Description=@", QSTRIP(HDRVALUE))
  %XOUT(T5, "def:DefineVersion=""1.0.0""")
  %XOUT(T5, "def:StandardName=""CDISC ADaM""")
  %XOUT(T5, "def:StandardVersion=""2.1"">")
  %XOUT()

  HDRKEY="SUPPIDS";HDRVALUE="";H1.find();
  IF lengthn(HDRVALUE) > 0 then Do;
```

```
    _IDX_=1;
    _SUPPID_=scan(HDRVALUE,_IDX_,"+");
    %XOUT(T5, "<def:SupplementalDoc>")
    Do While (lengthn(_SUPPID_)> 0);
      %XOUT(T6, "<def:DocumentRef leafID=@/>", QStrip(_SUPPID_));
      _IDX_=_IDX_ + 1;
      _SUPPID_=scan(HDRVALUE,_IDX_,"+");
    End;
    %XOUT(T5, "</def:SupplementalDoc>")
  END;
Run;
```

Apart from XML header elements, the above data step module implements other study related elements, such as Study, Globalvariables, Studyname, StudyDescription, ProtocolName, and MetaDataVersion. What is more, it implements the def:SupplementalDoc element by treating "SUPPIDS" key/value pair as a complex variable that consists of a list of supplemental document IDs separated by semi-colons. Because the module defines a hash object with %DefineHash, you will see how easy it is to access a value in the HDR dataset with a hash object.

Generating def:ValueListDef Elements

The *def:ValueListDef* is an extended element with the Define-XML schema. It is the container for child *ItemRef* elements that link to *ItemDef* elements for a complete definition of a Value-level Variable. It is required for each unique value of the ValueListOID attribute within the MataDataVersion. Below is its implementation with the PVL dataset.

```
DATA DEFLIB.ValueListDef(keep=LINE);
    LENGTH LINE $8192;
    LENGTH TMP1 TMP2 TMP3 TMP4 $128;

    %DefineHash(H1:ITER1,MODEL.PVL[DOMNAME VARNAME], multidata:'n');
    %DefineHash(H2      ,MODEL.PVL[DOMNAME VARNAME], multidata:'y');

    RC1=ITER1.FIRST();
    DO WHILE (RC1=0);

      TMP1=cats(DOMNAME,".",VARNAME);
      %XOUT(T1, "<def:ValueListDef OID=@>", qstrip("VL."||strip(TMP1)));
      RC2= H2.FIND();
      DO WHILE(RC2 = 0);
        TMP2=cats(TMP1,".",PVLID);
        TMP3="";if not missing(PVLCMNT) then TMP3=%CCAT("MethodOID=@",qstrip("MT."||TMP1));
        TMP4="";if not missing(PVLCMNT) then TMP3=%CCAT("MethodOID=@",qstrip("MT."||TMP2));

        %XOUT(T2, "<ItemRef ItemOID=@ Mandatory=@ @>", qstrip("IT."||strip(TMP1)) qstrip(PVLREQD) TMP3 );
        %XOUT(T3, "<def:WhereClauseRef WhereClauseOID=@ @/>", qstrip("WC."||strip(TMP2)) TMP4);
        %XOUT(T2, "</ItemRef>");

        RC2 = H2.FIND_NEXT();
      END;

      %XOUT(T1, "</def:ValueListDef>");
      RC1=ITER1.NEXT();
   END;
RUN;
```

Generating ItemGroupDef Elements

An *ItemGroupDef* element is used to describe the metadata about dataset, It contains a set of attributes and child elements that refer to *ItemDef* elements and the links to .XPT files. In order to create ItemGroupDef elements, we use two hash objects to access the model datasets: DOMAIN and DOMVARS. Below is the complete code.

```
DATA ItemGroupDef(keep=LINE);
    LENGTH LINE $8192 KeySequence 8;
    LENGTH TMP1 $1024;

    %DefineHash(H1:ITER1,MODEL.DOMAIN[DOMNAME], multidata:'n');
    %DefineHash(H2      ,MODEL.DOMVARS[DOMNAME],multidata:'y');

    RC1=ITER1.FIRST();
    DO WHILE (RC1=0);

        %XOUT(T1, "<ItemGroupDef OID=@ Name=@ SASDatasetName=@ Repeating=@ IsReferenceData=@
        Purpose=""Analysis""
              def:Structure=@ def:Class=@ def:CommentOID=@ def:ArchiveLocationID=@>",
```

```
            qstrip("IG."||DOMNAME) qstrip(DOMNAME) qstrip(DOMNAME) qstrip(DOMREP) qstrip(DOMISREF)
            qstrip(DOMSTRUCT) qstrip(DOMCLASS) qstrip("COM."||DOMNAME) qstrip("LF."||DOMNAME));

