PharmaSUG 2015 - Paper BB17

Fresh Cup of Joe: Utilizing Java to automate Define.XML for SDTM Origin mapping from SAS® aCRF PDFs.

Tony Cardozo, Theorem Clinical Research, King of Prussia, PA.

ABSTRACT

Define.XML specifications provide an extended Clinical Data Interchanges Standards Consortium (CDISC) Operational Data Model (ODM) to describe clinical data and statistical analyses submitted for FDA review. Part of the Define.XML process is identifying and mapping Study Data Tabulation Model (SDTM) variables to their SAS® annotated Case Report Form (aCRF) origin. This process can be labor intensive and time consuming. This paper first introduces the programming language Java as a powerful alternative to processing and manipulating PDF documents, and then demonstrates how it may be used to fully automate the SAS® aCRF to SDTM variable origin mapping process. In addition to explaining how to implement Java, this paper provides specific examples of how to use Apache PDFBox^{TM}, a free Java-PDF library, to extract all SAS® aCRF annotations while setting all XML accessible hyperlinks/destinations within the PDF document. Lastly, it covers how SAS® can utilize the extracted annotations to determine all one-to-one, one-to-many, and many-to-many SDTM variable to SAS® aCRF relationships. This allows for a fully programmatic solution to Define.XML variable origin mapping.

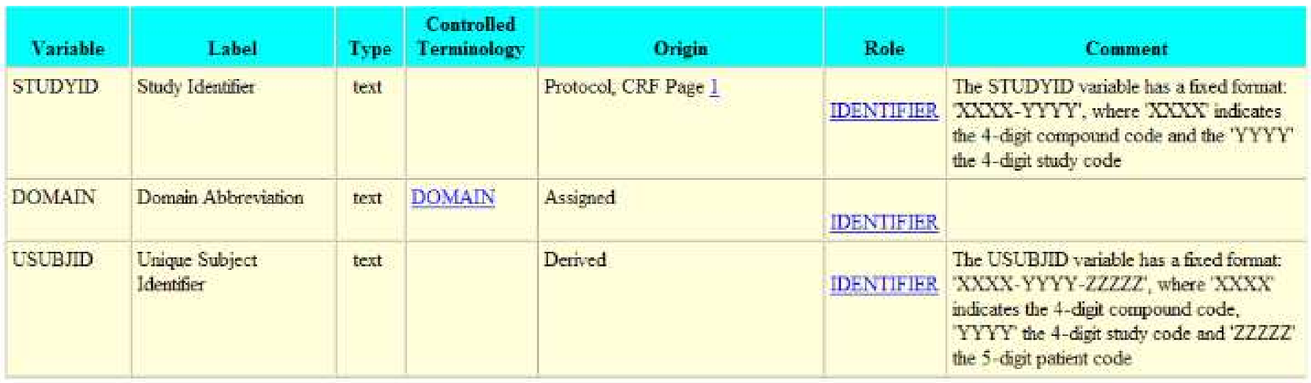
INTRODUCTION

Define.xml (Case Report Tabulation Data Definition Specification) is a document that the FDA requires for the electronic submission of a clinical trial. It is used to define the submitted datasets as well as the specific contents of each dataset of the trial. Within this metadata, the information to identify the specific data sources or the methodology used in data derivation is documented. The flow between the different components of the Define.xml is presented in a fluid manner using hyperlinks and bookmarks between the various files (datasets, annotated case report forms, reviewer's guide, supplemental data definitions file).

