PharmaSUG 2015 - Paper BB04

Process and Programming Challenges in Producing Define.xml

Mike Molter, d-Wise, Morrisville, NC

ABSTRACT

Companies all over the pharmaceutical industry are looking for solutions for creating define.xml. The unfamiliarity of a foreign format like XML; the business rules surrounding the content; the intricacies of the metadata - all of these paint a picture of a daunting task that requires specialized software to be operated by specialists in the field. In this paper we'll see that while budget-saving Base SAS® solutions do exist, the task is not trivial. We'll also see that while programming challenges are part of the puzzle, challenges in setup and process are where much of the task's energy and resources need to be dedicated.

INTRODUCTION

On the surface, the creation of define.xml appears to be a daunting and an intimidating task. For some SAS programmers, XML is a foreign language, requiring the services of a specialist with special software. For those programmers, this paper brings good news - all programmers with a basic knowledge of how the DATA step works have the necessary tools to build a define.xml. Part of the beauty of an XML file is that it's just text. Simple FILE and PUT statements inside the DATA step handle this easily. For that reason, coding will not be a central theme of this paper. Of course the metadata you're reading with the DATA step and writing to the text file has to come from somewhere. This is where the fun begins.

Define.xml is the metadata that describes the data for a clinical trial. More specifically, it is a combination of what I refer to as "clinical metadata" and "technical metadata." Clinical metadata is defined by a set of properties that speak to the clinical side of the data. Examples include the origin of the data (did a certain piece of data come from the CRF, was it derived, or did it come from another source?), the data set purpose (e.g. tabulation or analysis), as well as comments that explain domains or variables within domains to a reviewer. Metadata like this tends to evolve over the course of study design as well as throughout the trial itself, and is oftentimes not easily captured in a machine-readable way. Even when it is we don't always take the opportunity to do so. For these reasons, this kind of metadata often must be manually entered. Technical metadata consists of the attributes we typically associate with PROC CONTENTS output, such as the name, label, type, and length of a variable. While PROC CONTENTS may seem like a way to programmatically capture at least some of the necessary metadata, if we look closer, this isn't exactly the case. For starters, the LENGTH and TYPE properties required by define.xml are defined slightly differently than what SAS makes available. Even if these properties were the same, the output from this procedure merely reflects metadata that at one time was part of data specifications - a document whose population can be as manual as any we use to store clinical metadata.

Software exists these days that is capable of producing define.xml, and also capable of collecting metadata with minimal risk. This paper however is about producing define.xml with nothing more than BASE SAS. Along those lines, it is also written with the assumption that simple data entry tools such as Microsoft Excel or Access are used for collecting most, if not all, of the metadata. While this may not be ideal, it is, in practice, not uncommon for any number of reasons including cost, availability of resources, and others.

Such an approach requires a sustainable process for collecting and writing this metadata. It must consider resources available, the different roles they play, and the various levels of expertise they possess. For those entering metadata - what this paper will refer to as *users*, it must be intuitive and easy to use. Where possible and applicable, it should have controls on the type of text entered, or at the very least, checks that are executed while the metadata is being processed and before it is used. We also need to decide what programmers are going to do with the metadata collected. In this paper we will assume two programming stages. In the first stage, a *database programmer* imports the collected metadata and molds it into a *define database.* What should this database look like? How many data sets are in it? What variables are contained in each data set, and what does an observation represent in each? How, if at all, are these data sets related to each other? All of these decisions must be made armed with a foundational knowledge of define.xml and an understanding of what's realistic in an organization. In the second stage, with a well thought-out, intelligent define database, a *define programmer* uses the define database and straightforward data step code to write the XML.

What we'll concentrate on in this paper is considerations for how to structure our collection to make the process as intuitive as possible for users, while at the same time leading to an easy-to-use define database and ultimately to a define.xml. In this paper we'll use the term *designer* to refer to a person in the role of designing a collection system and a define database. It is the intention of this paper, not to recommend any one approach, but rather, to bring to the surface some of the aspects of define.xml that force a designer to consider these decisions, and present options.

The paper will begin with a high-level look at the layout of define.xml and the main sections contained within. As we then move into some of the details, we will discuss how the define database must account for such intricacies. We'll also explore and contrast the needs of the database against the needs for collection. The paper will conclude with a discussion of the challenges behind collection. The reader is expected to have a basic understanding of XML and some of its terminology. Although the paper begins with a brief description of the define.xml structure, some prior knowledge of the subject is helpful.

DEFINE.XML - AN OVERVIEW

If we think of metadata as a structured way of defining something we'll generically call *objects*, then it's not a significant stretch to think of define.xml as a document of definitions. In a sense, we can think of it as the dictionary of our study. But before going too far with this analogy, let's see how Webster defines its objects a little differently.

For starters, Webster defines only one kind of object. We call it a *word*. Define.xml defines several kinds of objects. One of the more prominent kinds of objects is the *item*. We can think of an item as anything that can conceptually pass as a variable. This includes variables from your data sets - what we'll call *real variables* - as well as *virtual variables* - something we might think of conceptually as a variable, but due to the vertical nature of submission data sets, ends up being the values of a variable for a subset of observations. Examples include a subject's height or a particular lab test result.

An *item group* is a second kind of object defined by define.xml. If we think of an item group as a grouping of variables, then it makes sense that item groups define data sets. Other kinds of objects defined by define.xml include the following:

- code list - a set of values allowed for one or more items
- value list - a list of all virtual variables that contribute to the definition of a real variable
- leaf - external files that accompany a study submission
- where clause - subsetting conditions
- method - algorithms used to derive item references
- comment - comments used to explain items or item groups

Second, Webster defines words with free, unstructured text. Define.xml defines its objects with the use of a distinct set of *properties*. Each different kind of object is defined by a unique set of properties. For example, the name, ODM data type, ODM length, and label are properties that contribute to the definition of an item. An item group's definition includes properties such as the name, label, and a free-text description of the structure of the data set as well as its list of real variables. This way of defining objects allows us to clearly state that two objects of the same type are the same objects only if they agree on the values of all defining properties. Put another way, two objects that appear to be the same (e.g. have the same value of the Name property) are different if their values of at least one defining property differ.

