PharmaSUG 2012 - Paper HW01

An Introduction to the Clinical Standards Toolkit and Clinical Data Compliance Checking

Mike Molter, d-Wise Technologies, Raleigh, North Carolina

ABSTRACT

Since the dawn of CDISC, pharmaceutical and biotech companies as well as their vendors have tried to inject standards compliance checking into their clinical and statistical programming flows. Such efforts are not without their challenges, both from technical as well as process standpoints. With the production of data sets and tabular results taking place inside of SAS® programs, it's tempting to add code to this flow that performs these checks. While the required code can be relatively straightforward for SAS programmers with even modest programming and industry experience, all too often the management of such code and the processes around its use is where the difficulties occur. Without proper management, seemingly simple tasks such as selecting which checks to run or changing process parameters become more complicated than necessary.

The Clinical Standards Toolkit (CST) is an attempt by SAS to build a stable framework for the consistent use of BASE SAS around the process of standards compliance checking by striking the proper balance between the flexibility of BASE SAS and the needed discipline of process parameter management. In this workshop we will take a tour of the CST components and execute compliance checks under multiple circumstances set by users in these components. In the end, users should know not only how to set up programs to achieve this task, but also how to manipulate files to make this process work for their own needs.This paragraph is used for the abstract. This is the paper body. This is the paper body. This is the paper body. This is the paper body. This is the paper body.

INTRODUCTION

The Clinical Standards Toolkit (CST) is a framework based in Base SAS and built by SAS Institute that is used for executing certain processes around clinical data. Such processes include data validation as well as the production and consumption of XML standards. In this paper, focus will be centered on the validation of clinical data based on CDISC's Standard Data Tabulation Model (SDTM). The description of the framework, however, applies to all processes, and the description of the validation process applies not only to SDTM data, but also to the validation of data based on CDISC's Analysis Data Model (ADaM) and XML models (ODM and define.xml).

The free and open nature of a Base SAS environment might easily lead the reader to wonder how such processes can be executed confidently and accurately without, for example, the discipline enforced through tight controls implemented through a friendly user interface. Hence, the term, *framework*. While SAS programmers at any moment are free to type anything they want into a SAS editor, this paper will show that the strategic placement of environmental and process parameters leads the programmer toward a common program structure for all processes. Reinforcing this structure is the provision by SAS of sample programs to serve as guidance.

This paper will begin with a high-level overview of how a process is executed, including how it uses parameters. We'll then take a tour of the CST, discussing location and purpose of directories and files that SAS provides upon installation. With this background in place we'll then walk through the steps necessary to apply a validation process to a specific study. This paper is intended to serve as a supplement, and not a replacement to the User's Guide that SAS offers for the CST on the website. We will not dive into the detail of every macro that is installed, every variable of every data set that is installed, or every macro variable available to the system. The meanings of these can be found in the User's Guide. Instead, this paper will try and give the user an idea of how to get started, some of the key concepts and some of the pitfalls.

THE CST FRAMEWORK

```
%sdtm_validate
```

With the roughly 100 macros that accompany CST installation, it's the one line above that when executed, kicks off the validation process of a set of SDTM data sets. What may be most striking about this fact is that the execution of this macro makes use of no macro parameters. The lack of macro parameters leaves the user asking questions such as "how does the macro know where the SDTM data is?", or "what exactly is the macro 'checking'?". One might conclude that the only explanation is that the macro makes assumptions about the answers to all such questions. Maybe the macro assumes source data is in a directory with a specific libref. Maybe it executes one set of static checks in a particular sequence. Maybe the programming for each of the checks is hard-coded into the macro. Maybe when a particular check is violated, the macro simply writes a hard-coded message to the log for you to find

and act upon. In other words, maybe strict macro requirements such as the location of source data and what to check force the user to develop a contrived environment that meets all the demands of the macro, accept all of the checks hard-coded in the macro as necessary for each execution, thereby rendering the macro, or more generally, the CST system itself, “inflexible”.

Contrary to this intuition, the system is a highly flexible, parameterized system that gives the user the freedom in designing study-specific environments to which they are accustomed. In other words, users are still free to store not only source data, but macros, formats, and other study files wherever they choose. As we’ll see in detail later, validation checks are managed in SAS data sets in which each row represents a unique check (with a small handful of exceptions). Users have control over not only which of these checks to execute, but also which data sets and columns are within scope of the current execution, to some extent the code that is executed (without having to alter macro code), and other validation parameters. Even the messages that CST uses to indicate violations of a check are accessible to users through SAS data sets.

On the surface, the concept of a highly parameterized macro that offers no macro parameters seems paradoxical. In a flexible system that offers no macro parameters, how does the system know the answers to the questions about location of source data, validation checks, or validation check messages? The resolution to this mystery is that *process* parameters such as the location of process files are centrally located in what SAS refers to as a *Sasreferences* data set (referred to throughout this paper as SASREFERENCES, which is often used as the name of the data set). This data set contains variables such as MEMNAME and PATH whose values respectively represent the name and location of these process files. MEMNAME is null when the location is only a directory (e.g. source data). Identifier variables such as TYPE and SUBTYPE represent the purpose of the files. For most parameters, the value of these variables comes from a list of allowable values (found in the STANDARDLOOKUP data set). Text variables such as SASREF contain values used as librefs or filerefs, and REFTYPE indicates whether the value of SASREF is a libref or a fileref. Though not always necessary, the ORDER variable allows users to set precedence for multiple entries of certain TYPE values.

Valid values for TYPE and SUBTYPE are set by the CST because in most cases, macro processing requires the retrieval and processing of certain kinds of observations from SASREFERENCES in order to accomplish certain kinds of tasks. These two variables serve the purpose of identifying types of observations. When an observation represents the location of source data, the value of TYPE must be “sourcedata”, while SUBTYPE is left null. TYPE=”messages” (SUBTYPE null) represents the location and name of the data set that contains messages. All CST processes should have at least one “messages” observation that provides name and location of framework messages – messages about general framework tasks. Framework messages are distinguished from other kinds of messages in the STANDARD variable, whose value is “CST-FRAMEWORK” for such messages. Validation processes should contain a second “messages” observation that point to validation-specific messages, identified again with the STANDARD and STANDARDVERSION columns. For example, messages about the results of individual checks against SDTM data will be identified by the name of your SDTM standard (e.g. SDTM 3.1.2). Because the data set that contains the checks to be executed might be described as a *control* file (a file whose data set contains process input), the value of TYPE for this data set is “control”. Because other data sets might also be described as control data sets, SUBTYPE must be used to uniquely identify the control file. The validation data set is identified by SUBTYPE=”validation”.

