**PharmaSUG Paper – AD-211**

# Validating Hyperlinks in SDTM define.xml Using Python

Brandon Welch, Greg Weller, Rho® Inc.

## ABSTRACT

As a one-stop location for a clinical trial's metadata, the define.xml file is a vital piece of an FDA submission. Held within this file are many hyperlinks. Some links are internally specific to the XML file, while others point to external locations. Of particular interest in SDTM submissions are the links that externally map to the study's annotated case report form (aCRF) – a PDF document. For a particular SDTM variable, a user clicks on the hyperlink and the annotated CRF opens to the variable's origin page. Depending on how these links are created in the define.xml, occasionally the page hyperlink fails to open the correct annotated CRF page. Manually testing each hyperlink is tedious and error prone. Fortunately, there are powerful Python modules for analyzing PDF and XML files. In this paper, we describe a technique using the Python programming language that checks each define.xml link against each page in the CRF PDF document. The techniques presented offer a good overview of basic Python techniques that will educate programmers at all levels.

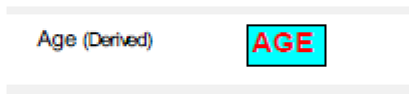
## INTRODUCTION

Hand-checking the resolution of define.xml hyperlinks is very time-consuming. For example, in the define.xml, suppose we navigate to the variable AGE:

| AGE | Age | | integer | 8 | | CRF Page 3 |
|---|---|---|---|---|---|---|

**Display 1. Rendered define.xml example**

If we click on the link "3" above, we should go to page three in the annotated CRF (a PDF file):



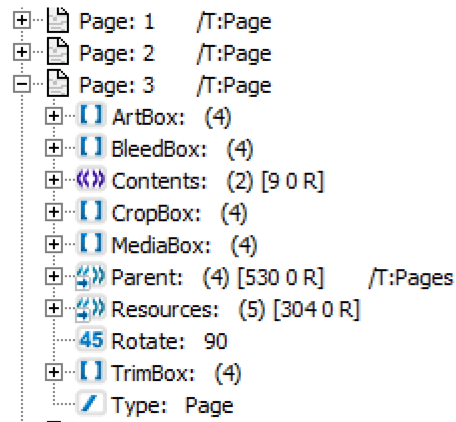

**Display 2. Annotated CRF example**

If the wrong page is presented, the metadata must be corrected such that the correct page is represented in the define.xml. Obviously, it isn't desirable to do this for the entire define.xml.
Behind the scenes both PDF and XML are structured in a tree-like fashion. Python provides the ability to navigate through these trees in both files and find matches. For example, in the raw define.xml, AGE is found at this branch:

```
<ItemDef OID="IT.DM.AGE" Name="AGE" SASFieldName="AGE" DataType="integer" Length="8">
        <Description>
                <TranslatedText xml:lang="en">Age</TranslatedText>
        </Description>
        <def:Origin Type="CRF">
        <def:DocumentRef leafID="LF.CRF" >
                        <def:PDFPageRef Type="PhysicalRef" PageRefs="3"/>
        </def:DocumentRef>
        </def:Origin>
</ItemDef>
```

**Display 3. Raw define.xml example 1**

Note at this branch the Origin is CRF and the PageRef = 3. In the annotated CRF, we view the tree structure on page three:




**Display 4. PDF tree**

In this tree we can find the string "AGE". If there is a match, we assume the link is working properly

## PYTHON TOOLS

In this paper, we use the `xml.etree.ElementTree` module to analyze the define.xml. For analyzing PDF files we use the `PDFMiner` module. The methods in this paper make use of the modules in the following way:

1. Use `xml.etree.ElementTree` to loop through each node to where the page number resides in the define.xml.
2. When the loop encounters the page number, use `PDFMiner` to open the aCRF at that page. Scan the page with regular expressions to check for the variable name.