This leads to our third difference. Webster doesn't define words with properties. Therefore, we both know and identify a word by the way we spell it. In casual conversation, we tend to know and identify an item or a variable by its name, or in other words, the value of its Name property. Define.xml, on the other hand, identifies objects by an identifying property, or what is formally called an object identifier, or OID. The OID is not a defining property. Its value has no meaning in the context of clinical trial data, and so conventions for assigning their values are strictly the decision of the organization creating the file. We'll soon see the value of having a unique identifier whose value is not tied to the defining metadata.

Define.xml requires exactly one instance of each unique definition. This generally means one item group per data set in the submission, but this might not be true of variables. For example, if a variable is defined one way in one data set, but the same-named variable is defined differently, with different values for at least one property in another, then two definitions must be documented because they are considered different items, and must therefore have different OID values. USUBJID is an example of a variable that appears in several SDTM data sets, but may only need one item definition because it can be defined the same way across data sets. On the other extreme, PARAMCD will most likely require one item definition for each BDS data set in your ADaM submission. This is because one of the defining properties - list of allowable values or codelist - will be different for each BDS data set. VISIT might be an example in between these two extremes.

How do these concepts translate into XML? At the top of define's element structure in the order in which they are nested, are the ODM element (the root element whose attribute describes the file) and the Study element. Alongside elements that contain study-level information, the Study element also nests the MetaDataVersion element. Nested

within MetaDataVersion are elements that correspond to all of the definition types noted above. The names of these definition elements are provided in Table 1 below.

Definition Type	Element Name
item group	ItemGroupDef
item	ItemDef
code list	CodeList
value list	def:ValueListDef
leaf	def:leaf
where clause	def:WhereClauseDef
method	MethodDef
comment	def:CommentDef

Table 1 - Definition types and their corresponding define.xml element names

Note that most (but not all) of these elements end with the suffix "def". These elements are sometimes casually known as *defs*. The properties that define metadata objects appear as XML attributes, either on the element definition itself or on a nested child element. The name of the property translates to the name of the attribute, and the value of the property becomes the attribute value, enclosed in quotes. OID properties translate into attributes on the definition element called OID. The exception to this rule is the def:leaf element whose OID attribute is called ID.

CONNECTING THE DOTS

So far the picture we have painted of define.xml is one of a collection of several different types of definitions. We have definitions of data sets, then definitions of variables, then definitions of allowable values, and so on, but at this point, they don't seem to have anything to do with each other. The heart of define.xml is in how these different definition types connect with each other. We know that data sets have variables, and we have both data set and variable definitions, so how do we connect a data set definition with the variable definitions it contains? We know that some variables have a list of allowable values. We have variable definitions, and we have lists of allowable values, so how do we connect a variable definition with the right set of allowable values? This is where OIDs become important.

While many of the properties that define objects have simple text strings for values, others are more complex and need to be defined. Rather than defining them inline, we populate the attribute with a reference to a target definition. More specifically, the reference is to the value of the OID attribute in the target definition element that defines the property. For the most part, these referencing properties originate in a child element of the source definition element. The name of this child element typically ends with "ref". We sometimes casually refer to such elements as *refs.* The OID reference is made in an attribute of the child ref whose name usually ends with "OID". We'll soon see examples of how data sets are linked to their variables, variables are linked to their codelists, and other sections are linked to definitions through the use of refs and OIDs.

We can think of the references in a define.xml the way we think of references in a book. Consider the following fictional book passage.

> Henry loved to read. He would read about any subject, from the theory of relativity[a] to Mark Twain[b] to Babe Ruth[c]. But his favorite subject was economics.

When reading a passage such as the one above, the reader knows that by following a reference, identified by the use of a superscript immediately following the text being described, either to the bottom of the page or to a section in the back of the book, they can obtain information about the text that the author didn't want to include in the main text. This allows an author to provide explanations to readers that are interested without breaking the flow of the story and distracting the reader. In the above example, a reader that follows the reference identified by the superscript "b" will find information about Mark Twain, while following the "c" reference takes the reader to information about Babe Ruth. We can think of define.xml as a document filled with similar references. The value of an OID attribute is analogous with the text in the superscript - we can follow its value to a definition element that contains an OID attribute with the same value to know more about what is being described. This creates natural links between two parts of the define.xml file.

In order to see this linking mechanism in action, we'll start with a simple example. Define.xml allows an organization to provide information about external documentation that supports the submission. Annotated case report forms (aCRFs) must be submitted and so must be documented in define.xml, but documentation of other submitted documents such as a reviewer's guide, a statistical analysis plan and others is encouraged too. Define.xml uses the definition reference mechanism for this documentation. The def:AnnotatedCRF element contains a list of each of the aCRFs submitted in the form of reference elements - the def:DocumentRef. The def:SupplementalDoc element also contains a series of def:DocumentRef elements for referencing other documents. def:DocumentRef has an attribute called leafID, an OID element (and an exception to the rule that OID elements end in "OID") whose value is meaningless, except for its match to the value of the ID attribute (another exception) of a def:leaf element. The

def:leaf element, the definition element, is where the document is defined in more detail. The def:SupplementalDoc element and its assocated def:leaf elements are illustrated in Figure 1 below.