Due to the nature of some of the validation checks, additional process parameters are required in SASREFERENCES for a validation process that aren’t so obvious. Certain checks use data sets whose contents represent metadata about the source data. Metadata is typically kept in two different data sets – one containing table-level metadata, the other containing column-level. Information about these data sets is entered into two observations of SASREFERENCES with a value of “sourcemetadata” for TYPE, and either “column” or “table” for SUBTYPE. Other checks compare study metadata to standard metadata – the metadata to which all studies are supposed to comply. The location of data sets that contain these standards, which are structured exactly like the study metadata, is entered into SASREFERENCES with a value of “referencemetadata” for TYPE, and again, either “column” or “table” for SUBTYPE. Macros that check compliance to controlled terminology expect this controlled terminology to be stored in SAS formats. The location of these format catalogs is entered into SASREFERENCES with a value of “fmtsearch” for TYPE. When multiple format catalogs are used, ORDER is used to specify searching precedence of format catalogs.

Finally, in addition to process-specific parameters, all SASREFERENCES data sets should have a set of observations that represent what we might call *environment* parameters. Validation processes (and other processes) should have an observation that represents a process-specific *properties* file (TYPE=”properties”). A properties file is a plain text file with name-value pairs that, through the CST_SETPROPERTIES macro, become global macro variables and values. As with messages, some properties are specific to a process, while others are considered framework properties. For reasons we’ll soon see, process-specific property files should be entered into SASREFERENCES while framework properties should be processed outside of the Sasreferences context. Additionally, SASREFERENCES should have a TYPE=”autocall” observation, which contains the location of macro

libraries. Of the approximately 100 macros that are installed with CST, about 70 are automatically part of the default SASAUTOS path. These 70 are framework-specific macros. Process- specific macros (such as SDTM_VALIDATE) are stored in different areas and are not immediately available to the user. Finally, multiple observations can be added with TYPE="results". Values of SUBTYPE such as "validationresults" and "validationmetrics" indicate different types of results. PATH and MEMNAME indicate the directory and data set to which these types of results should be stored.

Of course the mere existence of the Sasreferences data set and all the files it documents (e.g. properties files, validation control data set, etc) isn't enough for the CST to know about these files. For this reason, a SAS program that contains the call to the SDTM_VALIDATE macro must also contain code that processes (and depending on your preference for storage, even creates) the SASREFERENCES. *Processing* the data set generally means making the information contained in it accessible to the user. Through the CSTUTIL_ALLOCATESASREFERENCES macro, librefs and filerefs associated with files and directories such as those that contain source data and validation data sets are established. Additionally, for observations where TYPE="fmtsearch", CSTUTIL_ALLOCATESASREFERENCES sets the priority of the format catalog search by executing an OPTIONS statement to set the FMTSEARCH option. When TYPE="properties", CSTUTIL_ALLOCATESASREFERENCES executes the CST_SETPROPERTIES macro, thereby creating and initializing global macro variables used by other macros. When TYPE="autocall", CSTUTIL_ALLOCATESASREFERENCES adds the location to the autocall path. Users that need access to any combination of study-specific, company-wide, or any other source of macros or formats, would simply add an observation to SASREFERENCES for each location. In addition to this macro, users can also capture librefs, filerefs, and file names in macro variables through the CSTUTIL_GETSASREFERENCE macro.

The program that accomplishes all of this is known as a *driver program*. This is an important part of any CST process and worthy of additional emphasis.

> ***Every CST process is executed with a driver program that processes parameters and executes the appropriate process macro (e.g SDTM_VALIDATE).***

Though driver programs will vary slightly according to decisions made by users, every driver program should have at minimum, the following three components:

1. The initialization of framework properties through a call to the CST_SETSTANDARDPROPERTIES macro. We stated above that this should not be done through the Sasreferences processing. The reason is because among the framework properties are the name and location of the Sasreferences data set. Whereas CST_SETPROPERTIES requires users to supply the location of a property file through a macro parameter, this macro calls CST_SETPROPERTIES and automatically populates this parameter with the path it retrieves from information registered with the standard (we'll discuss this later).
2. The processing of the Sasreferences data set through a call to CSTUTIL_ALLOCATESASREFERENCES or CSTUTIL_PROCESSSETUP. As noted above, this establishes references to libraries and files, sets properties through global macro variables, sets search priorities for format catalogs, and establishes autocall libraries.
3. A call to the main process macro. In the case of SDTM validation, this is the SDTM_VALIDATE macro.

Other components of the driver program depend on some decisions made by the user. Some driver programs begin with the initialization of certain global macro variables. One such macro variable seen in examples throughout SAS documentation is StudyRootPath. This is used to store the path of the highest level directory that contains subdirectories and files specific to a study. This allows the Sasreferences data set to be that much more static by populating the PATH variable for study-specific observations with references to this macro variable. If users elect to store the Sasreferences data set in the WORK directory, then the driver program is the place to create this data set. The same can be said of other data sets the user decides belong in WORK.

Why this approach? Why not use the program editor to simply type LIBNAME and FILENAME statements, as well as OPTIONS statements that establish format search preferences and macro autocalls? Why not simply define macros like SDTM_VALIDATE with macro parameters, instead of taking the value of these macro parameters and moving them into a data set? We might think of the answers to these questions as being among the defining features of the CST *framework*. SAS's program editor is an open, free text, structure-less environment. Making changes to any process parameter usually means searching through a text file (a SAS program), making the change, and hoping that you caught all occurrences of text that needed changing. The CST framework, on the other hand, is an attempt to remove as much of this error-prone variability as possible and store it into more structured environments such as SAS data sets. It allows us to build our driver programs with more consistency, following a more predictable flow of macro execution, free of "clutter" that reflects user preferences. This separation of process execution from the parameters that feed it allows us to properly manage changing circumstances without changing core code. As mentioned earlier, the free-text Base SAS environment allows users to do anything they want, including using certain

aspects of the CST while ignoring others, but the more we ignore, the less advantage we gain from such a framework.

In addition, one of the central themes of the CST is data standards. We've been talking about its application to CDISC standards, but we can use it for any set of data standards. The notion of compliance checking assumes that something exists against which we compare our current study data. Suppose we decide, for example, that every study is going to make use of a variable X, whose length shall always have a length of 10, and whose label will always be "the letter X". On the one hand, we can write code that looks specifically for the variable X in the data set; that checks to make sure that the variable's length is 10 and label is "the letter X". But standards such as variable attributes maintained in code as hard-coded values become needles in haystacks, nearly impossible to manage and sustain when you consider all of the variables of a database, and all of the attributes and other checks that need to be executed. On the other hand, standards expressed as metadata and maintained in a database, thereby removed from the code haystack become easier to manage without having to touch core code. Throughout this paper we'll see how the CST manages data standards.

At this point it's clear that CST processes heavily rely on external files. Upon installation, SAS gives us access to dozens of files, some of which are meant to be customized, some not. At this point we'll now take a tour of the CST system, noting different locations where these files are installed, and where necessary, analyzing in detail the structure of these files.