        %XOUT(T2, "<Description>");
        %XOUT(T3, "<TranslatedText xml:lang=""en"">@</TranslatedText>", escs(DOMDESC));
        %XOUT(T2, "</Description>");

        RC2= H2.FIND();
        DO WHILE(RC2 = 0);
            KeySequence=getKeySequence(DOMKeys,VARNAME);
            TMP1="";
            if KeySequence > 0 then TMP1=%CCAT("KeySequence=@", qstrip(KeySequence));
            %XOUT(T2, "<ItemRef ItemOID=@ OrderNumber=@ Mandatory=@ @/>",
                  qstrip(catx(".","IT",DOMNAME,VARNAME)) qstrip(VARORDER) qstrip(VARREQD) TMP1);
            RC2 = H2.FIND_NEXT();
        END;

        %XOUT(T2, "<def:leaf ID=@ xlink:href=@>", qstrip("LF."||DOMNAME)
qstrip(cats(lowcase(DOMNAME),".xpt")));
        %XOUT(T3, "<def:title>@</def:title>", cats(lowcase(DOMNAME),".xpt"));
        %XOUT(T2, "</def:leaf>");
        %XOUT(T1, "</ItemGroupDef>");

        RC1=ITER1.NEXT();
   END;
Run;
```

Generating ItemDef Element

An *ItemDef* element is used to represent variable metadata. Variables are associated with datasets through *ItemRefs* contained in *ItemGroupDef* elements. Additional variable metadata that are associated with *ItemDefs* and *ItemRefs* are provided with following elements:

- *Codelist* for controlled terminology
- *def:ValueListDef* for value level metadata
- *MethodDef* for computational method
- *def:CommentDef* for comments
- *def:Origin* for variable origin

A variable can have both a codelist and a valuelist associated since they provide different semantic information for the variable. A codelist provides a list of allowable values that a variable can accept while a valuelist is used to define metadata based on the value of another variable in order to support data review and analysis when the variable metadata itself is not sufficient. In order to create *ItemDef* elements, we use hash objects to access the three model datasets: DOMAIN, DOMVARS and PVL. Below is the complete code.

```
DATA DEFLIB.ItemDef(keep=LINE);
    LENGTH LINE $8192;
    LENGTH TMP1 TMP2 TMP3 $1024;

    %DefineHash(H1:ITER1,DefLib.DOMAIN[DOMNAME], multidata:'n');
    %DefineHash(H2      ,DefLib.DOMVARS[DOMNAME],multidata:'y');
    %DefineHash(H3      ,DefLib.PVL[DOMNAME VARNAME/PVLID],multidata:'n');

    RC1=ITER1.FIRST();
    DO WHILE (RC1=0);

        RC2= H2.FIND();
        DO WHILE(RC2 = 0);
          TMP1="";
          if compare(varorigin,"Assigned", "IL:")=0 then TMP1=%CCAT(" def:CommentOID=@",
                                                    qstrip(catx(".","COM",DOMNAME,VARNAME)));
          TMP2="";
          if not missing(VARFMT) then TMP2=%CCAT(" def:DisplayFormat=@", qstrip(VARFMT));
           TMP3="";
          if lowcase(VARTYPE)="float" then do;
              TMP3=%CCAT(" SignificantDigits=@", qstrip(substrn(VARFMT,findc(VARFMT,'.')+1)));
          end;

          %XOUT(T1, "<ItemDef OID=@ Name=@ SASFieldName=@ DataType=@ Length=@ @ @ @>",
          qstrip(catx(".","IT",DOMNAME,VARNAME)) qstrip(VARNAME) qstrip(VARNAME)
```

```
                  qstrip(VARTYPE) qstrip(VARLEN)
                TMP1 TMP2 TMP3);

        %XOUT(T2, "<Description>");
        %XOUT(T3, "<TranslatedText xml:lang=""en"">@</TranslatedText>", escs(varlabel));
        %XOUT(T2, "</Description>");

        if not missing(VARCT) then do;
            %XOUT(T2, "<CodeListRef CodeListOID=@/>", qstrip(catx('.',"CL",VARCT)));
        end;

        if compare(varorigin,"Predecessor", "IL:")=0 then do;
          %XOUT(T2, "<def:Origin Type=""Predecessor"">");
          %XOUT(T3, "<Description>");
          %XOUT(T4, "<TranslatedText xml:lang=""en"">@</TranslatedText>", escs(varcmnt));
          %XOUT(T3, "</Description>");
          %XOUT(T2, "</def:Origin>");
        end;else  do;
          if H3.check() ne 0 then do;
            %XOUT(T2, "<def:Origin Type=@/>",qstrip(propcase(varorigin)));
          end;
        end;

        if H3.check()=0  then do;
            %XOUT(T2, "<def:ValueListRef ValueListOID=@/>", qstrip(catx(".","VL",DOMNAME,VARNAME)));
        end;