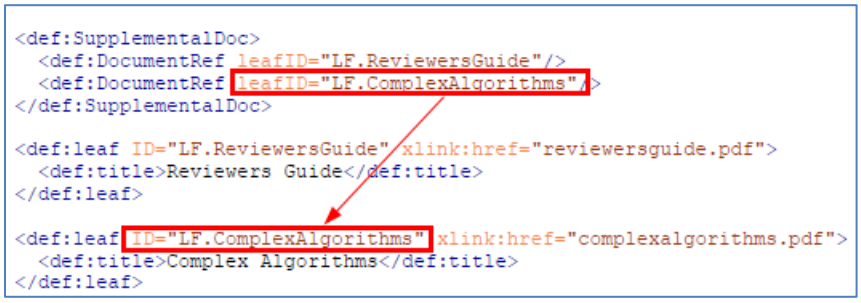


Figure 1 - Link from a document reference to its corresponding definition

Why is this linking mechanism necessary? Earlier we compared the use of references in define.xml to the use of references in a book - a way to refer a reader to a definition without distracting from the body of the text. In this case the definition seems short enough to make the use of such a mechanism unnecessary. A second reason though is that a single instance of a definition can be referenced multiple times. For example, the reviewer's guide is referenced from a list of documents as we see in Figure 1, but as we'll see later, it can also be referenced from a comment definition or a method definition. Similarly, codelist definitions can be referenced by more than one item definition. The definition reference mechanism allows us to define something like a document, an item, or a codelist only once rather than every time it is referenced.

These links occur all over the define.xml and we'll look more closely at other examples later. For now we'll turn our attention to collection and programming. Given these linking requirements, what are our options for collecting metadata and then turning it into something we can easily write into a define.xml?

COLLECTION, PROGRAMMING, AND DATA STRUCTURE

We'll soon start considering how we might structure our define database in order to write out the different sections of define.xml that we've discussed thus far, but for now, one general notion should be clear. While it is technically possible to stuff all of the metadata into one data set, best practices as well as ease of management would more likely lead us to split the variety of different types of metadata across the different sections into multiple data sets. At a high level, our program should contain a FILENAME statement (or a FILENAME function) that points to the XML file we're creating, and multiple DATA steps that use the FILE statement to direct output to the XML file.

```
filename define "mypath/define.xml" ;

/* data step 1 */
data _null_;
file define lrecl = pre-chosen record length ;
…
/* data step n */
data _null_ ;
file define mod lrecl = pre-chosen record length ;
```

A couple of observations of the code above are noteworthy here. We first note the use of the MOD option on the FILE statement in the last DATA step. This needs to be used in all DATA steps after the first, so that all output is appended to the previous output, rather than replacing it. The second observation is the use of the LRECL option. This might not always be necessary, but it is for sections that may have long text strings, such as comments or methods.

Two other general considerations should be taken into account. The use of LRECL allows us to increase the number of characters we can fit on a line, but we still have the danger of splitting text from one line to the next when we hit that limit on a given line, no matter how long it is. To avoid this, we might consider a macro to be applied to metadata values that have the potential of exceeding the LRECL in length (e.g. comments and methods). This macro would break the text string into pieces at spaces and write each piece to its own line in the output file.

A second consideration is special characters. Specifically, XML has specific purposes for angle brackets, quotation marks, apostrophes, and ampersands. Each instance of one of these characters in element or attribute values must be replaced by a corresponding XML entity. These entities are summarized in Table 1 below.

Character	Replacement
<	&lt;
>	&gt;
&	&amp;
‘	&apos;
“	&quot;

Table 1 - Special characters and their corresponding entities

COLLECTION

We know that define.xml is made up of distinct sections, each corresponding to what we've called up until now, an object. Each of these sections has its own set of metadata. Ideally, the metadata for each of these sections could be collected separately, in what we will refer to generically as its own *form.* A form might be an individual tab of an Excel workbook. We could then collect it in a structure that was immediately ready to be written to define.xml. But the need for links that reach across sections complicates matters. When collecting a certain kind of definition in one form, and that definition includes properties that refer to other definitions, how do you collect those referenced definitions? Do you collect them in the same form? In other words, do you ask for an object to be defined with each reference to it? We'll refer to this as the *Inline method.* If so, this means asking a user to repeat their definition specification. This means asking a user to be aware of when an object being referenced has already been defined. Not getting this right could lead to multiple objects being defined where only one was necessary. On the other hand, collecting referenced definitions in a separate form makes it easier to define the property just once, but it forces the user to know how to link to that referenced definition. We'll refer to this as the *Database method* because it is more in line with common database practices. It is also closer to the way define.xml is structured, and so should make for easier programming. Either way the user has to keep track of other definitions while defining another. This paper won't try to claim that any approach, either one of these two extremes or something in the middle, is the right way or the wrong way. Rather, we'll take a look at options for each required link, along with pros and cons. Individual organizations need to weigh these against their own practices in order to decide which approach works best for them.

Before we look at examples of specific links, let's come to an understanding of some basic principles around collection that should at least guide your decisions. The following are considerations centered on the user.

U1. Make the collection system as easy and intuitive as possible

   a. Collect only properties that users understand. For example, don't ask users to understand OIDs

U2. Avoid asking the user to specify anything more than once

U3. Collect only what is necessary

   a. if organizational practices determine the value of a certain property, then don't collect it

   b. if anything can be programmatically derived, then don't collect it

U4. Where possible, implement controls (e.g. dropdown menus) for properties whose values should be restricted.

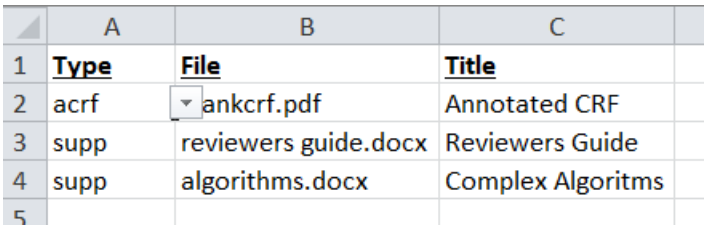
We mentioned above that while users define an object, they may have to know something about other definitions. This will mean that they'll have to know about links between forms, which in some way violates U1.a above, or they'll have to repeat definitions, which violates U2 above. Following all of the guidelines may not always be possible, but in considering their own practices, organizations can minimize risks with certain compromises.

Collection decisions should consider programming too. Programming should fill any gaps left as a result of a decision made for the benefit of users, but we'd also like to keep it as simple as possible. Ideally, we'd like to collect as close to the database method as we can to minimize data transformations. Again, the compromises made to minimize risk may be different from one organization to another, but most of the time, the predictability of tested SAS code, even if transformations are needed, will be preferred over the unpredictability of user input into a system that they don't fully comprehend.