SAS-INSTALLED FILES

As mentioned earlier, CST comes installed with over 100 macros, but not all are installed in the same place. Approximately 70 of these are known as *framework macros*, and are stored in !sasroot/cstframework/sasmacro, where !sasroot is the core folder for your SAS installation. Framework macros are intentionally buried in a remote location like this, which, depending on your network setup, may not be writeable, because they serve as the backbone of the CST and are not meant to be modified.

Upon installation the framework macros are automatically added to SASAUTOS, your default autocall library, making them accessible immediately. One quick way to test your access to these macros is to execute CSTUTIL_SETCSTGROOT. Depending on the macro options you have set, you won't see much in your log, but if it is accessible, you won't see errors about not finding the macro.

Of course any of the framework macros would have sufficed for this test, but CSTUTIL_SETCSTGROOT was chosen for its brevity and simplicity. The only purpose of this macro is to initialize the global macro variable _cstGRoot. Executing `%put &_cstGRoot ;` after the call to this macro uncovers a directory path that was chosen by the user at installation. This path represents the root of most other files installed with CST. Several framework macros execute this macro in order to have this path available. The path is also often referenced in control data sets such as SASREFERENCES, and is referenced frequently in documentation. Throughout the rest of this paper, this path will be referenced the same way it's referenced in code - &_cstgroot. We're now ready to see what files and directories SAS installs in this area.

Four directories are installed within &_cstgroot – Metadata, Schema-repository, Standards, and Xsl-repository. Schema-repository and Xsl-repository are storage areas for XML schemas and XSL stylesheets respectively, and there usually isn't much reason to change any of those files. We'll spend most of our time in the Metadata and Standards directories, where each of the files falls into one of two categories.

FILES RELATED TO THE REGISTRATION OF STANDARDS

The first of these categories represents files related to the registration of standards. For a validation process to work, a set of data standards must be *registered* with the CST. Technically, all that means is that information about the standard must be present in the two data sets – STANDARDS and STANDARDSASREFERENCES – in &_cstgroot/metadata; and that the standards must be manifested through metadata in &_cstgroot/standards/*standard-directory*/metadata. CST is installed with several data standards registered, including an SDTM 3.1.2 standard and an ADaM 2.1 standard. The STANDARDS data set contains one record per registered standard. The main information in this data set is the location of the root folder for this standard – normally in &_cstgroot/standards (e.g. &_cstgroot/standards/cdisc-sdtm-3.1.2-1.4). The name of the other data set in &_cstgroot/metadata indicates its relationship to the Sasreferences data set discussed earlier. Both data sets have the exact same variables. STANDARDSASREFERENCES contains for each registered standard, one observation for each file or directory related to that standard. SASREFERENCES, on the other hand, is specific to a process. For example, SASREFERENCES for an SDTM validation process will contain many of the observations from STANDARDSASREFERENCES related to the SDTM standard, but may also add records related to other standards, such as a TYPE="messages" record where the STANDARD="CST-FRAMEWORK", or a TYPE="fmtsearch" record associated with a terminology standard. Also, unlike SASREFERENCES, STANDARDSASREFERENCES has no study-specific records for the standards registered at installation.

Moving out of &_cstgroot/metadata and into &_cstgroot/standards, we find one subdirectory for each standard installed by SAS, including cdisc-sdtm-3.1.2-1.4. Within each of these is the Control subdirectory. Control contains two data sets that correspond to the two in &_cstgroot/metadata. The content is the standard-specific information that ends up in &_cstgroot/metadata upon registration of the standard (without the path, which is specified through a macro parameter). Since these standards are registered upon installation, these data sets as installed serve no further purpose.

REFERENCE METADATA

Also installed in &_cstgroot/standards/cdisc-sdtm-3.1.2-1.4 is a subdirectory called Metadata that contains the data sets REFERENCE_TABLES and REFERENCE_COLUMNS. We'll sometimes refer to these kinds of files as *reference metadata* files. They are identified in a SASREFERENCES data set with TYPE="referencemetadata". As indicated earlier, the purpose of such data sets is to provide standards through metadata. It's important to note that these two specific data sets provided by SAS are meant only to serve as a starting point for your standard, but not necessarily to be used as your standard. The content of these data sets is based on the model laid out in the SDTM 3.1.2 Implementation Guideline. In other words, REFERENCE_TABLES has every data set modeled in the IG, and REFERENCE_COLUMNS has every column. For several reasons, users should consider building their own reference metadata based on their own standards. One reason is that the IG contains nothing about efficacy data sets. Assuming your studies require efficacy analyses, and therefore your SDTM database will require custom domains, your reference metadata must account for these domains. A second reason is that your reference metadata should only have metadata for the domains and variables you plan to use, which most likely does not consist of every safety domain and every variable within those domains defined by the IG. Along those lines, a third reason is that these data sets carry metadata for a data set called SUPPQUAL. In 3.1.2 sponsors are submitting multiple domain-specific Suppqual data sets rather than combining them all into one data set.

Although reference metadata is used as the basis for comparison for compliance checking, not all of its columns are used in this way. We'll discuss in detail later the study-specific counterpart to reference metadata that reflects an instance of a study – *source metadata*. As with reference metadata, source metadata is split in two data sets - one for table-level and one for column-level metadata – that we will refer to in this paper as SOURCE_TABLES and SOURCE_COLUMNS. These two data sets are structured exactly the same as their reference metadata counterparts. Column-level columns like COLUMN, LABEL, TYPE, and LENGTH are used by certain checks to make sure that the source metadata reflects the standard. Other columns, on the other hand, are not used for comparison, but rather as a guide in reference metadata for populating source metadata. For example, both table- and column-level metadata contain columns that are meant to feed define.xml. Examples include the variables XMLPATH at the table level and ROLE and ORIGIN at the column level. Standards developers may choose to pre-populate XMLPATH in REFERENCE_TABLES with a typical path to the transport file, relative the location of define.xml. Note that the REFERENCE_TABLES installed by SAS pre-populates this variable with "../transport/*domain-name*.xpt". The REFERENCE_TABLES you build for your standard should pre-populate according to your directory structure. ROLE in the REFERENCE_COLUMNS installed by SAS reflects the role assigned by CDISC in the IG and should not change. ORIGIN comes unpopulated but you may choose to pre-populate it in your standards. Other variables such as KEYS (data set primary keys) at the table level and QUALIFIERS (indicate whether variable values should be uppercased) at the column level aren't used for comparison but are used in certain checks. Variables like these that are not used for comparison can be copied from the reference metadata to the source metadata to provide a starting point for that kind of study-specific metadata. We'll see later how the production of study-specific metadata can begin with a copy of these columns from reference metadata combined with PROC CONTENTS metadata that reflects the actual SDTM database (which will be compared to reference metadata).