        %XOUT(T1, "</ItemDef>");

        RC2 = H2.FIND_NEXT();
      END;
      RC1=ITER1.NEXT();
    END;
Run;
```

Generating CodeList Element

A CodeList element is required for each controlled terminology referenced by variables and value lists in a study. It provides the definition of the controlled terminology, either an internal or external codelist.

The internal codelist includes a list of allowable codes and, if applicable, its corresponding decodes. If the controlled terminology is only a list of allowed values, a subelement, called *EnumeratedItem* is used to define the list of values. If the coded value has corresponding decodes, then a different sub-element, called *CodeListItem*, can be used. A codelist element may contain one or more *Alias* sub-elements in order to facilitate the identification of the codelist items in an external system. In the DEFINE-XML specification v2.0, the *Alias* sub-element is used to identify codelists and coded terms within the National Cancer Institue's Enterprise Vocabulary System.

The codelist element and its subelements can have optional or conditional attributes. For example, if the entries in a codelist have a numeric significance, *Rank* attributes can be optionally used within *EnumeratedItem or CodeListItem* sub-elements; If the coded value in a codelist is an extended value, then, the *def:ExtendedValue* attribute has to be set to "Yes". Below is the data step code that creates the codelist elements using CT dataset:

```
DATA DEFLIB.CodeList(keep=LINE);
    LENGTH LINE $8192 ORDER 8;
    LENGTH TMP1 TMP2 TMP_CTNAME $1024;

    %DefineHash(H1:ITER1,DefLib.CT[CTID], multidata:'n');
    %DefineHash(H2      ,DefLib.CT[CTID], multidata:'y');
     %DefineHash(H3      ,DefLib.sdtm_nci_terms[TERM_NAME CDISC_Submission_Value/TERM_CODE]
                        ,   multidata:'y');
    %DefineHash(H4      ,DefLib.sdtm_nci_terms[TERM_NAME/TERM_GROUPCODE], multidata:'n');

    RC1=ITER1.FIRST();
    Do while (rc1=0);

    tmp1="";
    if lengthn(CTSASFMT) then tmp1=%CCAT("SASFormatName=@", qstrip(CTSASFMT));
    if missing(CTNAME) then do; CTNAME=CTID; TMP_CTNAME=CTID; end; else
          TMP_CTNAME=%CCAT("@ (@)", CTNAME CTID);
```

```
    %XOUT(T1, "<CodeList OID=@ Name=@ DataType=@ @>", qstrip("CL."||CTID)
              qstrip(TMP_CTNAME) qstrip(CTTYPE) tmp1);
      RC2= H2.FIND(); ORDER=0;
      do while(rc2 = 0);
      order=order+1;

      tmp2="";
      if lengthn(CTEXTVAL) then tmp2=%CCAT("def:ExtendedValue=@", qstrip(CTEXTVAL));

      if not missing(CTDECODE) then do;
        if  not missing(CTRANK) then do;
          %XOUT(T2, "<CodeListItem CodedValue=@ Rank=@ @>", qstrip(escs(CTCODE)) qstrip(CTRANK) tmp2);
        end; else do;
          %XOUT(T2, "<CodeListItem CodedValue=@ OrderNumber=@ @>",
                  qstrip(escs(CTCODE)) qstrip(order) tmp2);
        end;
          %XOUT(T3,"<Decode>")
          %XOUT(T4,"<TranslatedText xml:lang=""en"">@</TranslatedText>", escs(CTDECODE))

        TERM_CODE="";TERM_NAME=CTNAME;CDISC_Submission_Value=CTDECODE;rc=h3.find();
        if not missing(TERM_CODE) then do;
           %XOUT(T5, "<Alias Name=@ Context=""nci:ExtCodeID""/>",qstrip(TERM_CODE));
        end;

          %XOUT(T3,"</Decode>")

        TERM_CODE="";TERM_NAME=CTNAME;CDISC_Submission_Value=CTCODE;rc=h3.find();
        if not missing(TERM_CODE) then do;
           %XOUT(T3, "<Alias Name=@ Context=""nci:ExtCodeID""/>",qstrip(TERM_CODE));
        end;

          %XOUT(T2,"</CodeListItem>")
      end; else do;