DOCUMENT METADATA LINKS

Document metadata is a good place to start. Define.xml has two container elements whose purpose is to simply list all external documents that accompany the submission. def:AnnotatedCRF lists all annotated CRFs and def:SupplementalDoc lists all other documents. More specifically, between these two container elements, each document is referenced exactly once with the reference element def:DocumentRef. Additionally, the references are made only in isolation and not as part of another definition, as seen in the def:DocumentRef elements nested within def:SupplementalDoc in Figure 1 above. These two facts lead to a natural collection structure of one record for each document. The developer can choose to split the annotated CRF definitions from the rest of the document definitions

using two different forms if they choose, (since their references are in separate XML element containers) or keep them in the same form with a column that indicates whether a document is an aCRF or not. The former is illustrated in Figure 2 below.



	A	B	C
1	**Type**	**File**	**Title**
2	acrf	ankcrf.pdf	Annotated CRF
3	supp	reviewers guide.docx	Reviewers Guide
4	supp	algorithms.docx	Complex Algoritms
5			

Figure 2 - Storage of document metadata

Here we see that we've adhered to the guidelines above. We've kept collection to a minimum, collecting only what is necessary, adding dropdowns where the column data is to be restricted. Furthermore, because each definition is referenced only once, no decision needed to be made regarding the separation of references and definitions. If an organization uses the same file name or title for each kind of document across all studies, then perhaps more dropdowns could have been added, or the organization might feel that this metadata doesn't need to be collectd at all. From a programming point of view, the spreadsheet above has all that's needed to generate both the container elements for the references (def:AnnotatedCRF and def:SupplementalDoc, each containing def:DocumentRef elements) and the definition elements (def:leaf). Figure 1 above shows us that we need a leafID attribute for the def:DocumentRef reference elements, and a matching ID attribute for the corresponding def:leaf elements, but these can be derived in programming and so don't need to be collected. Table 2 below explains each of the spreadsheet columns and where they are inserted into the XML.

Variable name	Explanation	XML location
type	Used only to distinguish between aCRF and non-aCRF document metadata	NA
file	location of file relative to location of define.xml	Value of xlink:href attribute of def:leaf
title	Description of file	Value of def:title element nested within def:leaf

Table 2 - variable explanation of document metadata

As mentioned earlier, conventions for OID values are up to the submitting organization. The following code creates a single data set for the define database. It creates OID values as a combination of TYPE ("acrf" or "supp") and a unique zero-padded numeric suffix.

```
proc sort data=docmdfromexcel ;
by type ;
run;

data definedb.documentmd;
length oid $6 ;
set docmdfromexcel;
by type ;
if first.type then counter = 0 ;
counter + 1 ;
oid = cats(type,put(counter,z2.)) ;
run;
```

The following then generates the container elements with their reference child elements from the define database.

```
data _null_ ;
file define mod ;
set definedb.documentmd;
by type ;
if first.type then do;
  if type eq 'acrf' then put '<def:AnnotatedCRF>' ;
  else put '<def:SupplementalDoc>' ;
end;
put '<def:DocumentRef leafID="' oid '"/>' ;
if last.type then do;
  if type eq 'acrf' then put '</def:AnnotatedCRF>' ;
```

```
  else put '</def:SupplementalDoc>' ;
end;
run;
```

The following generates the definition elements.

```
data _null_ ;
file define mod ;
set definedb.documentmd;
   put '<def:leaf ID="' oid '" xlink:href="' file '">' ;
   put '<def:title>' title '</def:title>' ;
   put '</def:leaf>' ;
   run;
```

This was all that was necessary in version 1, but version 2 of define.xml throws in a twist. In version 2, documents can be referenced from more than just these containers. They can also be referenced from method definitions and comment definitions (also new to version 2). More specifically, comment and method definitions can nest the same def:DocumentRef elements that the other containers nested. This becomes useful when a comment or a method is better explained by pointing the consumer of the file to an external document rather than try and reproduce it. Figure 3 illustrates an example.

```
<!-- Method Definition: Algorithm included or expanded in an external file -->
<MethodDef OID="MT.EGDRVFL" Name="Algorithm to derive EGDRVFL" Type="Computation">
  <Description>
    <TranslatedText xml:lang="en">EGDRVFL = "Y" for derived EGTESTCDs QTCB and QTCF.
      Null otherwise. </TranslatedText>
  </Description>
  <def:DocumentRef leafID="LF.ComplexAlgorithms">
    <def:PDFPageRef PageRefs="EG" Type="NamedDestination"/>
  </def:DocumentRef>
</MethodDef>
<def:leaf ID="LF.ComplexAlgorithms" xlink:href="complexalgorithms.pdf">
  <def:title>Complex Algorithms</def:title>
</def:leaf>
```

Figure 3 - Document references in a method definition

Similarly, aCRF documents must be referenced from within the new def:Origin element when the value of its Type attribute is set to "CRF", as seen in Figure 4 below.

```
<!-- Item Definition: Variable Level (BRTHDTC) -->
<ItemDef OID="IT.DM.BRTHDTC" Name="BRTHDTC" DataType="date" SASFieldName="BRTHDTC">
  <Description>
    <TranslatedText xml:lang="en">Date/Time of Birth</TranslatedText>
  </Description>
  <def:Origin Type="CRF">
    <def:DocumentRef leafID="LF.blankcrf">
      <def:PDFPageRef PageRefs="6" Type="PhysicalRef"/>
    </def:DocumentRef>
  </def:Origin>
</ItemDef>
```

Figure 4 - aCRF references in the def:Origin element

Sometimes references are accompanied by properties that further describe the context of the reference. We'll refer to these as *reference properties.* In the XML, these translate into attributes and child elements of the reference elements. We see this in examples 3 and 4 above. The PageRefs attribute names page numbers or PDF destinations, while Type indicates whether the value of PageRefs is a named destination or page numbers. Because these properties are a function of the reference and not defining properties, it makes sense to collect them with the reference. But now that we're back to multiple references to a single definition, we're back to the same collection questions we asked before. Do we collect the references to definitions, along with their reference properties, thereby still collecting definitions only once but forcing users to know how to reference the definition? Or do we collect the defining properties each time the document is referenced, thereby collecting the same definition more than once and making programming more difficult, but relieving the user of having to know anything about definition references?