The second category of files might be described as "tools". We've now seen that files that contain information about standards are contained in &_cstgroot/metadata. We've also seen that contained within &_cstgroot/standards is one directory for each registered standard. Each of these standard directories contains a Metadata directory that contains reference metadata. In addition, a standard directory will contain other subdirectories to contain these tools. The term "tools" is being used here to describe any files that somehow aid in a process (e.g. SAS macros, driver programs, control data sets, etc.). Let's now look at some of these tools.

Although this paper is focusing on the validation process, every process has to use certain framework tools. We saw earlier that framework macros are stored in a separate location to protect them from modification, but SAS does install in &_cstgroot/Standards a directory for framework tools (cst-framework-1.4). Because of this, some might even refer to the framework as a standard, but that's a matter of semantics that we don't need to get into in this paper.

MESSAGES

Both the framework directory and the standard directory contain a Messages directory, each containing a MESSAGES data set. The structure for the two is the same. Each contains message identifiers (RESULTID) and

message text. Messages can be parameterized where parameter values are found in PARAMETER1 and PARAMETER2. These parameter values can be overridden by macro parameters of macros that process these data sets. Both have columns that contain information about the source of the message but this is only meaningful in validation messages where the source of the validation check is an external source (e.g. WebSDM).

It's important to note the difference between these two data sets. The SDTM messages are messages about specific validations checks and how they were violated. These violations are violations about the SDTM data itself. The checks that SAS gives us with installation represent SAS interpretations of the SDTM IG. A record in SASREFERENCES with TYPE="messasges" and STANDARD set to the name of your SDTM standard must be included in order to understand the violations that the validation process finds. Framework messages are meant to document the results of certain processes, including anything that went wrong, outside the context of a specific standard. This includes any requirements for a process to run, such as a properly structured and populated Sasreferences data set. SASREFERENCES must contain a second TYPE="messages" record where STANDARD="CST-FRAMEWORK" in order to understand certain process steps and errors.

Unlike the reference metadata that SAS gives us to use as a basis for building our own standards, the MESSAGES data sets installed with CST require little, if any modification. Users might choose to change the wording of a particular message if they feel the default wording is misleading or difficult to understand, or they may choose to change parameter values. They may also choose to elevate certain violations, for example, from warning to error, if violation of such issues is of a particular significance in the organization. Users that want to add validation checks will want to add corresponding messages to the SDTM messages data set with a RESULTID that matches the identifier variable value of the check (which we'll see when we discuss the validation control data set).

PROPERTIES

Both the framework and the standard directories also come with a subdirectory called "programs". These include property files, which are simple text files with a single column of name-value pairs. The meaning of each of these properties is documented in the CST User's Guide. As with messages, SDTM properties are specific to the validation process. Access to these properties through global macro variables is gained through the execution of CST_SETPROPERTIES where the path and filename of the property file is required through macro parameters, or CST_SETSTANDARDPROPERTIES, where the macro gets path and file name information from STANDARDSASREFERENCES, using the name of the standard provided through a macro parameter. Optionally, users can add a record to SASREFERENCES, where TYPE="properties". For this process, because CST contains two different property files, SUBTYPE must be set to "initialize" or "validation", or have one record for each. Execution of CSTUTIL_ALLOCATESASREFERENCES will call on CST_SETPROPERTIES with the path and file name information it contains. Among the properties in the framework property file installed by SAS are those that represent the name and location of the Sasreferences file. For that reason, these macro variables must be set before the Sasreferences data set is processed, since CSTUTIL_ALLOCATESASREFERENCES otherwise won't know where the Sasreferences data set is. For this reason, this file is often processed at the beginning of a driver program using CST_SETSTANDARDPROPERTIES.

Property files provided by SAS also can usually be left alone, although a user may want to do some research on their meanings and change some values. For example, if users want temporary data sets to be kept after process execution, they may choose to set the _CSTDEBUG property to 1. Users may also decide to store SASREFERENCES somewhere other than the default WORK location.

MACROS

Also installed in the standard directory are the Macros subdirectories. Macros contains SAS macros relevant to the standard, including SDTM_VALIDATE. These get added to the autocall path only by their inclusion in SASREFERENCES with TYPE="autocall" and the execution of CSTUTIL_ALLOCATESASREFERENCES. For the most part, these should require no modification, but users should get to know these before using them. We'll see later that SDTMUTIL_CREATESRCMETAFROMSASLIB is a macro that was actually written as a shell as is meant to be modified.

VALIDATION CONTROL

Directories that represent standards that support validation such as the SAS-installed SDTM standard contain a Validation directory, which in turn contains a Control directory. Contained in here is the VALIDATION_MASTER data set, or what we will describe as a master validation control data set. Each observation represents one validation check, and each observation from a run-time control data set (a subset of the master control data set, discussed later) is processed by SDTM_VALIDATE one at a time. The variables represent a mix of data that combines information for the user and data that is used by the validation process. Informational variables include CHECKSOURCE and SOURCEID which give users an idea of from where the check originated. CHECKTYPE is a useful guide for how to group checks together for execution. Values such as "Metadata", "Date", and "MultiRecord" give an idea of what is being checked. CHECKSTATUS is used to indicate whether or not the check is ready for

execution. Upon installation, positive values mean that the check is active. A value of 0 means inactive, but users can change them to active when appropriate. For example, check SDTM0450 is initially set to 0, but when the user has a data set that contains dictionary terms against which a variable's values can be compared, then the user can set this to Active. Variables used by SDTM_VALIDATE include CODESOURCE, TABLESCOPE, COLUMNSCOPE, CODELOGIC, LOOKUPTYPE, and LOOKUPSOURCE. CODESOURCE names a macro to be called as part of the process. TABLESCOPE and COLUMNSCOPE use a convention to tell the macro which data sets and columns to check. The value of CODESOURCE is valid SAS code used only by certain macros. LOOKUPTYPE and LOOKUPSOURCE provide information about where to find controlled terminology lists against which to check data. What this data set does not contain is a variable that gives the user an idea of what is being checked. Messages about the meaning of checks are found in the Messages data set for the same standard. A useful exercise is to merge these two data sets by the check identifier.

For those getting started with validation checks through the CST, this data set should need no modification. Users might discover after a while that changing values of certain informational variables, such as CHECKSTATUS as discussed above, or elevating the severity of a check from "Warning" to "Error" through the CHECKSEVERITY variable is more useful to their needs. Only the advanced user that has become intimately familiar with how the validation process and each of its components works should make attempts at changing values of variables that the process uses. For example, making changes to the SAS code that serves as the value of the CODELOGIC variable requires an in-depth knowledge of exactly where in the process this code is executed, as well as the names of temporary data sets available to it and what is expected to come out of it. Adding custom validation checks means adding new observations to the master data set. Populating each of these variables for the new check also requires a detailed knowledge of where each fits into the process.