The define.xml describes metadata at 5 Levels:

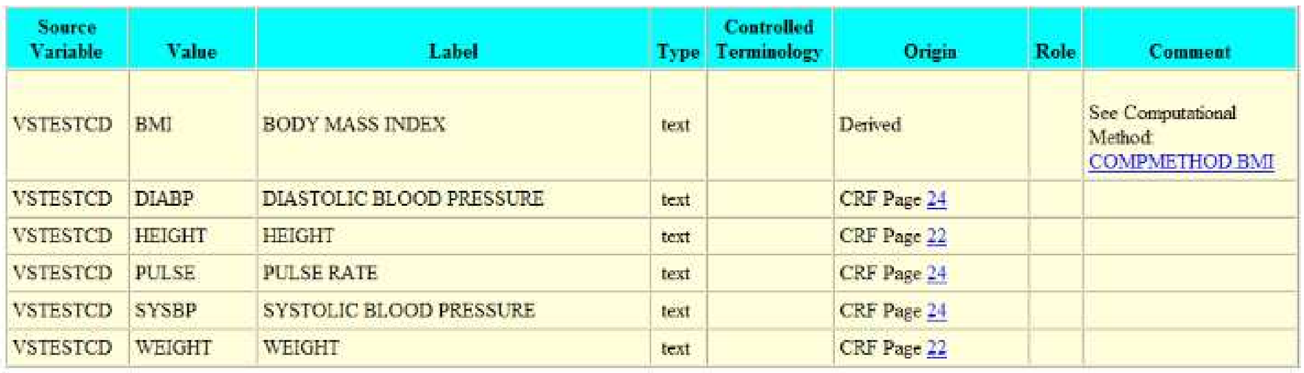
1. Dataset level metadata
2. Variable level metadata (Fig. 1)
3. Value level metadata (Fig. 2)
4. Controlled Terminology metadata
5. Computational Algorithm metadata

The define.xml can contain metadata for a wide range of standards including CDISC SDTM, CDISC Analysis Dataset Model (ADaM), and Standard for Exchange of Non-clinical Data (SEND). However, this paper will focus on the variable and value level origins metadata of the CDISC SDTM standard. Additionally, this paper with not cover the overall Define.xml creation process but focus on the automation of the time consuming process of variable Origin mapping from the aCRF.



Variable	Label	Type	Controlled Terminology	Origin	Role	Comment
STUDYID	Study Identifier	text		Protocol, CRF Page 1	IDENTIFIER	The STUDYID variable has a fixed format: 'XXXX-YYYY', where 'XXXX' indicates the 4-digit compound code and the 'YYYY' the 4-digit study code
DOMAIN	Domain Abbreviation	text	DOMAIN	Assigned	IDENTIFIER	
USUBJID	Unique Subject Identifier	text		Derived	IDENTIFIER	The USUBJID variable has a fixed format: 'XXXX-YYYY-ZZZZZ', where 'XXXX' indicates the 4-digit compound code, 'YYYY' the 4-digit study code and 'ZZZZZ' the 5-digit patient code

FIGURE 1. EXAMPLE OF VARIABLE LEVEL METADATA[1]



Source Variable	Value	Label	Type	Controlled Terminology	Origin	Role	Comment
VSTESTCD	BMI	BODY MASS INDEX	text		Derived		See Computational Method: COMPMETHOD BMI
VSTESTCD	DIABP	DIASTOLIC BLOOD PRESSURE	text		CRF Page 24		
VSTESTCD	HEIGHT	HEIGHT	text		CRF Page 22		
VSTESTCD	PULSE	PULSE RATE	text		CRF Page 24		
VSTESTCD	SYSBP	SYSTOLIC BLOOD PRESSURE	text		CRF Page 24		
VSTESTCD	WEIGHT	WEIGHT	text		CRF Page 22		

FIGURE 2. EXAMPLE OF VALUE LEVEL METADATA[1]

Currently, the most common way to map variable origin metadata is by hand. Someone physically matches annotations from the aCRF with a variable list and determines the mapping. Depending on the study and CRF design, this can be an extremely time consuming and tedious process. There are some processes where annotations can be exported from the aCRF PDF into an XML Forms Data Formatted (XFDF) file and parsed using Perl Regular Expressions. However, these processes can be complex and are prone to data loss [2].