Figure 5 illustrates the database approach to collecting method definitions, collecting references to document definitions rather than document definitions (along with reference properties).

	A	B	C	D	E	F
1	**InlineMethod**	**MethodFile**	**PageRefs**	**Firstpage**	**LastPage**	**PageRefType**
2	(day of event) - (reference start date) + 1					
3	See pages 38-44 in Complex Algorithms.docx	algorithms.docx		38	44	PhysicalRef
4						

Figure 5 - Part of a method definition data set

The first row defines a method that is short enough to be described within the define document and doesn't need a reference to an external document. In the second row, the user has provided some inline text, but also a pointer to an external document where the method is fully defined. In this example, because the designer chose not to collect an OID attribute in the document definition form, he can't collect it here, and so must collect another property that can uniquely identify the appropriate document definition. Here he has chosen the name of the external document (MethodFile). This is more intuitive for the user, but it does force extra programming when building the define database to generate the value of the LeafID attribute.

The second option is to do away with the document definition form and collect defining properties where they're being referenced. While on the surface this appears to be less than ideal, it might not be so bad with document references. Recall that Figure 2 above showed us that documents are defined only by two properties. The first option forces us to enter one of them (the name of the document) just to link to the definition, so the second option simply means adding one more defining property to the reference collection. It does introduce the potential for more error since the Title property should be consistent for each reference to the same document. These are consequences that a designer will have to weigh against each other to make decisions on collection structure.

One more factor in a designer's decisions is business practices. If an organization has global policies regarding certain metadata properties, then the designer doesn't have to collect them. For example, if an organization has globally defined the value of the Title property for each kind of document, then collecting it from a user for each study is unnecessary, and even the need for a document definition form disappears. Perhaps these global values are stored in a central location to be brought into each study's define database. Such practices can make collection easier and reduce error potential.

It may seem that we've dedicated a significant portion of this paper to a relatively small part of define.xml - the document definitions, but in fact the principles centered around the linking mechanism discussed here apply to all other sections, and the same considerations have to be given to all other parts of the define database. In the next section we'll apply them to the link between data set definitions and their variables.

DATA SET AND VARIABLE DEFINITIONS (ITEMGROUPS AND ITEMS)

ItemGroup metadata contains much of what you would expect in defining a data set - name of the data set, its label, its class (data set category). It contains attributes that allow consumers to understand its structure, such as def:Structure (a description of what makes observations unique), IsReferenceData (a flag that indicates whether the data is subject-related), def:CommentOID (a reference to comments about the data set), and Repeating (a flag that indicates whether a subject has data on more than one observation). Item metadata properties describe variables - name, length, label, data type, as well as consumer information such as the origin of the variable, methods, comments, and references to other definitions. These properties make collection seem straightforward - one record per data set and one record per variable. Additional references, including a link between data set and variable definitions, make collection more challenging.

While variables are not associated with data sets, data sets do have variables, and a list of them is part of each data set definition. This list comes in the form of references, a series of ItemRef elements nested within ItemGroupDef. These references contain more than just pointers (OIDs) to item definitions. They also contain reference properties - attributes that describe the variable's role in the data set. So how does this affect collection of data set metadata?

Can we still collect with the one-record-per-data set structure mentioned above, or must we expand to one record per variable per data set? Or is there something in between? Insisting on one record per data set means cramming the list of variables into a single value of one variable. While programming code that parses such a string isn't impossible, it's not an attractive option from a database standpoint. The fact that we need to store additional information about each of these variables along with their references makes it far less attractive. For this reason, we'll assume that we need at least one form that contains one record per variable per data set. Let's look at some options.

One option is to insist on collecting all data set metadata together. We'll refer to this as Option 1. Collecting each variable of the data set in its own record means that while reference properties in the form will be unique across records, the collection of definition properties that describe the data set (e.g. name, label, class) must be repeated.

A second option (Option 2) is to split these properties into two forms. One form collects all of the data set properties except the list of variables. This form contains one record per data set. A second form contains one record per variable per data set along with the reference properties. Once again, splitting means we need a link or a merging variable that ties data sets to their variables. The name of the data set would be a good candidate.

While Option 2 may be appealing, note the programming requirements it brings with it. To a degree, our collection has introduced more separation than what's in define.xml. When the programmer builds the define database, he'll have to merge these two forms before writing to the file. The reason is that the variable references for one data set must appear before the data set metadata for the next. This might tempt the designer to go back to Option 1. Once again the pros and cons, this time of collecting repeated metadata vs. asking the programmer to merge, must be weighed against each other.

What about item metadata? Having already been forced to collect some information about each variable of each data set, it's not only tempting, but also common to collect all defining properties alongside the reference properties in Option 1 and Option 2. We'll refer to this option as Option 3. Unlike document metadata though, item metadata has many defining properties. This approach could lead unnecessarily, not only to the creation of several definitions where only one is needed, but also to extra work.

As you may have guessed, Option 4 is the collection of item-level metadata in its own form, along with a way to reference an item from the itemgroup form. This allows us to define each item only once, but requires a unique identifier to reference from the itemgroup form. While it's tempting to use the name of the variable as this link, remember that unlike a data set name, a variable name doesn't uniquely define an item. Since different tests are administered at different visits, multiple items may be defined in which VISIT is the value of the NAME property, but each has a different codelist. In ADaM we expect one item whose NAME value is PARAMCD for each BDS data set. For this reason, a designer has to get a little closer to collecting an OID value. He needs to add something to his collection to make the records in this form unique, but still make it easy on the user. In the itemgroup collection, it must be clear that the reference to the item is not meant to answer the question "which variable?", but rather, "which instance of which variable."