This wraps up our discussion of the files that SAS gives to us, some that are necessary for any data standards we wish to implement, some to get us started down the road of building our own standards. In summary, most of these files can be used as installed with little to no modification. Properties can be adjusted to suit user preferences. Once users feel comfortable with the way the system works with the files, customization of checks to meet an organization's needs can be implemented through changes to the validation checks and Messages data sets. The two files that will almost certainly need modification to implement custom data standards are the two reference metadata files.

Before we move on to the next section, let's be clear about what is meant by "modification" of files. In effect, what SAS has installed for us is a set of *sample* data standards (e.g. SDTM, ADaM). We can say this because while many of the tools that come with a standard can be used across standards, the reference metadata, which is the defining feature of the standard, cannot. The reference metadata that SAS has installed serves as a starting point, a guide for our own reference metadata, based on everything defined in CDISC implementation guides. Modification could mean the direct modification of the files SAS has given us. However, in order to preserve the original files and use them as a starting point for all future custom standards, a better practice would be to make modifications to copies of these files, to serve as a new standard.

USING CST

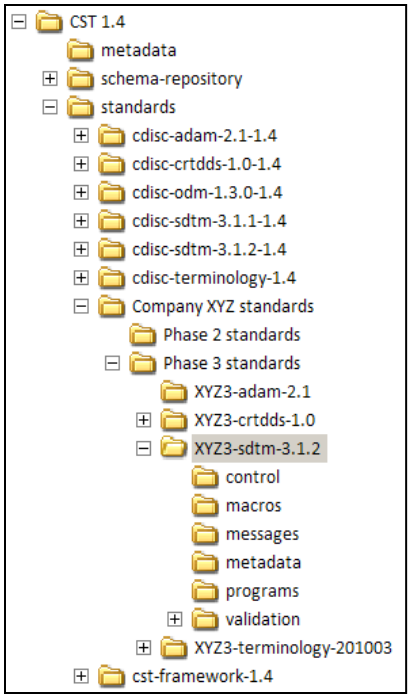
For the rest of this paper, in order to demonstrate the steps required to use CST for the purpose of validating SDTM data, let's imagine that we work for a pharmaceutical company called XYZ. XYZ runs both phase 2 and phase 3 trials, and maintains data in SDTM and ADaM formats. They also maintain controlled terminology and produce define.xml. For SDTM, each of the phases consistently produces a subset of the safety domains documented in the SDTM 3.1.2 IG, a certain set of custom efficacy domains, and a handful of SUPPQUAL data sets. While the set of data sets and variables is consistent within a phase, they are not the same between the two phases. XYZ would like to be able to execute compliance checks on phase 2 or phase 3 SDTM data according to the appropriate standards. These differing standards will be manifested through custom reference metadata.

Note: In reality, an organization, depending on its size and the kinds of trials it manages, may require multiple sets of standards, such as one for each therapeutic area.

Before we can ever begin to use the CST, we have to make sure that all of the setup files that we discussed above are in place and customized to our organization's needs. For this exercise, we'll make the reasonable assumption that all of the framework tools (properties and messages), as well as all of the SDTM properties, messages, macros, and the master validation data set installed by SAS can be used for both standards. Only the reference metadata needs customizing, as well as the information needed to register the standards.

REGISTERING A STANDARD

Regardless of how much customization is required, the standards files that SAS gives to us at installation are always a good place to start to build our standard. For that reason, a good first step is to create a directory somewhere within the installed Standards directory for each standard you're creating. Within each new standard directory, we'll copy and paste all of the SAS-installed directories and included files from within cdisc-sdtm-3.1.2-1.4.



Display 1: Directory Structure for Installed and Custom Standards

In Display 1, we see that alongside each of the standards folders that SAS has installed for us, we have created one folder dedicated to all of XYZ's standards, and then inside divided them between phase 2 and phase 3. Because we want not only SDTM, but also ADaM and other standards, we have an additional layer of subdirectories. Underneath XYZ3-sdtm-3.1.2 are copies of the directories and files that SAS installed in cdisc-sdtm-3.1.2-1.4 that we have described above. The user is now ready to make changes to the reference metadata data sets stored in XYZ3-sdtm-3.1.2/metadata (as well as the corresponding phase 2 data sets).

In addition to changing the standard metadata, in preparation for registering the new standard, changes will have to be made to the two data sets in XYZ3-sdtm-3.1.2/control. The registration process adds the contents of these two data sets to their counterparts in &_cstgroot/metadata. Having copied these from the SAS-installed cdisc-sdtm-3.1.2-1.4 directory, the value of STANDARD and STANDARDVERSION still reflects that of the SAS-installed SDTM standard. At minimum, prior to registration, the user must change these values to reflect the name and version of the standard they are creating. In STANDARDS, this should be all that's needed to change. Remember that STANDARDSASREFERENCES contains paths and filenames within the standard directory. Since we are keeping those same directory structures, no changes are needed in this copy either.

With all of the files within XYZ3-sdtm-3.1.2 customized, we're now ready to register our new standard. Any program that registers a standard must have two main components: a call to CST_SETSTANDARDPROPERTIES to initialize framework properties, and a call to CST_REGISTERSTANDARD to register the standard.

```
%cst_setstandardproperties(
_cststandard=CST-FRAMEWORK,
_cstsubtype=initialize)
```

This macro reaches into the STANDARDSASREFERENCES data set in &_cstgroot/metadata and finds the observation uniquely identified by the macro parameter values through the values of STANDARD and SUBTYPE. From this record it knows from the PATH and MEMNAME variables which properties file to process.

```
%cst_registerstandard(
_cstrootpath=%nrstr(&_cstgroot/standards/Company XYZ standard/Phase 3 standards/
   XYZ3-sdtm-3.1.2,
_cstcontrolsubpath=control,
_cststddsname=standards,
```

```
   _cstStdSASRefsDSName=standardsasreferences)
```

This macro takes the one observation from the data set whose name matches that of the third parameter, found in the directory formed by the combination of the first two parameters, populates the ROOTPATH variable with the value of the first parameter, and adds the observation to STANDARDS in &_cstgroot/metadata. Similarly, it takes all of the observations from the data set whose name matches that of the fourth parameter (also found in the directory named for the combination of the first two), adds the value of PATH to the value of the first parameter to form the new value of PATH, and again, adds the result to STANDARDSASREFERENCES in &_cstgroot/metadata.

SAS provides us with a sample program called Registerstandards.sas, located in cdisc-sdtm-3.1.2-1.4/programs that accomplishes this task. This program goes to some effort to first find out if the standard is already registered, and only registers it if a previous version of the standard does not exist.