In this paper we'll discuss an alternative method of annotation extraction using Java, Apache PDFBox™ and SAS®. Additionally, the proposed process will efficiently map the aCRF origin metadata to the corresponding domain variables.

SAS ANNOTATED CRF PDFS

CDISC has established regulatory requirements to ensure that variable and value level annotations found in the aCRF are compliant both in form and syntax [3]. Having a standard approach to the annotation of the aCRF is essential, especially if a programmatic solution is desired. This paper assumes you have a general understanding of the annotation process and how to actually annotate an aCRF PDF.

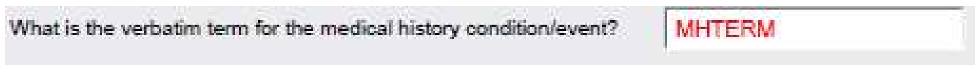
For this paper we will cover the standard aCRF annotations:

1. Single Variable Annotations
2. Supplemental Qualifier Annotations
3. Conditional Annotations

For this paper, the aCRF annotation process has been simplified so we may focus on the programming. Annotations may become ambiguous between domains. Although the programmatic process outlined in this paper can handle complex annotation logic, we will focus on simple examples to convey feasibility.

SINGLE VARIABLE ANNOTATIONS

Single variable annotations are the simplest and depending on the study, the most widely used annotation. They are used to annotate the direct mapping of an aCRF field to its CDISC SDTM variable.

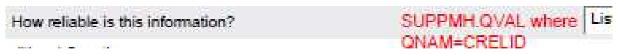


What is the verbatim term for the medical history condition/event?

MHTERM

SUPPLEMENTAL QUALIFIER ANNOTATIONS

Supplemental Qualifier Annotations define relationships between the supplemental and parent domains.



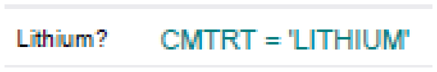
How reliable is this information?

SUPPMH.QVAL where QNAM=CRELID

Lis

CONDITIONAL ANNOTATIONS

Conditional Annotations are very common in aCRFs and can be annotated in a standard format or shorthand. We'll be using the shorthand approach.



ANNOTATION EXTRACTION

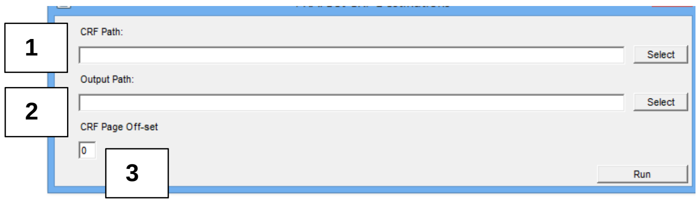
Once there is an aCRF we may begin the process of variable origin mapping. As mentioned above, aCRF annotations can be extracted into an XFDF file or text file. Both options require having the aCRF open and software that can extract the annotations. The Java approach we are about to layout does not require either. One very important note about a requirement shared by any approach, the aCRF has to have the annotations in text form. It cannot be a scanned in version of the aCRF since that will produce an image and not text.

JAVA CRASH COURSE

Java has been around since the early 90s and is still heavily utilized today. Although Java is best known as a programming language to design internet applications, it can be used as a general programming language independent of the internet. Java is object orientated, platform independent, and class based which makes it powerful and scalable [4]. One big advantage Java has over some other programming languages is that you can find free programming environments that are fully functional and deployable. There are vast amounts of information about Java programming online. Although programming environment setup is not within the scope of this paper, it should be noted that all Java programming was done with a free Java Complier called Eclipse (www.eclipse.org). The Eclipse website does have all the necessary documentation to set up a Java programming environment on any platform. In addition to Eclipse, we'll need Apache PDFBox^{TM}. The Apache PDFBox™ library is an open source Java tool for working with PDF documents.

DEFINE.XML ORIGIN MAPPING PROCESS

This paper assumes we need a basic Graphical User Interface (GUI) to assist the user processing their aCRF. GUI applet development will not be covered in this paper, but a simple applet as seen in Figure 4 below can handle the basic functionality we need.