        TERM_CODE="";TERM_NAME=CTNAME;CDISC_Submission_Value=CTCODE;rc=h3.find();

        if missing(TERM_CODE) then do;
          if  not missing(CTRANK) then do;
            %XOUT(T2, "<EnumeratedItem CodedValue=@ Rank=@ @/>", qstrip(escs(CTCODE)) qstrip(CTRANK) tmp2);
          end;else do;
            %XOUT(T2, "<EnumeratedItem CodedValue=@ OrderNumber=@ @/>",
                    qstrip(escs(CTCODE)) qstrip(order) tmp2);
          end;
        end; else do;
          if  not missing(CTRANK) then do;
            %XOUT(T2, "<EnumeratedItem CodedValue=@ Rank=@ @/>", qstrip(escs(CTCODE)) qstrip(CTRANK) tmp2);
          end;else do;
            %XOUT(T2, "<EnumeratedItem CodedValue=@ OrderNumber=@ @/>",
                        qstrip(escs(CTCODE)) qstrip(order) tmp2);
          end;
          %XOUT(T3, "<Alias Name=@ Context=""nci:ExtCodeID""/>",qstrip(TERM_CODE));
          %XOUT(T2, "</EnumeratedItem>");
        end;
      end;
      rc2 = h2.find_next();
    end;

    TERM_GROUPCODE="";TERM_NAME=CTNAME;rc=h4.find();
    if not missing(TERM_GROUPCODE) then do;
       %XOUT(T2, "<Alias Name=@ Context=""nci:ExtCodeID""/>",qstrip(TERM_GROUPCODE));
    end;

    %XOUT(T1,"</CodeList>")
    %XOUT()

    RC1=ITER1.NEXT();
   end;
Run;
```

Generating MethodDef And def:CommentDef Elements

A *MethodDef* element can be part of metadata about variables and values in a valuelist. It is must be provided if any variables or values are defined as Derived. Each MethodDef element must contain a child Description element. In addition, it can have one or multiple def:DocumentRef element that point to external files. The tricky part of implementing a MethodDef element is how to know whether there are one or more child def:DocuemntRef elements for additional or external documents to be attached. One of the solutions is to ask users to make a tag-like phrase for

each of the document references at the end of each of method fields of the metadata sheets. The phrase is required to follow following pattern

<LF.*linkID*>

Where, the *linkID* is the leaf ID defined in the LINK dataset. The method variable is treated as a complex variable, and a subroutine called getDocRefs() is developed to extract all document references from the method fields in order to create the child *def:DecumnetREF* elements. Below is the sample implementation with the DOMVARS and PVL datasets.

```
DATA DEFLIB.MethodDef(keep=LINE);
    LENGTH LINE $8192;
    LENGTH TMP1 TMP2 $1024 num 8 trunc 8 idx 8;
    ARRAY  DOCREFS[32] $1024 _temporary_;

    %DefineHash(H1:ITER1,MODEL.DOMVARS[DOMNAME VARNAME], multidata:'n');
    %DefineHash(H2:ITER2,MODEL.PVL[DOMNAME VARNAME PVLID], multidata:'n');

    RC1=ITER1.FIRST();
    DO WHILE (RC1=0);
      IF compare(VAROrigin, "Derived", "IL:")=0 THEN DO;
        TMP1=Catx(".","MT",DOMName, VARNAME); TMP2=Catx(".","CM",DOMName, VARNAME);
        num=0;trunc=0;
        Call GetDocRefs(varcmnt, docrefs, num, trunc);

        %XOUT(T1, "<MethodDef OID=@ Name=@ Type=""Computation"">", qstrip(TMP1) qstrip(TMP2));
        %XOUT(T2, "<Description>");
        %XOUT(T3, "<TranslatedText xml:lang=""en"">@</TranslatedText>",escs(VARCMNT));
        %XOUT(T2, "</Description>");

        DO IDX=1 TO NUM;
          %XOUT(T2, "<def:DocumentRef leafID=@/>", qstrip(docrefs[idx]));
        END;

        %XOUT(T1, "</MethodDef>");
      END;

      RC1=ITER1.NEXT();
    END;

    RC2=ITER2.FIRST();
    DO WHILE (RC2=0);
      IF compare(PVLORIGIN, "Derived", "IL:")=0 THEN DO;
        TMP1=Catx(".","MT",DOMName, VARNAME, PVLID); TMP2=Catx(".","CM",DOMName, VARNAME, PVLID);

        num=0;trunc=0;
        Call GetDocRefs(PVLCMNT, docrefs, num, trunc);
        %XOUT(T1, "<MethodDef OID=@ Name=@ Type=""Computation"">", qstrip(TMP1) qstrip(TMP2));
        %XOUT(T2, "<Description>");
        %XOUT(T3, "<TranslatedText xml:lang=""en"">@</TranslatedText>",escs(PVLCMNT));
        %XOUT(T2, "</Description>");

        DO IDX=1 TO NUM;
          %XOUT(T2, "<def:DocumentRef leafID=@/>", qstrip(docrefs[idx]));
        END;