PREPARING A STUDY

Now that the standard is in place, we're ready to begin working with specific studies. Although CST comes with samples of study validation processes as we'll soon see, unlike with the standards files, it doesn't recommend any one directory structure. This is to say that the CST is flexible enough to work with whatever directory structure you have in place, as long as you record these in the SASREFERENCES data set. Of course, whatever that structure is, certain checks require certain files to be in place, so before we start looking at the checks themselves, let's take a look at what these files are.

SDTM DATA

Maybe the most obvious requirement is the need for data, or at least data sets. Certain checks that look for violations in metadata (i.e. attributes) only require data sets with the appropriate variables, even if they have no observations. Most checks though are checking for compliance in the content, and best practices might say that it's best to run all the checks after zero-observation data sets have been populated.

SAS doesn't provide any guidance on how to use CST to create the data sets themselves. Certainly because of the different ways data is collected and the different interpretations of the SDTM guidelines, it would be difficult to provide a driver program that could accomplish this in all or even most cases. However, even if CST can't help us populate data sets with data, we can use the reference metadata to build zero-observation domains that are compliant at least in structure. Remember that structural compliance of a database in the CST is defined by the matching of its attributes to the reference metadata. By populating such domains with data, we should have structural compliance simply because of the defining source. The following assumes that "refmeta" has been established as a libref for location of reference metadata and that "srcdata" has been established as a libref for the location of the SDTM data. The SQL step joins the data set label with the column-level metadata. The DATA step then reads the result and through CALL EXECUTE, generates code that builds each zero-observation data set using the ATTRIB statement to set labels, lengths, and types.

```
   proc sql ;
       create table meta as
       select a.table as dsname, a.label as dslabel,b.column,b.label,
b.order,b.type,b.length
       from refmeta.reference_tables a,refmeta.reference_columns b
       where a.table=b.table
       order by a.table,b.order ;
       quit;

   data _null_ ;
       set meta ;
       by dsname ;
       if first.dsname then call execute ('data
srcdata.'||compress(dsname)||'(label="'||trim(left(dslabel))||'");') ;
       call execute('attrib '||compress(column)||' label="'||trim(left(label))||'"
length=');
       if type eq 'C' then call execute('$');
       call execute(compress(put(length,best.))||';') ;
       if last.dsname then call execute(' stop; run; ') ;
       run;
```

FORMATS

Most checks that evaluate the compliance of data values to controlled terminology standards, do so by default by looking for certain formats that are developed in just the right way. Knowing which format against which to compare depends on the check. Check SDTM0221 scans the SOURCE_COLUMNS data set (study-specific column level

metadata) for all variables (rows) where XMLCODELIST is populated. It assumes that this variable is populated with the name of a format. The validation master control data set also contains several checks, each of which checks one variable against one format, the name of which is found in the LOOKUPSOURCE variable when LOOKUPTYPE="FORMAT". In both cases, the macro, once it finds the format, turns it into a data set (using the CNTLOUT option on a PROC FORMAT statement). The comparison is then made between data values and the values of the *LABEL* variable of the CNTLOUT data set. Recall that it's the values on the *right* side of the equal signs in the VALUE statement of a PROC FORMAT that define this variable. In other words, make sure that for every variable subject to controlled terminology according either to XMLCODELIST or LOOKUPSOURCE, a format is created that is named for the value of one of those variables, and that no matter what is on the left side of the equal signs in the VALUE statement, the controlled term list to which the variable is subject is on the right.

From a study perspective, the development of this controlled terminology will require some concentrated effort. SAS gives us in the controlled terminology standard data sets that represent all of the controlled terminology published by CDISC. They also turn these into formats according to the method described in the last paragraph. For most studies, if not all, this exhaustive list of terms will not be relevant. Some lists won't be needed. Others will be needed with some modification, either by extending extensible lists, or removing irrelevant terms. For this reason, it's a good idea in each study to develop a study-specific controlled terminology library (format catalog), stored somewhere in the study area. This catalog would contain modifications of CDISC lists, using the same name, as well as new lists. The SASREFERENCES data set would then contain TYPE="fmtsearch" records pointing to each catalog, but assigning a value of 1 to the ORDER variable on the record pointing to the study catalog and 2 for the record pointing to the SAS-installed catalog.

EXTERNAL DICTIONARIES

Some variables are subject to controlled term lists defined by external organizations such as Meddra. Such lists by default are maintained not in format catalogs, but in data sets. Check SDTM0451, by default, is looking for a data set named "meddra" and comparing the values of AEDECOD to the values of a variable in this data set called PT_NAME. Obviously this data set with this variable must exist for this check to work. Check SDTM0450 is a more general version. For this check to work, a user must enter the name of the data set that contains the dictionary (e.g. WHODRUG) into the value of the LOOKUPSOURCE variable, and also enter the name of the variable %let statement in the CODELOGIC variable.

METADATA

Checks that evaluate the compliance of metadata do so by using data sets that contain this metadata – source metadata. The structure of the source metadata must match the structure of its reference metadata counterparts. Keep in mind though that some of the variables of these data sets are only used for define.xml purposes and not for compliance checking. Although all variables must be present, in this paper we'll only concern ourselves with variables used for compliance checking. Unlike the case with study data or study-specific format catalogs, SAS does provide us with some guidance on using the CST to build this metadata through the use of a driver program and a macro stored in the SDTM standard macros directory. However, users should use this guidance with caution for several reasons. The driver program is written to accomplish the task of creating source metadata using either of two very different sources – the SDTM itself (the source we are using here), or from the define.xml. The user must make this selection through one of several global macro variables initialized at the beginning of the program. The program then goes through some typical driver program tasks such as executing CST_SETSTANDARDPROPERTIES and creating a SASREFERENCES data set. At the end, the program executes CSTUTIL_GETSASREFERENCE several times. Recall that this macro creates macro variables out of chosen values of SASREF and MEMNAME from SASREFERENCES. After this is a single reference to a macro variable which resolves to a macro call, based on the method chosen for creating the metadata. The purpose of the macro variables created by CSTUTIL_GETSASREFERENCES isn't clear at this point until one starts to dive into either of the macros called.

Inside the macro definition (SDTMUTIL_CREATESRCMETAFROMSASLIB) are several comments that warn you of hard-coding, and the fact that the source metadata being created is based on assumptions, and that the user should take some time to analyze the results and decide if any post-execution modification is necessary. For the most part, the macro wants to merge metadata that comes from PROC CONTENTS (e.g. data set name and label, variable name, label, length, type, order) with the metadata reported in the reference metadata that you can't get from PROC CONTENTS, including define.xml metadata. Most of the assumptions built into the macro are based on how to guess at some of this metadata when the metadata is for some reason not found in the reference metadata. For example, data set keys, in this situation, are extracted from the sort order when the data set is sorted according to PROC CONTENTS. When it isn't, the keys are assigned based on the domain class.