DISPLAY 1: USER INTERFACE

1. User defined path to aCRF
2. User defined output path and desired filename;
3. CRF Off-set to cover aCRFs that have an off-set between PDF page 1 and CRF page 1.

Once the user enters in the aCRF path, desired output path, any page off-sets and hits run a couple different programmed functions are called.

- Pullannotations()
- SetDestinations()

EXTRACTING ANNOTATIONS

The Pullannotations() function is used to extract the annotations from the aCRF.

```
private void Pullannotations() throws IOException          1
{
        try {
                PDDocument pddDocument = PDDocument.load(new File(this.folderPath));   2
                List allPages = pddDocument.getDocumentCatalog().getAllPages();

                String path = this.folderPath.replace(".pdf", "_comments.txt");
                PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(path, true)));   3

                for (int i = 0; i < allPages.size(); i++)
                {
                   4    int pageNum = i + 1;
                        PDPage page = (PDPage) allPages.get(i);

                   5    List<PDAnnotation> la = page.getAnnotations();

                        if (la.size() < 1)
                        {
                                continue;
                        }

                   6    out.println("Page: " + pageNum);

                   7    for (int j = 0; j < la.size(); j++)
                        {
                                PDAnnotation pdfAnnot = la.get(j);

                                System.out.println("Base " + pdfAnnot.getContents());
                                String myString = pdfAnnot.getContents();

                                if (myString != null && !myString.equals(""))
                                {
                                        String myString2 = myString.replaceAll("\\s+", " ");
                                        out.println(myString2);
                                }
                        }
                }

          8     out.close();
                pddDocument.close();

          9     } catch (Exception ex) {
                        ex.printStackTrace();
                        setErrorText("Error: Cannot process PDF");
                }

}/*End doAction*/
```

1. Private void Pullannotations() is the function called on by the user interface to make the text file with all annotations.
2. Using PDDocument.load and pddDocument.getDocumentCatalog().getAllPages() classes from the PDBox library we can load the aCRF into memory and establish the total number of pages for looping purposes.
3. We take the path entered into the user interface and prepare Java to output a text file with the same path and filename as entered by the user with _comments added.
4. Loop through all the pages keeping track of the current page.

5. Put all the annotations on the current page in a list container. You can think of the list container as a vertical array of text strings.
6. Output a line to the text file that has “Page:” and the current page number. This line can later be retained during SAS processing to keep all the annotations on a page together.
7. Loop through all the annotations listed in the list container and output them to the text file.
8. Java opens elements into memory and they need to be closed.
9. Check for PDF processing errors and let the user know if any were found.

Once the PullAnnotations() function completes a text file is outputted to the specified folder from the GUI form.

```
Page: 19
MHTERM
MHSTDTC
If [Yes] then MHENRF='DURING/AFTER'
MHENDTC
DOMAIN = MH
MHCAT = 'GENERAL MEDICAL HISTORY'
Page: 21
DOMAIN = CM
CMCAT = 'ADHD MEDICATION HISTORY'
```

OUTPUT 1. OUTPUT FROM PULLANNOTATIONS()