        %XOUT(T1, "</MethodDef>");
      END;

      RC2=ITER2.NEXT();
    END;
Run;
```

def:CommentDef is a new element introduced in Define-XML 2.0. It is intended to replace the deprecated ODM Comment attribute. Although it involves three datasets DOMAIN, DOMVARS, and PVL, since it can be created with the similar approach as MethodDef element, its implementation will not be further discussed.

Generating def:WhereClauseDef Element

A def:WhereClauseDef element is associated with value level metadata. It is used to describe the conditions to obtain a subset of the dataset rows that share similar metadata. Each value definition may have a Where Clause attached to it in order to describe when that value applies. Where Clauses define a condition with one or more Range Checks. This allows the construction of compound Where Clauses. The difficult part of implementing a def:WhereClauseDef element is how to create child RangeCheck elements. One of the solutions is to treat it as a complex variable and to ask users to annotate each of where clauses with statements like followings:

Rangechcek 1: Variable1 *Comparator_1* [*Val11 Val12* …]
Rangechcek 2: Variable2 *Comparator_2* [*Val21 Val22* …]
. . .

If the variable in the Range check comes from another dataset, then use following format

RangeCheck *n*: *Dataset.Variable* *Comparator_n* [*VALn1 VALn2* . . .]

Because the def:WhereClauseDef is treated as a complex variable, a subroutine called getRangeChecks() is developed to extract all range checks from the where clause paragraphs to create child RangeCheck elements. Below is the sample implementation with thePVL datasets.

```
DATA DEFLIB.WhereClauseDef(keep=LINE);
    LENGTH LINE $8292;
    LENGTH TMP1 TMP2 $1024 num 8 trunc 8 idx idx2 8 loc 8;
    LENGTH ITEMLST $1024 SOFTHARD $16 loc 8 ITEM $256  COMP_VAR $32 COMP_OP $16;
    ARRAY  Comparators[64] $1024 _temporary_;
    ARRAY  Items[64] $1024 _temporary_;

    %DefineHash(H1:ITER1,MODEL.PVL[DOMNAME VARNAME PVLID], multidata:'n');

    RC1=ITER1.FIRST();
    DO WHILE (RC1=0);

      num=0;trunc=0;
      Call GetRangeChecks(PVLWC, Comparators, Items, num,trunc);

      TMP1=catx(".", "WC", DOMNAME, VARNAME, PVLID);
      %XOUT(T1, "<def:WhereClauseDef OID=@>", qstrip(TMP1));
      DO idx=1 to num;
        loc=findc(Items[idx],"/");
        if loc > 1 then do;
          Softhard=propcase(substrn(Items[idx],loc+1));
          ITEMLST=substrn(Items[idx],1,loc-1);
        end;else do;
          SoftHard="Soft";
          ITEMLST=Items[idx];
        end;

        Comp_var=scan(Comparators[idx],1, " ");
        Comp_op=scan(Comparators[idx],2, " ");
        loc =index(comp_var,".");
        if loc =0 then TMP2=catx(".", "IT", DOMNAME, Comp_var);
        else TMP2=catx(".", "IT", Comp_var);

        %XOUT(T2, "<RangeCheck Comparator=@ SoftHard=@ def:ItemOID=@ >", qustrip(Comp_op)
         qstrip(softhard) qustrip(TMP2));
        idx2=1;
        ITEM=scan(ITEMLST,idx2," ,","q");
        Do While (lengthn(ITEM));
          %XOUT(T3, "<CheckValue>@</CheckValue>", strip(ITEM));
          idx2=idx2+1;
          ITEM=scan(ITEMLST,idx2," ,","q");
        END;
          %XOUT(T2, "</RangeCheck>");
      END;
      %XOUT(T1,"</def:WhereClauseDef>");

      RC1=ITER1.NEXT();
   END;
RUN;
```

Put All The Pieces Together

So far we have described how to create common define.xml v2.0 elements using CCAT utilities. With the similar approach, it is not difficult to create other elements such as def:AnnotatedCRF (for SDTM) and elements for Analysis Results Meatdata. Once all the elements for a define.xml file have been created, your define.xml generator can call %ToDefXML() to put all the pieces together to create a final define.xml file. Here is the example code:

```
%TODEFXML(
  HDR ValueListDef WhereClauseDef ItemGroupDef ItemDef CodeList MethodDef CommentDef leaf
  ,OUTDIR=define-file-folder
  ,OUTFILE=Define.XML
)
```

CONCLUSION

This paper presents an OR data model as a repository for the define.xml metadata sheets and a lightweight framework for the implementation of define.xml generators through the hash objects and user-defined functions in SAS® 9. The approach and techniques proposed in this paper will not only help you overcome many challenges in the implementation of a CDISC define.xml generator, but also give you a new way to develop many other similar SAS applications.

REFERENCES

1. CDISC Define-XML Specification, V2.0, Mar., 2013 (http://www.cdisc.org/define-xml)
2. CDISC Specification for the Operational Data Model (ODM), V1.3, Dec. 2006 (http://www.cdisc.org)
3. Lex Jansen, "*Understanding the define.xml File and Converting It to a Relational Database*" SAS Global Forum 2010.
4. CDISC SDTM/ADaM Pilot Project – Project Report, Jan., 2008 (http://www.cdisc.org)
5. Paul Brown, "*Object-Relational Database Development: A Plumber's Guide*", Prentice Hall, 2001.
6. Yves Deguire & Xiyun wang, "*Using SAS PROC FCMP in SAS System Development – Real Examples*", SAS Global Forum 2013
7. Paul M. Dorfman & Koen Vyverman, "*Data Step Hash Objects as Programming Tools*", SUGI 30, 2005.

CONTACT INFORMATION

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

Lei Zhang
Celgene Corporation
Summit, NJ, USA
E-mail: lezhang@celegene.com

Appendix: Main User-Defined Functions In the CCAT Framework

```
LIBNAME FCMPLib ".";
options cmplib=(FCMPLIB.utcdisc) linesize=200;

PROC FCMP outlib=FCMPLIB.utcdisc.string  /* listall flow */;

  /* ESCS() - Replace the XML special characters.*/
  FUNCTION ESCS(
      s  $     /*  A text variable or expression          */
  ) $32766;

  Return(TRANWRD(TRANWRD(TRANWRD(TRANWRD(TRANWRD(TRANWRD(s, '&', '&amp;'),"<", '&lt;'),">", '&gt;'),"'",
'&apos;'),'"', '&quot;'),'0a'x, '&#xA;'));
  ENDSUB;

 /* QTRIMN() - Trim spaces on the left side of a text expression and quote the text. */

  FUNCTION QTRIMN(
      s  $     /*  A text variable or expression          */
  ) $32766;
    Return(IFC(lengthn(s)>0,quote(trimn(s)),""" """));
    ENDSUB;

  /* QSTRIP() -  Strip and quote a text. If blank, return " ". */
  FUNCTION QSTRIP(
      s  $     /*  A text variable or expression          */
  ) $32766;
    Return(IFC(lengthn(s)>0,quote(strip(s)),""" """));
    ENDSUB;

  /* CCAT() -  Create CAT function call code from a template */
  FUNCTION CCAT(
    TEMPLATE $   /*  The code template for the CAT function call using @ a meta chacater */
    ,VAREXPS $     /*   A list of text variables or expressions                          */
  ) $32766;

  Length catexp $4096 Meta $1 citem $1024 exp1 $1024;
  catexp='';
```

```
if lengthn(TEMPLATE) = 0 then do;
  return('""');
end;

if lengthn(VAREXPS) = 0 then do;
  return(quote(TEMPLATE));
end;

meta='@';
maxlen=lengthn(TEMPLATE);
metacnt=0;
start=0;

DO WHILE(START<MAXLEN);

  loc=findc(TEMPLATE,meta, start+1);
  citem='';

  if loc >= start+1 then do;
    metacnt=metacnt+1;
    exp1=strip(scan(VAREXPS,metacnt,' ','Q'));
    if lengthn(exp1) > 0 then do;
      exp1=cats('trimn(',exp1,')');

      if loc > start + 1 then do;
        citem=quote(substrn(TEMPLATE,start+1, loc-start-1));

        exp1=cats(',',exp1);
        if metacnt=1 then do;
          catexp=cats(citem,exp1);
        end;else do;
          catexp=cats(catexp,',',citem,exp1);
        end;
      end; else do;
        if metacnt=1 then do;
          catexp=cats(exp1);
        end;else do;
          catexp=cats(catexp,',',exp1);
        end;
      end;
      start=loc;
    end; else do;
      citem=quote(substrn(TEMPLATE,start+1,maxlen-start));
      if metacnt > 0 then do;
        catexp=cats(catexp,',',citem);
      end; else do;
        cateexp=citem;
      end;
      start=maxlen;
     end;
  end; else do;
    citem=quote(substrn(TEMPLATE,start+1,maxlen-start));
    if metacnt > 0 then do;
      catexp=cats(catexp,',',citem);
    end; else do;
      cateexp=citem;
    end;
    start=maxlen;
  end;
END;
  return(cats('CAT(',catexp,')'));
ENDSUB;

/* GetRangeCheck - Get RangeCheck components from a WhereClause paragraphs */
SUBROUTINE GetRangeChecks(
   DOC $
  ,Comparator[*] $
  ,Entry[*] $
  ,n
  ,trunc
);
  outargs Comparator, Entry, n, trunc;