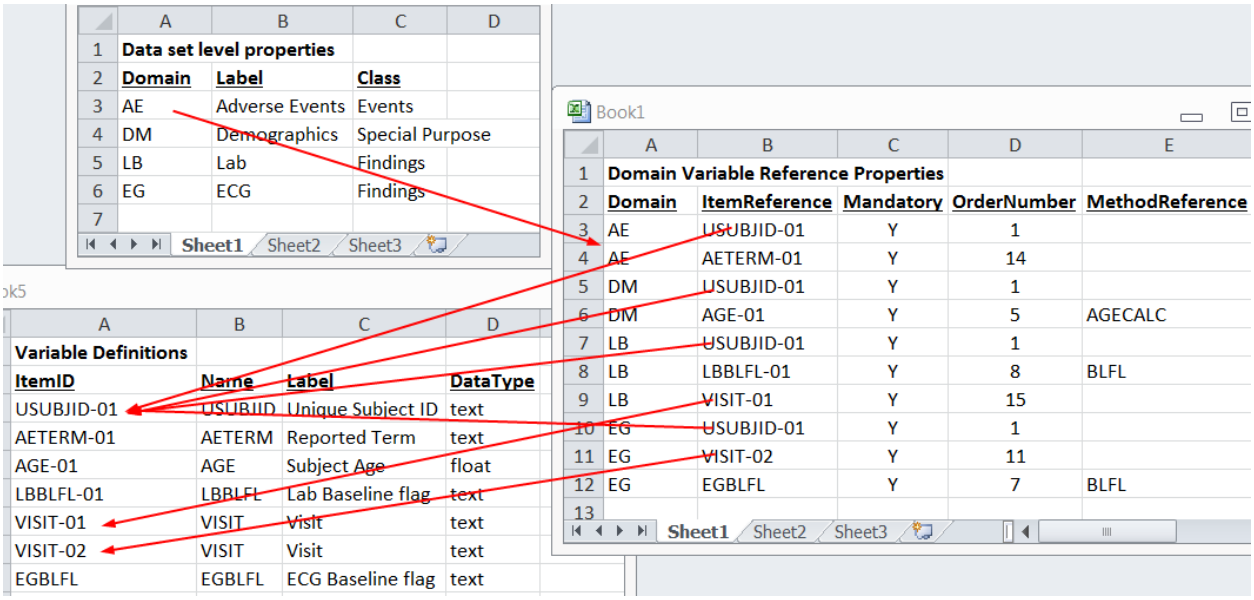


Figure 6 - Option 4 - a subset of the properties in each form and their connections

We've spent much time discussing the link between itemgroupdefs (data set metadata) to itemdefs (variable metadata), but other links exist from itemgroupdefs to other sections of the document. Table 3 below describes all links that originate within the ItemGroupDef element.

Source of link from within ItemGroupDef	Target of link that originates in ItemGroupDef	Description
ItemGroupDef/ItemRef/@ItemOID	ItemDef/@OID	Definition of variables in data set
ItemGroupDef/@def:ArchiveLocationID	ItemGroupDef/def:leaf/@ID	Definition of transport file corresponding to data

		set
ItemGroupDef/@def:CommentOID	def:CommentDef/@OID	Definition of Comment about the data set
ItemGropuDef/ItemRef/@MethodOID	MethodDef/@OID	Definition of Method used to derive the variable in the data set

Table 3 - Summary of links that originate in ItemGroupDef

While links exist from data set to variable metadata, links also exist from variable metadata to other sections. Table 4 summarizes these.

Source of link from within ItemDef	Target of link that originates in ItemDef	Description
ItemDef/CodeListRef/@CodeListOID	CodeList/@OID	Definition of the set of allowable values for the variable
ItemDef/def:ValueListRef/@ValueListOID	def:ValueListDef/@OID	For a given real variable, this is the definition of the set of associated virtual variables
ItemDef/@def:CommentOID	def:CommentDef/@OID	Definition of Comment about the variable
ItemDef/def:Origin/def:DocumentRef/@LeafID	def:leaf/@ID	Definition of the aCRF document from which the variable is a source

Table 4 - Summary of links that originate in ItemDef

We've now covered links that originate in ItemGroupDef and ItemDef. Earlier we looked at links that originate in method and comment definitions. We now turn our attention to a more advanced web of links in value-level metadata.

VALUE-LEVEL METADATA

One of the aspects of define.xml that involves multiple links is in the value-level metadata. Before we get into the details behind these links, let's quickly review what value-level metadata is.

Earlier we discussed the concept of a virtual variable. A virtual variable is one that doesn't show up as a SAS variable in a SAS data set, but whose values, along with values from other virtual variables, all show up in a single variable. With such a potpourri of data in this single variable, a second variable is used to identify the type of data in that variable on a given record. Two good examples are illustrated with SDTM's VS domain. We stuff a lot of information into VSORRES - height measurements, weight measurements, blood pressure, frame size, and others. Each of these measurements is a virtual variable. We can imagine, if it weren't for the vertical structure requirement of VS, each of these as a real SAS variable in a SAS data set. In fact the data sets that represent collection may look like this. Instead we put them all into VSORRES. So VSORRES is overloaded with different kinds of results. Some may be integers, some may have more precision than others, and in the case of frame size, some may be text. Normally, a variable's metadata is assumed to describe *all* values of the variable or all rows of the data set. In the case of VSORRES, we can see that one set of metadata is not sufficient to describe the variable. Rather, we need one set of metadata for each distinct value of VSTESTCD.

As a second example, we can say the same of VSORRESU. While it's safe to say that all of VSORRESU will be text, the set of allowable values, another defining property of a variable, varies with VSTESTCD. VSORRESU is a variable whose values range from "kg"and "lb" to "mmHg", "cm", and "in". But to try and describe VSORRESU and its allowable values with one codelist that includes all of these misses the fact that "kg" and "lb" are allowable values only when VSTESTCD="WEIGHT". For these reasons, we need to describe variables like VSORRES and VSORRESU with VSTESTCD-specific metadata, or value-level metadata.