While STANDARD and STANDARDVERSION must be populated at both the table and column level for processing purposes, the only metadata at the table level that doesn't come from PROC CONTENTS that's used in compliance checking is KEYS. The only such metadata at the column level is XMLCODELIST (discussed above), CORE, and QUALIFIERS. Because these pieces of metadata don't come from PROC CONTENTS, the best we can do at the

study level is to pre-populate them with their standard-level values (from reference metadata), and then go back and manually review them to see if they apply for the current study. This is inevitable no matter which method you use to create your source metadata. Users need to make the decision whether they want to use the guidance provided by SAS or develop something on their own. Another option is to do something in between. We've stated our case for the benefits of using the CST framework as much as possible. Users might choose to stay with the driver program approach. The macro contains hard-coding that is provided only as a placeholder so it's expected to be modified anyway. The user might take this opportunity to customize the macro. For example, the hard-coded values might instead be solicited through macro parameters. Users might also decide that upon discovery that a data set was not found in the reference data, a message should be delivered to the user rather and perhaps processing should be stopped, rather than trying to guess at what the metadata should be.

THE DRIVER PROGRAM AND THE RUN-TIME VALIDATION CONTROL DATA SET

The final piece of the study pie is the driver program. We discussed the basic structure of this file earlier so we won't repeat the details, but let's at least fill in some gaps. We know that the minimum components of this structure are the call to CST_SETSTANDARDPROPERTIES, to CSTUTIL_ALLOCATESASREFERENCES, and to SDTM_VALIDATE. We also mentioned that any data sets that we decided would be stored in the WORK directory would need to be created in this driver program. Though not required, in our example and the SAS examples, we have decided that SASREFERENCES was one such data set. We now know that some of the records in SASREFERENCES must point to study areas (e.g. study data, study formats, study metadata). We can insert these study paths directly into the SASREFERENCES data set, or, as SAS examples illustrate, we can instead insert a macro variable reference in their places and initialize the macro variable at the beginning of the program. SAS demonstrates the use of two such macro variables: STUDYROOTPATH and STUDYOUTPUTPATH, representing the top level of the study area and the directory within the study area would output such as results data sets would be saved, respectively. Either way, it is this requirement of SASREFERENCES that makes the driver program a study-specific data set. Programmers who like to separate the study-specific code from everything else might choose to store only the initialization of the two study-specific macro variables in the driver program, followed by a %INCLUDE to a more centrally located program that takes care of the rest of the validation tasks.

The record in SASREFERENCES that points to the run-time validation control data set must have TYPE set to "control" and SUBTYPE set to "validation". For several reasons that include execution time, computing resources, and availability of log space, SAS highly recommends not executing all the checks available at one time. Thus, the "run-time" description refers to a subset of the master validation data set, optimized for these purposes. Although not required and contrary to the examples SAS gives us, one option for storing this data set is in WORK. Because we can't execute them all at once, we will most likely be executing them in groups, one after the other. If we decide that we will always run them in the same groups, then we might decide to store each of the groups in their own permanent data sets. However, storing them in WORK gives the user the freedom to execute them in groups that are convenient at the time. For example, suppose that a user executes 10 checks at once, and one of them fails. After fixing the problem, the user may just want to run that one check by itself, rather than with the nine others that have already succeeded.

EXECUTING VALIDATION CHECKS

Since we are putting the run-time control data set into WORK, the code used to create it belongs in the driver program. At minimum, this should include a DATA step (or PROC SQL) that reads the master validation data set and subsets. Keep in mind that SASREFERENCES has information on the name and location (in this case, WORK) of where SDTM_VALIDATE should expect to find the run-time data set, so the data set created in our DATA step must comply. One can easily look at their SASREFERENCES data set, see what the libref and data set name are, and type the DATA step accordingly. However, the following code is more in the CST spirit.

```
%cstutil_getsasreference(
_cststandard=XYZ3 SDTM,
_cststandardversion=3.1.2,
_cstsasreftype=control,
_cstsasrefsubtype=validation,
_cstsasrefsasref=vcref,
_cstsasrefmember=vcmember) ;

data &vcref..&vcmember ;
…
```

The call to CSTUTIL_GETSASREFERENCES creates a macro variable called VCREF (fifth parameter) and assigns it the value of the SASREF variable in SASREFERENCES for the observation on which STANDARD="XYZ3 SDTM" (first parameter), STANDARDVERSION="3.1.2" (second parameter), TYPE="control" (third parameter) and

SUBTYPE="validation" (fourth parameter). It also creates a macro variable called VCMEMBER (sixth parameter) whose value is set to the value of MEMNAME from the same observation. These values are then used in the DATA statement.

The SET statement will need to read the master validation data set, but of course in doing so, it will need a libref and data set name too. Although it's not used in the process, this information can be entered into SASREFERENCES with TYPE="referencecontrol" and SUBTYPE="validation". CSTUTIL_GETSASREFERENCES can again be called to pass the libref and data set name into macro variables, to be referenced in the SET statement.

In addition to subsetting, another useful way to limit the scope of a run-time control data set is by changing TABLESCOPE and COLUMNSCOPE. For example, check SDTM0221 checks the values of the all the variables in which XMLCODELIST is populated against their corresponding codelists. Let's suppose that after having executed this check, one variable was in violation. After making the necessary fixes, you only want to execute the same check against the one variable that was in violation. The following example illustrates this with the AESEV variable from the AE domain.

```
data &vcref..&vcmember ;
set &mcref..&mcmember ;
where checked eq "SDTM0221" ;
tablescope="AE" ;
columnscope="AESEV" ;
run;
```

RUN-TIME EXECUTION SUBSETS

With over 200 observations in the SAS-installed master validation data set, how does one decide where to start, and how to subset the checks? How does the user decide which checks are even worthy of executing? For starters, we can use the variables provided in the data set as a guide.

The CHECKSTATUS might be the first variable to look at. It makes sense to immediately eliminate any checks that aren't active. This includes inactive, deprecated, and not-yet-implemented checks. Depending on your circumstances, you may only need to be concerned about a certain level of severity, which would require a filter on CHECKSEVERITY. CHECKTYPE provides a more logical grouping based on what is being checked. With nine groupings, the number of checks in each group on average becomes more manageable.

Most likely, the use of these SAS-supplied variables will serve only as a beginning for a subsetting strategy. A thorough examination of the code that is executed, organizational decisions regarding the significance or lack thereof of certain checks, careful thought about a logical flow into which the checks can be organized, along with testing and experience will, over time, help your organization develop a natural process for handling these checks. We'll conclude this paper with some factors to consider beyond the variables in the master control data set when starting down this road.