  Length RegexID 8 Text $1024 found $32  count 8 maxrows 8 loc1 8 loc2 8;
  Length found $1024;

  maxrows=dim(entry);
    RegexID = prxparse('/RangeCheck\s*\d*\s*:/im');
    start = 1;
    stop = lengthn(DOC);
  prev_loc=0; count=0;n=count; trunc=0;
```

```
  if stop=0 then return;
    call prxnext(RegexID, start, stop, DOC, position, length);
      do while (position > 0);
      if prev_loc > 0 then do;
        count=count+1;
      if count > maxrows then do; trunc=1; count=ccount-1;end;
      else do;
            found = strip(substrn(DOC, prev_loc+1, position - prev_loc -1));
        loc1=findc(found,'['); loc2=findc(found,']', -lengthn(found));
        if loc1 ne 0 and loc2 ne 0 then do;
          Comparator[count]=substrn(found,1,loc1-1);
          Entry[count]=substrn(found, loc1+1, loc2-loc1-1);
        end; else do;
          Comparator[count]="";
          Entry[count]=found;
        end;
      end;
    end;

    prev_loc=position+length-1;
        call prxnext(RegexID, start, stop, DOC, position, length);
      end;

  found='';
  if (prev_loc > 0) then do;
      found = strip(substrn(DOC, prev_loc+1, stop-prev_loc));
  end; else if count=0 then do;
    found=doc;
  end;

  if lengthn(found) > 0 then do;
    count=count+1;
    if count > maxrows then do; trunc=1; count=count-1; end;
    else do;
      loc1=findc(found,'['); loc2=findc(found,']', -lengthn(found));
      if loc1 ne 0 and loc2 ne 0 then do;
        Comparator[count]=substrn(found,1,loc1-1);
        Entry[count]=substrn(found, loc1+1, loc2-loc1-1);
      end; else do;
        Comparator[count]="";
        Entry[count]=found;
      end;
    end;
  end;
  n=count;
ENDSUB;

/* getDocRefs - get the documemt link IDs from a comment or method complex variable */
SUBROUTINE GetDocRefs(
   DOC $
  ,LeafRef[*] $
  ,n
  ,trunc
);

  outargs LeafRef, n, trunc;

  Length RegexID 8 Text $1024 found $1024  count 8 maxrows 8 loc1 8 loc2 8;

  maxrows=dim(LeafRef);
    RegexID = prxparse('/\<\s*LF\..+?\>/im');
    start = 1;
    stop = lengthn(DOC);
  prev_loc=0; count=0;n=count;trunc=0;
  if stop=0 then return;

    call prxnext(RegexID, start, stop, DOC, position, length);
      do while (position > 0);
      count=count+1;
    if count > maxrows then do; trunc=1; count=count-1; end;
    else do;
          found = substr(DOC, position+1, length-2);
      LeafRef[count]=found;
          call prxnext(RegexID, start, stop, DOC, position, length);
    end;
  end;

  n=count;
ENDSUB;

/* GetDSNVARS - Get variable name list and definitions */
```

```
  SUBROUTINE GetDSNVARS(
     DSNEXP   $    /* dataset name with a list of variables. Format: DSN[x1 x2 x3 . . .]
                      if  no variable list, returns all the variables of the dataset     */
    ,outvars  $    /* a list of variable names extracted. Format: x1 x2 x3               */
    ,outvardefs $  /* a list of variable defnitions created.
                   Format:
                   x1 8  X2 $20 x3 $200                                                  */
  );
    outargs outvars, outvardefs;
    Length DSNEXP $128 OUTVARS $1024 OUTVARDEFS $1024;
    Length dsid 8 dsname $32 dsvars $1024 loc 8 VName $32;
    Length Vlist  $4096 NVARS 8 idx 8 VTYPE $8 VLENGTH 8;

    loc =findc(DSNEXP,'[');
    if loc > 0 then do;
      dsname=substrn(DSNEXP,1,loc-1);
      dsvars=substrn(DSNEXP,loc+1, lengthn(DSNEXP)-loc-1);
      dsvars=upcase(dsvars);
    end; else do;
      dsname=dsnexp;
      dsvars="";
    end;
    dsid= OPEN(DSNAME,"is");
    if dsid = 0 then do;
          rc= CLOSE(dsid);
          put "ERROR: Can't open the dataset provided.";
          return;
    end;