Because these virtual variables each have their own set of metadata, it makes sense that each would have its own ItemDef element. The question is how such an ItemDef is referenced. We know that ItemDef elements that represent real variables are referenced from the ItemGroupDef element(s) that represent data set(s) that contain those variables, but no data set contains a virtual variable. The answer is in a value list.

When a variable like VSORRES must be described with value-level metadata, then the ItemDef element for that variable needs to point to a list of all of the virtual variables, such as HEIGHT, WEIGTH, DIABP, etc. These are all contained in a value list. This link begins in the def:ValueListRef child element of ItemDef, using the ValueListOID attribute as the link. The value of this attribute will match the value of the OID attribute in a def:ValueListDef element. Contained within def:ValueListDef is a set of ItemRef elements, each referencing the definition of a virtual variable that contributes to the real variable. These ItemRef child elements contain the same attributes that they do inside ItemGroupDef. Unlike those in ItemGroupDef, however, these contain a child element that references the condition

that defines the virtual variable. For example, the ItemRef that references the HEIGHT definition also points to a condition that defines which records in the data set identify the virtual variable - in this case, records on which VSTESTCD = “HEIGHT”. The reference is made using the element def:WhereClauseRef. This element contains an OID attribute called WhereClauseOID, whose value matches the value of OID in a definition element called def:WhereClauseDef. In this definition, the real variable (more specifically, the corresponding item) upon which the metadata depends (in this case, VSTESTCD), a comparator (e.g. “EQ”, “IN”), and one or more values are specified. The real variable is specified with the attribute def:ItemOID whose value matches the value of OID in the corresponding ItemDef. Figure 7 below illustrates the links.

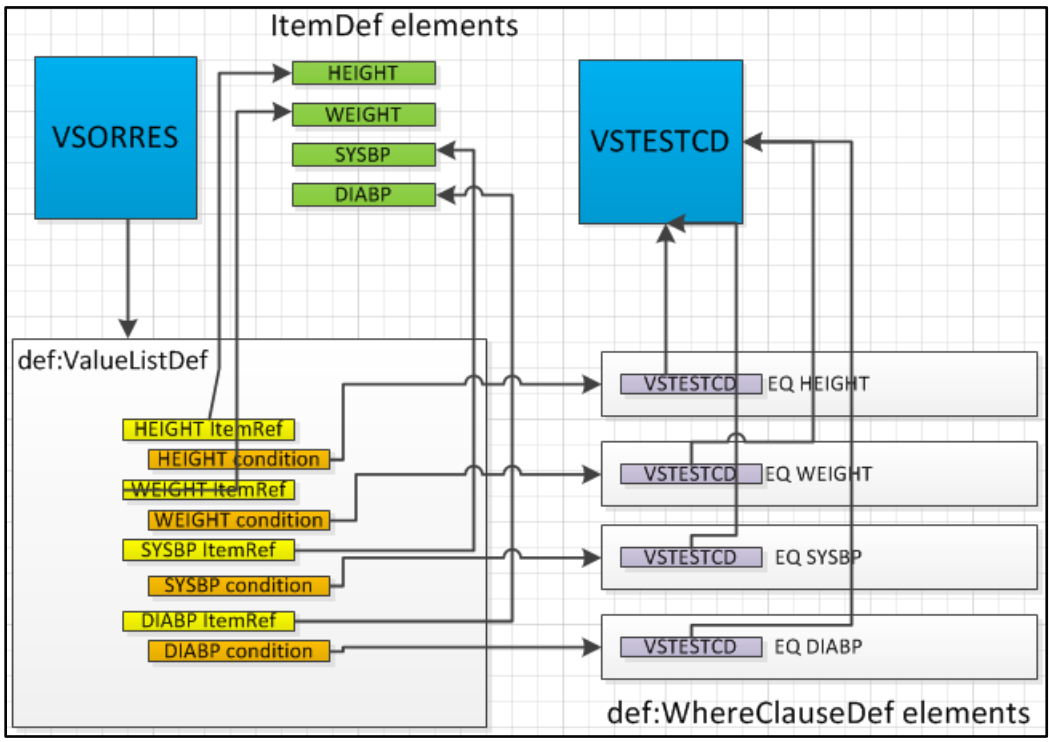


Figure 7 - Value-level metadata links

While the XML behind value-level metadata gets a little complicated, very little additional metadata needs to be collected. After all, we are creating the same Itemdef elements we created for variable-level metadata, so we can collect all of the same properties. When we collect metadata for VSORRES when VSTESTCD is set to SYSBP, the only extra metadata we need is the name of the variable upon which the metadata depends (VSTESTCD) and the value of that variable (SYSBP). The Inline approach to collection would just add these two properties to the variable-level metadata. This would extend the collection of item metadata from one-record per data set per real variable, to one record per data set per real variable per virtual variable. Extra programming would be required to generate all of the OID attributes and the multiple sections of the XML. The Database approach would keep the variable-level metadata collection the way it is, but collect the value of the OID attribute of the corresponding value list. A second form would contain the same properties, plus one each for the conditional variable and conditional value. A strict Database approach might be the collection of the OID attribute of the corresponding condition (def:WhereClauseDef) within this second form along with a third form that defines these conditions. However, this may not be necessary since each condition should only be referenced once.

CONCLUSION

Figure 8 below summarizes at a high level the relationships that exist between different sections of define.xml.