All the code presented below were submitted using Python 3.6

## NAVIGATING TREES: XML

The first step in this process is to scour the define.xml file and find where the origin of the variable is "CRF", i.e., the source PDF file. The define.xml follows the Operational Data Model (ODM) schema. If you open the define.xml in a text editor, and search for "PDF", you will see blocks like this:

```
<ItemDef OID="IT.AE.AETERM" Name="AETERM" SASFieldName="AETERM" DataType="text" Length="200">
          <Description>          <!-- pgmLoc  6.36 layer  8.10 -->
                    <TranslatedText xml:lang="en">Reported Term for the Adverse Event</TranslatedText>
          </Description>
          <def:Origin Type="CRF">
          <def:DocumentRef leafID="LF.CRF" >
                              <def:PDFPageRef Type="PhysicalRef" PageRefs="42"/>
          </def:DocumentRef>
          </def:Origin>
</ItemDef>                  <!-- Variable AE.AETERM -->
```

**Display 3. Raw define.xml example 2**

Notice the information we have at our disposal: domain name variable name. However, the most important pieces are "PDFPageRef" and PagRefs="42". For the variable AETERM, if the user clicks on "42" in the rendered version of the define.xml, the annotated CRF should open at page 42. In Python, we use the `xml.etree.ElementTree` module to navigate the tree.

Here we import the `etree` module as well as the regular expressions module `re` and read the XML file.

```
import xml.etree.ElementTree as ET, re
inputxml = 'path\file.xml'
tree = ET.parse(inputxml)
```

Note that in the Display 3 above, our information is stored in the `ItemDef` branch. We can navigate to that branch by using:

```
for node in tree.findall('.//{http://www.cdisc.org/ns/odm/v1.3}ItemDef'):
    print(node.tag,node.attrib)
```

Partial output:

```
{'OID': 'IT.AE.AESOCCD', 'Name': 'AESOCCD', 'SASFieldName': 'AESOCCD',
'DataType': 'integer', 'Length': '8'}
{'OID': 'IT.AE.AESPID', 'Name': 'AESPID', 'SASFieldName': 'AESPID',
'DataType': 'text', 'Length': '25'}
{'OID': 'IT.AE.AESTDTC', 'Name': 'AESTDTC', 'SASFieldName': 'AESTDTC',
'DataType': 'partialDatetime'}
{'OID': 'IT.AE.AESTDY', 'Name': 'AESTDY', 'SASFieldName': 'AESTDY',
'DataType': 'integer', 'Length': '8'}
{'OID': 'IT.AE.AETERM', 'Name': 'AETERM', 'SASFieldName': 'AETERM',
'DataType': 'text', 'Length': '200'}
```

**Output 1. Output from print function**

This gives us the attributes for the `ItemDef` node/child, and from this dictionary, we can extract the variable name and the corresponding domain. However, to arrive at the page number we navigate further in the tree.

```
for node in tree.findall('.//{http://www.cdisc.org/ns/odm/v1.3}ItemDef'):
    domain = node.attrib['OID'].split('.')[1]
    variable = node.attrib['SASFieldName'].strip()
    for child in node:
        for grandchild in child.iter():
            if re.search(r'PDF',grandchild.tag):
                page = grandchild.attrib['PageRefs']
                print('Variable: ',variable,', Page: ',page)
```

Partial output:

```
Variable:  AESER ,   Page:  42
Variable:  AESEV ,   Page:  42
Variable:  AESHOSP , Page:  42
Variable:  AESLIFE , Page:  42
Variable:  AESMIE ,  Page:  43
Variable:  AESTDTC , Page:  46
Variable:  AETERM ,  Page:  42
Variable:  SEX ,     Page:  3
Variable:  SUBJID ,  Page:  1
Variable:  DSDECOD , Page:  10 44
Variable:  DSSTDTC , Page:  3 10 44 45
Variable:  DSTERM ,  Page:  44 45
```

**Output 2. Output from print function**

Notice how the page numbers are sometimes a sequence of values (for example, 10 44). We use slicing to parse out each page number. Full syntax:

```
for node in tree.findall('.//{http://www.cdisc.org/ns/odm/v1.3}ItemDef'):
    domain = node.attrib['OID'].split('.')[1]
    variable = node.attrib['SASFieldName'].strip()
    for child in node:
        for grandchild in child.iter():
            if re.search(r'PDF',grandchild.tag):
                page = grandchild.attrib['PageRefs']
                page_list = page.split()
                for j in range(len(page_list)):
                    xmlpage = int(page_list[j])-1
                    print("Variable = ",variable,", CRF Page:, ",xmlpage )
```