SETTING XML HYPERLINKS/DESTINATIONS

```
private void SetDestinations() throws IOException
    {
   PDDocument document = null;
   try
   {
1     document = PDDocument.load(new File(this.folderPath));
      if( document.isEncrypted() )
      {
         System.err.println( "Error: Cannot add destinations to encrypted document." );
         System.exit( 1 );
      }
      PDDocumentOutline outline =  new PDDocumentOutline();
2     document.getDocumentCatalog().setDocumentOutline( outline );
      PDOutlineItem pagesOutline = new PDOutlineItem();

      outline.appendChild( pagesOutline );
      @SuppressWarnings("rawtypes")
3     List pages = document.getDocumentCatalog().getAllPages();
      for( int i=0; i<pages.size(); i++ )
      {
         setStatusText("Processing Page: " + Integer.toString(i) + " of " + Integer.toString(pages.size()));
4        PDPage page = (PDPage)pages.get( i );
         PDPageFitWidthDestination dest = new PDPageFitWidthDestination();
         dest.setPage( page );
5        PDOutlineItem bookmark = new PDOutlineItem();
         bookmark.setDestination( dest );
         bookmark.setTitle(Integer.toString(i+1+ Integer.parseInt(this.outSetText.getText().toString())));
6        pagesOutline.appendChild( bookmark );
      }
      pagesOutline.openNode();
      outline.openNode();
7     String path = this.folderPath.replace(".pdf", "_temp.pdf");
```

```
          try {
                        document.save( path );
               }
           catch (COSVisitorException e)
              {
                       setErrorText("Error! Saving Document");
              }
8       }
        finally
        {
          if( document != null )
          {
            document.close();
          }
        }

   }
```

1. Follow the same process as PullAnnotations(), open the aCRF PDF document double checking that the document is not encrypted.
2. The getDocumentCatalog() class makes an in-memory representation of the PDF document and structures an outline to support the destinations using setDocumentOutline.
3. Make a list of all the pages for looping.
4. Get the current page, make a destination element and link it to the current page.
5. The bookmark.setDestination(dest) call actually sets the destination to the current page. Each page will have a destination pointing to the current page number.
6. Set the title to the destination. This is the title that represents the value seen in the Define.xml. E.g., you see an origin of “CRF Page 6” in the Define.xml doc, the title of the destination will be 6. The “i” represents the current page. Since the first page element is element 0 we add 1 to i to adjust.
   The + Integer.parseInt(this.outSetText.getText().toString())) accounts for any additional adjustments needed to the page number mapping defined by the user on the user interface.
7. Update aCRF filename so the aCRF with the destinations does not overwrite the previous version. An additional step could be added here removing all annotations and outputting the actual blankcrf.pdf.
8. Save the aCRF and close the open documents in memory.

The above Java applet was able to process a 143 page aCRF in 6 seconds. The applet was programmed as a standalone GUI application for the purpose of this paper. However, it could be programmed as a Java Archive file (JAR) and executed directly from SAS using command line pipe commands.

FORMATTING ANNOTATIONS

Now that we have the aCRF annotation in text form they can be processed in SAS and merged onto the corresponding SDTM variable list and outputted to the metadata document being used.

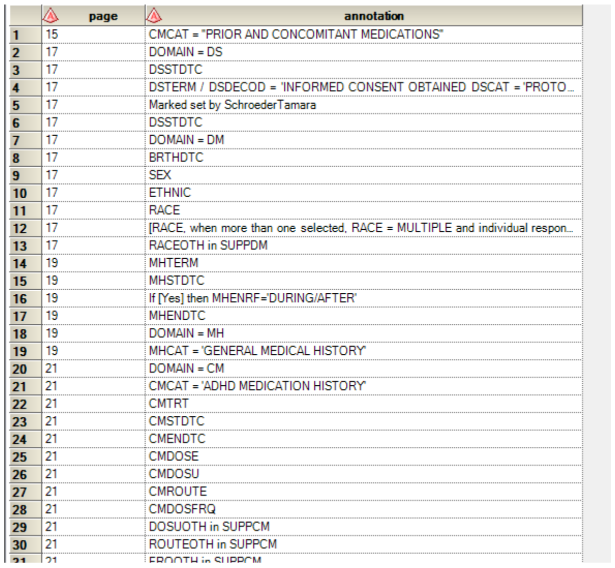
Some basic data cleaning will get the output in **Output 1** above to a useable dataset ready to merge with SDTM variables list.

```
data crf_annotations;
        length  page $4 annotation $2000;

        infile "/Path_to_your/Comments_text_file.txt" dsd truncover;
        input annotation $200.;
        retain page;
        if annotation = ' ' then delete;
1       if index(annotation,'Author') ^= 0 then delete;
        if index(annotation,'*') ^= 0 then delete;
        if index(annotation,'[Not Submitted]') ^= 0 then delete;

2
        if index(annotation,'Page: ') then
        do;
```

```
                page= strip(substr(annotation,7,3));
        end;

        if index(annotation,'Page: ') then delete;

run;
```