    NVARS=ATTRN(dsid, "NVARS");
    OUTVARDEFS="";
    OUTVARS="";
    Do idx=1 to NVARS;
        VNAME=upcase(VARNAME(DSID, IDX));
        VTYPE=VARTYPE(DSID, IDX);
        VLENGTH=VARLen(DSID,IDX);
        loc=findw(strip(DSVARS),strip(VNAME)," ,[]");
        if lengthn(dsvars)=0 OR  loc > 0 then do;
            if VTYPE="N" then do;
              OUTVARDEFS=catx(" ", OUTVARDEFS, VNAME, VLENGTH);
              OUTVARS=catx(" ", OUTVARS, VNAME);
            end; else do;
              OUTVARDEFS=catx(" ", OUTVARDEFS, VNAME, cats("$",VLENGTH));
              OUTVARS=catx(" ", OUTVARS, VNAME);
            end;
        end;
    End;
    rc= CLOSE(dsid);
    return;
  ENDSUB;

  /* GetKeySequence - Get the sequence number from a ordered list of key variables */
  FUNCTION GetKeySequence(
     Keylst   $    /* a listof ordered keys separated by space or comma or semicolon  */
    ,Varname $     /* a variable name                                                 */
  );
    LENGTH idx 8 item $32;

    idx=1;
    item=scan(Keylst,idx, " ,;");
    DO While (lengthn(item)>0);
      if compare(Varname, item, "IL")=0 then return(idx);
      idx=idx+1;
      item=scan(Keylst,idx, " ,;");
    END;
    return(0);
  ENDSUB;

  /* %DefineHash: Define a hash table and its optional hash iterator.*/
  Function DefineHash(
   HasHexp $     /*  Hassh[:Hash-iterator] (required)                    */
  ,DSNEXP  $     /*  DSN[key1 key2 . . . /var1 var2 . . .] (required)    */
  ,HashOpts $    /*   Hash options such as multidata:'yes'                */
  ) $8192;

    Length Hashnm $32 Iternn $32 DSNnm $32 VARlst $8192 Keylst $1024 Datalst $1024 Comblst $1024 combdeflst
$1024;
    Length VarDeflst $1024 HashDef $1024 IterDef $1024  KeyDef $1024 DataDef $1024 Itemdef $1024 missdef
$1024;
    Length Keylstx $1024 Datalstx $1024;
    Length loc 8 locl 8 idx 8;
```

```
    /* Get hash name or hash iteration name */
    loc=index(Hashexp,':');
    if loc > 1 then do;
      Hashnm=substrn(Hashexp,1,loc-1);
      Iternm=substrn(Hashexp,loc+1);
    end; else do;
      Hashnm=HashExp;
      Iternm='';
    end;
    /* get data name and the user-defined key and data variable lists */
    KeyLstx='';
    Datalstx='';
    loc=index(DSNExp,'[');
    if loc > 1 then do;
      DSNnm=substrn(DSNexp,1,loc-1);
      VARLST=substrn(DSNexp,loc+1, lengthn(DSNExp)-loc -1);
      loc1=index(VARlst,'/');
      if loc1 > 1 then do;
        Keylstx=substrn(VARlst,1,loc1-1);
        Datalstx=substrn(VARlst,loc1+1);
      end; else do;
        Keylstx=VARLst;
      end;
    end; else do;
      DSNnm=DSNExp;
    end;

    Keylst='';
    Datalst='';
    Comblst='';
    Combdeflst='';

    CALL getDSNVars(cats(DSNnm,'[',Keylstx,']'), keylst, combdeflst);
    CALL getDSNVars(cats(DSNnm,'[',Datalstx,']'),datalst, combdeflst);
    comblstx=cat(strip(Keylst),' ', strip(Datalst));
    CALL getDSNVars(cats(DSNnm,'[',comblstx,']'),comblst, combdeflst);

    VarDeflst=%ccat('Length @', combdeflst);

    if lengthn(HashOpts) then do;
      HashDef=%ccat('DECLARE HASH @(dataset:@,@)', Hashnm singlequote(DSNnm) HashOpts);
    end; else do;
      HashDef=%ccat('DECLARE HASH @(dataset:@)', Hashnm singlequote(DSNnm));
    end;

    IterDef='';
    if lengthn(ITERnm) then do;
      IterDef=%ccat('DECLARE HITER @(@)',ITERnm singlequote(Hashnm));
    end;

    itemdef=qvarlst(Keylst);
    Keydef=%ccat('rc=@.defineKey(@)',HashNM itemdef);

    itemdef=qvarlst(Datalst);
    Datadef=%CCAT('rc=@.definedata(@)',HashNM itemdef) ;

    CommaVarlst=translate(strip(Comblst),',',' ');
    MissDef=%CCAT('call missing(@)', CommaVarlst);

    comblst=catx(';'||byte(10),VarDeflst, HashDef, IterDef, KeyDef, DataDef,
%ccat('rc=@.DefineDone()',Hashnm),MissDef);
    return(cats(comblst,';'));
  ENDSUB;
RUN;QUIT;
```