Partial output:

```
Variable =  AESER ,   CRF Page:,  41
Variable =  AESEV ,   CRF Page:,  41
Variable =  AESHOSP , CRF Page:,  41
Variable =  AESLIFE , CRF Page:,  41
Variable =  AESMIE ,  CRF Page:,  42
Variable =  AESTDTC , CRF Page:,  45
Variable =  AETERM ,  CRF Page:,  41
Variable =  SEX ,     CRF Page:,  2
Variable =  SUBJID ,  CRF Page:,  0
Variable =  DSDECOD , CRF Page:,  9
Variable =  DSDECOD , CRF Page:,  43
Variable =  DSSTDTC , CRF Page:,  2
Variable =  DSSTDTC , CRF Page:,  9
Variable =  DSSTDTC , CRF Page:,  43
Variable =  DSSTDTC , CRF Page:,  44
Variable =  DSTERM ,  CRF Page:,  43
Variable =  DSTERM ,  CRF Page:,  44
```

**Output 3. Output from print function**

Now we have all page numbers for each variable as they appear in the define.xml. Notice that SUBJID is on page 0. We subtract one from the page numbers to align the values with the PDF files. PDF pages always begin on page zero. We are now in position to pass these values to `PDFMiner`.

## NAVIGATING TREES: PDF

Navigating the PDF tree is very complicated, since PDFs contain more than just text – graphics for example. Fortunately, we have `PDFMiner` to do the heavy lifting and retrieve the text we need. Given the complexity of the PDF structure, we use several modules. Here are the imports for `PDFMiner` relevant to extracting text from the aCRF:

```
from pdfminer.pdfinterp import PDFResourceManager
from pdfminer.pdfinterp import PDFPageInterpreter
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from pdfminer.pdfpage import PDFPage
```

The details of these are described in Yusuke Shinyama's documentation on `PDFMiner`. And to fully understand them, we advise the reader to review each module's source code.

Our goal is to create an interpreter by combining a resource manager and device – in our case we use a text converter device. Once we have the interpreter, we scan over each PDF page and extract text. For illustration, here we scan over the first page of the aCRF without using the interpreter:

```
rsrcmgr = PDFResourceManager()
retstr = StringIO()
laparams = LAParams()
device = TextConverter(rsrcmgr, retstr, laparams=laparams)
fp = open(inputpdf, 'rb')
interpreter = PDFPageInterpreter(rsrcmgr, device)
for pageNumber, page in enumerate(PDFPage.get_pages(fp)):
    if pageNumber == 1:
        print(page)
```

Output:

```
<PDFPage:
  Resources={'ColorSpace':
  {'Cs10': <PDFObjRef:432>, 'Cs11': <PDFObjRef:431>, 'Cs6': <PDFObjRef:558>,
   'Cs9': <PDFObjRef:560>},
  'ExtGState': {'GS1': <PDFObjRef:561>, 'GS2': <PDFObjRef:562>},
  'Font': {'TT3': <PDFObjRef:571>, 'TT4': <PDFObjRef:574>, 'TT5':
<PDFObjRef:429>, 'TT6': <PDFObjRef:430>},
  'ProcSet': [/'PDF', /'Text', /'ImageC', /'ImageI'],
  'XObject': {'Im3': <PDFObjRef:255>, 'Im4': <PDFObjRef:253>}},
  MediaBox=[0, 0, 612, 792]>
```

**Output 4. Output from print function**

In Output 4, you see the overlap with some of the nodes in Display 4 – e.g. MediaBox, Resources, etc. Notice these data are at a high level in the PDF tree. In order to extract the text on the page, we use the interpreter to process the page:

```
rsrcmgr = PDFResourceManager()
retstr = StringIO()
laparams = LAParams()
device = TextConverter(rsrcmgr, retstr, laparams=laparams)
fp = open(inputpdf, 'rb')
interpreter = PDFPageInterpreter(rsrcmgr, device)
for pageNumber, page in enumerate(PDFPage.get_pages(fp)):
   if pageNumber == 1:
     interpreter.process_page(page)
     text = retstr.getvalue()
     print(text)
```

Partial Output:

```
Version 3.0 01MAR2018 Page | 10

SUPPAE.AEACNPSUPPAE.AECMYNSUPPAE.AEDISSUPPAE.AERELPAE.AEACNAE.AEENDTCAE.AEOUT
AE.AERELAE.AESERAE.AESEVAE.AESPIDAE.AESTDTCAE.AETERMDM.DTHDTCDM.DTHFLAE =
Adverse EventsDM = Demographics
```

**Output 5. Output from print function**

The output, albeit not aesthetically pleasing, contains the information we need – SDTM domain and variable names. We now combine this logic with the XML logic from above.