1. Basic cleaning of text file to remove unwanted records. Technically this does not have to be done since annotations will be mapped to matching SDTM variables. You can also add any study specific processing here if needed.
2. The "Page:" key word was added in step 6 of the Pullannotations() function above. Now we can use it retain the page number for all the annotations on the same page.

After the basic cleaning we get a workable dataset.



	page	annotation
1	15	CMCAT = "PRIOR AND CONCOMITANT MEDICATIONS"
2	17	DOMAIN = DS
3	17	DSSTDTC
4	17	DSTERM / DSDECOD = 'INFORMED CONSENT OBTAINED DSCAT = 'PROTO...
5	17	Marked set by SchroederTamara
6	17	DSSTDTC
7	17	DOMAIN = DM
8	17	BRTHDTC
9	17	SEX
10	17	ETHNIC
11	17	RACE
12	17	[RACE, when more than one selected, RACE = MULTIPLE and individual respon...
13	17	RACEOTH in SUPPDM
14	19	MHTERM
15	19	MHSTDTC
16	19	If [Yes] then MHENRF='DURING/AFTER'
17	19	MHENDTC
18	19	DOMAIN = MH
19	19	MHCAT = 'GENERAL MEDICAL HISTORY'
20	21	DOMAIN = CM
21	21	CMCAT = 'ADHD MEDICATION HISTORY'
22	21	CMTRT
23	21	CMSTDTC
24	21	CMENDTC
25	21	CMDOSE
26	21	CMDOSU
27	21	CMROUTE
28	21	CMDOSFRQ
29	21	DOSUOTH in SUPPCM
30	21	ROUTEOTH in SUPPCM

OUTPUT 2. aCRF ANNOTATIONS AND RELATING PAGE NUMBERS

IMPORTING ANNOTATIONS INTO SAS ENVIRONMENT AND MAPPING ORIGIN

In order to merge the aCRF page information to the appropriate variable we first need to have a list of all the variables in the study. Since this can be done several ways, this paper will not outline a specific method. Obtaining a variable list for the PROC CONTENTS of a SAS library or importing a document containing the variables names are both common practices. The end result is the same, a dataset containing all the variables that need aCRF origin mapping.

```
%macro map_annotations();
data _null_;
        set variable_source end=eof;
                                                          1
        varnm = 'var' || left(put(_n_,5.));
        call symput(varnm, upcase(variable_name));

        if eof then call symput('variable_total',trim(left(put(_n_,5.))));
run;

proc datasets lib=work nolist nowarn;
    delete combo;
  run;
```

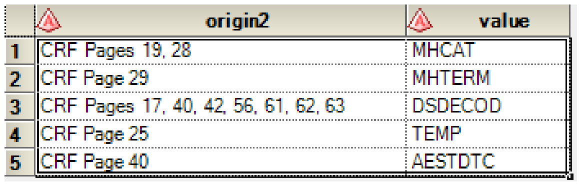
```
%do i=1 %to &variable_total;     2

           proc sort data=crf_annotations out=temp_crf nodupkey;
             by page;
             where index(strip(upcase(annotation)),strip("&&var&i"))^=0;     3
           run;

           data temp_crf;
             set temp_crf end=eof;
             length origin2 $200;
             retain origin2;
             retain firstin;

     4       if _n_ = 1 then firstin=1;
             variable_position = index(strip(upcase(annotation)),strip("&&var&i"));
             if variable_position > 1 then str = strip(compress((substr(strip(upcase(annotation)),
                                               (variable_position-  1),(length(strip("&&var&i"))+1))), "=", "P"));
             else str = strip(compress((substr(strip(upcase(annotation)),variable_position,(length(strip("&&var&i"))+1))), "=", "P"));
             value = "&&var&i";

             if str = strip("&&var&i") then
             do;
               if firstin =1 then
                 do;
                   origin2 = 'CRF Page '|| strip(page);
                   firstin = 0;
                 end;
     5         else if firstin = 0 then
                 do;
                   origin2 = tranwrd(origin2,"Page","Pages");
                   origin2 = strip(origin2) || ', ' || strip(page);
                   firstin = 2;
                 end;
               else if _n_ > 2 then origin2 = strip(origin2) || ', ' || strip(page);
             end;

             if not eof then delete;
                       keep origin2 value;
           run;

  6        proc append base=combo data=temp_crf force;
               run;
           %end;
%mend map_annotations;
```