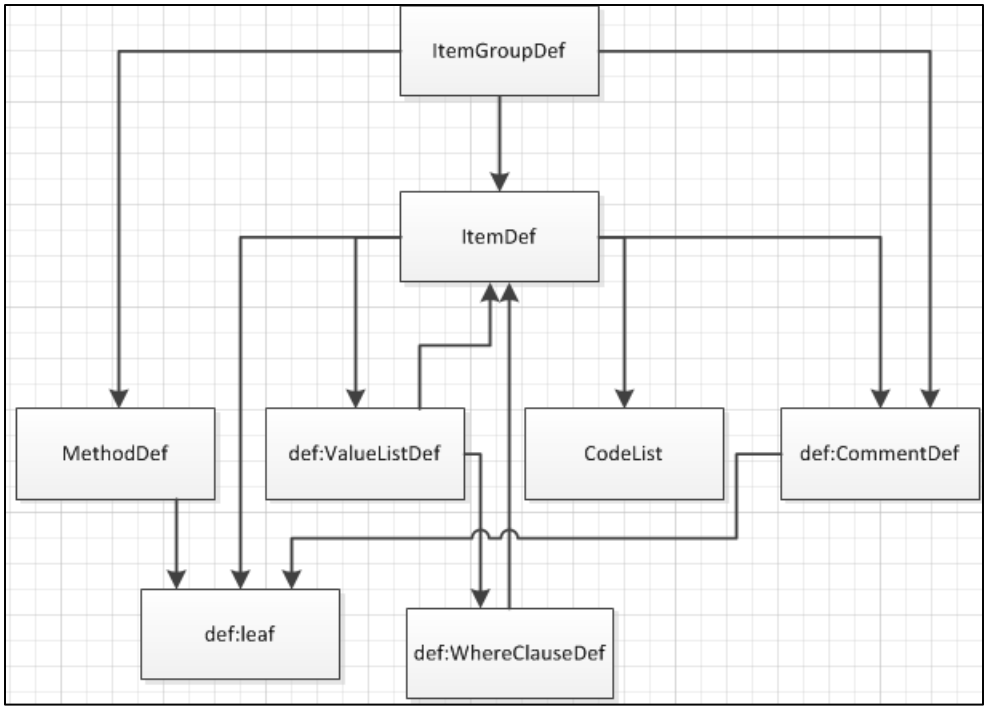


Figure 8 - High-level view of define.xml links

As we've seen throughout this paper and as summarized in Figure 8 above, the complex network of relationships between different sections of metadata presents significant challenges for collecting and processing metadata. If each box represented an isolated section of metadata and the arrows above didn't exist, then we could easily collect each section in its own form without regard to what was collected in another. But the arrows force us to make decisions. On the one extreme, if we insist on collecting defining properties side by side with their references, what we've called the Inline approach, then Figure 8 above leads us to collecting all metadata in one large form. At first this appears that while this does lead to repeated collection of the same definition, it may seem easier on the user, relieving them of having to know how to link between forms. But where does the expansion of the form structure end? We talked about expanding the collection of data set metadata from one record per data set to one record per variable per data set. We also talked about expanding this to one record per virtual variable per real variable per data set. Imagine how unwieldy this structure might get if we insisted on expanding this to allow each allowable value of a variable that is subject to controlled terminology to have its own record. On the other extreme, how reasonable is it to expect a user to know how object definitions are identified for the purposes of referencing? How practical is it?

In reality, the approach that an organization takes will end up somewhere in the middle of these two extremes. Where exactly depends on a combination of several factors.

- **Resources** - Who are the users? What is their experience level? Do they have any database background? What is their familiarity with define.xml? The same questions can be asked of the database programmer.
- **Tools** - Throughout this paper we've assumed basic tools such as Excel for the input of metadata. Whether it is Excel or something else, how much intelligence can we build into the tool? How much potential error can be reduced with the use of controls such as dropdown menus and other functionalities the tool might have? Can a tool be created that validates the final metadata product, so that gaps in the database resulting from a lack of intelligence in the collection tool can be detected and acted upon prior to the production of define.xml?
- **Regulatory and business policies, practicality, and the 80/20 rule** - The FDA wants your submission to be accompanied by a set of annotated case report forms in a PDF file called acrf.pdf. While on occasion, a study may have a second set of aCRFs, for most studies, is it necessary to collect document definitions for aCRFs? While a database approach may be more attractive (particularly to programmers), is the collection of value-level metadata made more practical by adding just two properties (conditional variable such as VSTESTCD and its value) to the variable-level metadata form and expanding its structure rather than introducing additional forms and links between them?

All of these factors are centered around the detailed ways in which collection forms can be designed to make easier the jobs of the user and the programmers. In addition to form design, other factors should also be considered.

- **Timing** - At what points in time do certain properties have to be collected. At what points do they first become available? Does it make sense to have a process in which different metadata is defined at different points in time? Perhaps by different people?
- **Alternate collection methods** - How else can we collect metadata other than asking a user to manually enter it?
    - **Standards** - How much of the metadata we've talked about can be standardized? We've talked about the standardization of document definitions. Metadata that belongs in the header of define.xml doesn't change much and may be a candidate for standardization. We might even be able to standardize a set of data set-, variable-, and value-level definitions for a specific therapeutic area. To the extent that this is possible, which properties can be standardized and therefore pre-populated. Among these, which can be modified for a study, and which must remain static and therefore not have to be collected? While this may still involve manual collection, it may be collected by different, non-study resources such as standards and/or therapeutic area experts.
    - **Derived** - Recently the FDA has been asking for economically-sized data sets, which includes character variable lengths that are only as large as their largest values. This means that the Length attribute in ItemDef needs to be data-driven. What other metadata can be programmatically derived? DataType? Others?

XML files are nothing more than text that any SAS programmer with even a basic understanding of the DATA step should be able to create. The challenge in creating define.xml is in the process. How does the metadata reliably get from places like standards, from data, or from our own heads into a relational database? Several solutions are possible that don't necessarily require additional software resources, any more than producing tables, listings and figures requires these. But producing define.xml does require knowledge of define.xml, some database knowledge, and maybe most importantly, well thought-out planning.

CONTACT INFORMATION

Your comments and questions are welcomed. Please contact me in any of the following ways:

Mike Molter
d-Wise
1500 Perimeter Park Drive, Suite 150
Morrisville, NC 27560
919-600-6237 (office)
919-414-7736 (mobile)
mike.molter@d-wise.com
molter.mike@gmail.com