For starters, users should read through the checks, their messages, and in some cases, maybe even their code in an effort to decide whether or not the check is even worthy of executing. For example, many of the checks whose CHECKSEVERITY is "Note" may be checking issues that aren't of concern to you. Examples include check SDTM0202 which reports on expected variables that have null values. SDTM permits null value in Expected variables. Check SDTM0005 checks for custom data sets to make sure their names start with "X", "Y", or "Z" - a CDISC suggestion, but not a requirement. Check SDTM 0014 makes sure permissible variables are present - again, not a CDISC requirement.

In some cases, what a particular check is trying to uncover overlaps with the intentions of another check. SDTM0607 checks to make sure that SITEID isn't always null, a check that is also accomplished SDTM0605 which makes sure all variables are populated on at least one observation. Perhaps the most overlap occurs in the CNTLTERM category. Check SDTM0221 makes sure that all variables whose metadata variable XMLCODELIST is populated have values in the format defined and named by XMLCODELIST. Most of the other CNTLTERM checks compare the values of an individual variable named in COLUMNSCOPE in the validation control data set to the format named in the corresponding value of LOOKUPSOURCE. Any variable named in one of these checks whose metadata variable XMLCODELIST is populated will be checked against its controlled terminology in two different checks. For example, the value of XMLCODELIST in the REFERENCE_COLUMNS provided by SAS is AESEV. That means that check SDTM0221 will check values of this variable against a SAS format called AESEV (as well as other variables with a non-null value for XMLCODELIST). In the master validation control data set, the value of LOOKUPSOURCE for

check SDTM0467, a CNTLTERM check, is $AESEV. Both of these in the run-time control data set will result in the same compliance issue being checked.

Technically, there's nothing in the CST that prevents us from executing checks in any order we want. However, a careful examination of all of the checks and the topics they cover reveals a logical flow that can serve to guide the user both in terms of order as well as execution groups. The groups defined by SAS give us a good start, but we can dive a little deeper.

SAS categorizes the checks into the following groups: CNTLTERM, COLUMN, COLUMNATTRIBUE, COLUMNVALUE, DATE, DERIVATION, METADATA, MULTIRECORD, and MULTITABLE. While METADATA and COLUMNATTRIBUTE are concerned with data set and column structure, the rest of the categories are concerned with data values. The overall approach advocated in this paper follows a natural sequence that begins with structural checks. These are checks that involve metadata and comparisons between reference and source metadata. This starts at the domain level and then works its way down to the column level. Once confident that these are in place, and that the source metadata is set up the way it is supposed to be according to the reference metadata, then we're ready to move more into value-level checks. In terms of the SAS categories, on a high level, this means starting with the METADATA checks, and then progressing to the COLUMNATTRIBUTE checks. More specifically, within METADATA, checks SDTM0004 and SDTM0006 are good to start with. They make sure that data set-level alignment exists between reference metadata, source metadata, and the data itself. Check SDTM0001 makes sure that all domains have observations. Once data set alignment is known to be in place, we move into column alignment and existence. Still within METADATA, SDTM0015 checks alignment between reference and source metadata, while SDTM0012 and SDTM0013 make sure that required and expected columns, as determined by metadata, are present. Once column existence is established, then we can look at adherence to standard column attributes such as length (SDTM0022 and SDTM0023), labels (SDTM0030), type (SDTM0019), and order (SDTM0020). These checks still fall into SAS's METADATA category, but this is also a good time for SAS's COLUMNATTRIBUTE category, which make sure that the number of characters in values of certain variables don't exceed thresholds specified in the SDTM IG. Examples include the eight character limit on the values of --TESTCD, QNAM, ETCD, etc.

We can begin our venture into the value level checks with those concerned with the existence of data. Many of these are found in the COLUMN category such as SDTM0201 and SDTM0271 which make sure required and key variables have values. The MULTIRECORD check SDTM0605 makes sure all variables have values for at least one observation. Others include making sure variables aren't don't have null values when another variable is non-null. One example of this is check SDTM0231 which makes sure AGE isn't null when AGEU isn't null. SDTM0507 checks the opposite, that AGEU has a value when AGE is populated.

Next, single value checks can be executed. These include all of the controlled terminology checks (CNTLTERM), plus various COLUMNVALUE and DATE checks. COLUMNVALUE checks make sure column values make sense, such as AGE>0 (SDTM0506), DOMAIN=the name of the domain (SDTM0206), study day variables (--DY) are never 0 (SDTM0222). DATE checks like SDTM0101 and SDTM0102 make sure date and duration values are ISO8601 compliant. Following these are certain COLUMN and DATE checks that check consistency of values across variables, such as the requirement that ARM is set to "Screen Failure" or "Not Assigned" when ARMCE is "SCRNFAIL" or "NOTASSGN" (checks SDTM0500 and SDTM0501).

Once comfortable that variable values are populated when necessary, and populated appropriately and consistently with other variables, we can then start checking that they're consistent with variable values from other records (MULTIRECORD) and even with other data sets (MULTITABLE). Examples of the former include checks for uniqueness (SDTM0602 checks uniqueness in the key variables, 0641 checks USUBJID uniqueness in DM). Once a data set's data appears clean within its own context, then the MULTITABLE checks can be executed to make sure rules that span data sets are met. Examples include making sure that USUBJID values found outside of DM are also found in DM (SDTM0801), subject element codes (ETCD) and visits found outside of TE and TV respectively, are also found in those domains (SDTM0811 and SDTM0846).

The checks mentioned in the examples above are far from exhaustive, and the approach is just one of a number of approaches to test. The above discussion is meant to serve as a guide for those getting started with compliance checks, but in the end, it will require time and experimentation for an organization to decide which checks are important, and which execution strategy works best for them.

CONCLUSION

Since the dawn of organized clinical trial data standards when data managers, programmers, and statisticians started to see language in the documentation about required data elements, organizations have been trying in some fashion not only to create compliant data, but somehow "make sure" it was right. Because we create the data by writing SAS code, using the same method to check compliance seems a natural extension. Unfortunately, without discipline built around this process, compliance checks get lost in free-text code, often localized to individual work stations, and become nearly impossible to maintain and spread across an organization. In short, they become difficult to manage.

The CST attempts within the free text editor of BASE SAS to take a step back, look at everything that needs to feed into such a process, account for it all with an intelligent directory and data set structure, and build a framework around it. Yes, opportunity still exists for programmers to slip outside of the process, but with proper foresight and management of central files, this can be kept to a minimum.

RECOMMENDED READING

- SAS Clinical Standards Toolkit User's Guide 1.4
- CDISC SDTM Implementation Guide

CONTACT INFORMATION <HEADING 1>

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

Name:	Mike Molter
Company:	d-Wise Technologies
Work Phone:	919-600-6237
E-mail:	mmolter@d-wise.com