1. Using the dataset containing all the variables/values needing aCRF mapping, make macro variables for each variable value and a total variable counter.
2. Loop through all the macro variables made in step one.
3. Pull all the aCRF annotations from the annotations dataset that has a fuzzy match. At this point we are only narrowing the annotations. E.g. a fuzzy match would determine “ATTEMP” and “TEMP” as a match.
4. Determine if the fuzzy match from step 3 is an actual match. By determining the starting position of the variable within the annotation, we can take the length of the variable and add a leading and trailing character. This would turn “ATTEMP” into “TTEMP” from the example above. We compress leading and trailing spaces and also known punctuation. If any special study-specific logic is needed, it could be added

here. Once the sub-string of the annotation is determined, it is checked against the variable currently being processed.

5. Using the FIRSTIN toggle, we determine the number of pages the current variable matches in the aCRF. Since Define.xml naming conventions are different for multiple aCRF annotations we make sure to use the correct syntax of “CRF Page:” or “CRF Pages:” followed by the matching page numbers.
6. Append the currently processed variable to the preceding loop iterations.

We now have a dataset containing the variables\values we wanted mapped to the aCRF with its actual mapping.



	origin2	value
1	CRF Pages 19, 28	MHCAT
2	CRF Page 29	MHTERM
3	CRF Pages 17, 40, 42, 56, 61, 62, 63	DSDECOD
4	CRF Page 25	TEMP
5	CRF Page 40	AESTDTC

OUTPUT 3. VARIABLE WITH RELATED ACRF MAPPING

The only remaining step is to merge the aCRF origin mapping data with the corresponding document containing the rest of the value level metadata. This can be done multiple ways, depending on the format of the documentation. PROC IMPORT and PROC EXPORT could be used to read in an Excel document, merge on the aCRF mapping data and export the document back out.

CONCLUSION

The Define.xml is a required document for electronic submissions of clinical trial data to the FDA. Its creation can be a tedious, time consuming process. There are many tools available to speed up certain parts of the process. This paper explained an option to fully automate the SAS® aCRF to SDTM variable origin mapping process using Java, Apache PDFBox^{TM} and SAS®. The outlined solution is cross-platform compliant and customizable for study specific requirements. Although the examples provided in this paper were oversimplified to show a general programming approach, more complex logic can be applied to support more dynamic annotaton.

REFERENCES

[1] Rohit Banga, Business & Decision Life Sciences, Generate Define.xml & Define.pdf from Metadata Environment, PhUSE 2009
[2] Joel Campbell, Ryan Wilkins, Importing and Parsing Comments From a PDF Document With Help From Perl Regular Expressions, PharmaSUG 2011
[3] Wilkins RB, Campbell JK, A Regular Language: The Annotated Case Report Form, PharmaSUG 2011

[4] http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html

CONTACT INFORMATION

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

Name:	Tony Cardozo
Enterprise	Theorem Clinical Research
Address:	1016 West Ninth Avenue
City, State ZIP:	King of Prussia, PA 19406
Work Phone:	(816) 600-8755
E-mail:	Antonio.Cardozo@TheoremClinical.com