## PUTTING IT ALL TOGETHER

Now that we have our XML and PDF logic, we wrap them together. Recall we scroll through the define.xml and when we encoutner the aCRF page number we check the aCRF's page for the variable existence.

```
fp = open(inputpdf, 'rb')

tree = ET.parse(inputxml)
for node in tree.findall('.//{http://www.cdisc.org/ns/odm/v1.3}ItemDef'):
    domain = node.attrib['OID'].split('.')[1]
    variable = node.attrib['SASFieldName'].strip()
    domvar = domain+'.'+variable
    for child in node:
        for grandchild in child.iter():
            if re.search(r'PDF',grandchild.tag):
                page = grandchild.attrib['PageRefs']
                page = re.sub(r',',r' ',page)
                page_list = page.split()
                for j in range(len(page_list)):
                    xmlpage = int(page_list[j])-1
                    for pageNumber, page in enumerate(PDFPage.get_pages(fp)):
                        if pageNumber == xmlpage:
                            interpreter.process_page(page)
                    text = retstr.getvalue()
                    if re.search(variable,text):
                        print('Success,',domvar,'found on page =', xmlpage+1)
                    else:
                        print('Failure,',domvar,'not found on page =',
                              xmlpage + 1)
fp.close()
```

Using regular expressions (`re.search()`), we find the variable name on that page. If the string is found, we print a success message, otherwise we print a failure message.

```
Success, AE.AESER found on page = 41
Success, AE.AETERM found on page = 41
Failure, LB.LBSTRESU not found on page = 6
```

**Output 6. Output from print function**

## CONCLUSION

The methods presented work well, but are not the most efficient. Even when isolating to page numbers found in the XML tree, enumerating is time consuming. This is particularly true for large aCRFs. Secondly, the program we present only works on 'flattened' PDFs, which put all annotations as part of the PDF data stream. In other words, all annotation boxes are no longer editable. Lastly, this program will not detect errors in which the page number entered is larger than the size of the aCRF. For example, if the aCRF is 100 pages and a user accidentally enters 110 in the metadata (which flows into the define.xml), this case will not be counted as a failure.

Despite the caveats outlined above, the methods presented give a Python programmer a good place to start for building a define.xml/aCRF checking tool.

## REFERENCES

CDISC define.xml Team. 2005. “Case Report Tabulation Data Definition Specification (define.xml).” Accessed April 27, 2019
https://www.cdisc.org/system/files/all/standard_category/application/pdf/crt_ddspecification1_0_0.pdf

Shinyama, Yusuke. “Programming with PDFMiner”. 2013. Available at
https://pdfminerdocs.readthedocs.io/programming.html

## ACKNOWLEDGMENTS

Eva J. Welch
Steve Noga

## CONTACT INFORMATION

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

Brandon Welch
Rho Inc.
919-595-6592
Brandon_Welch@rhoworld.com

## APPENDIX

```
import re
import xml.etree.ElementTree as ET
from pdfminer.pdfinterp import PDFResourceManager
from pdfminer.pdfinterp import PDFPageInterpreter
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from pdfminer.pdfpage import PDFPage
from io import StringIO

rsrcmgr = PDFResourceManager()
retstr = StringIO()
codec = 'utf-8'
laparams = LAParams()
device = TextConverter(rsrcmgr, retstr, codec=codec, laparams=laparams)
interpreter = PDFPageInterpreter(rsrcmgr, device)

inputxml = r'E:\python\xml_pdf\define.xml'
inputpdf = r'E:\python\xml_pdf\acrf_flattened.pdf'
fp = open(inputpdf, 'rb')

tree = ET.parse(inputxml)
for node in tree.findall('.//{http://www.cdisc.org/ns/odm/v1.3}ItemDef'):
    domain = node.attrib['OID'].split('.')[1]
    variable = node.attrib['SASFieldName'].strip()
    domvar = domain+'.'+variable
    for child in node:
        for grandchild in child.iter():
            if re.search(r'PDF',grandchild.tag):
                page = grandchild.attrib['PageRefs']
                page = re.sub(r',',r' ',page)
                page_list = page.split()
                for j in range(len(page_list)):
                    xmlpage = int(page_list[j])-1
                    for pageNumber, page in enumerate(PDFPage.get_pages(fp)):
                        if pageNumber == xmlpage:
                            interpreter.process_page(page)
                    text = retstr.getvalue()
```

```
                    if re.search(variable,text):
                        print('Success,',domvar,'found on page =', xmlpage + 1)
                    else:
                        print('Failure,',domvar,'not found on page =', xmlpage + 1)
fp.close()
```