Paper TS05

Controlling OpenCDISC Validator using R

Martin Gregory, Merck Serono, Darmstadt, Germany

1 Abstract

The OpenCDISC Validator GUI writes output to a fixed location in the installation folder, a behaviour which is not acceptable in a server environment. Furthermore, on each call the user must navigate to the location of the files to be checked. OpenCDISC does include a command line interface (CLI) which offers user control of both input and output locations but requires relatively complex commands.

One solution is to use a scripting language to read the input parameters for the OpenCDISC CLI from a file and make the appropriate calls to OpenCDISC. This approach can be independent of operating system and also facilitates repeated runs. Scripting languages such as Perl or Python are well suited to such tasks but are not always available. We show that R, usually available for statistical analysis reasons, can also carry out such tasks with ease and in a manner which hides operating system level details from the programmer.

2 Introduction

The OpenCDISC Validator is a project of OpenCDISC [1], an open source community dedicate to building frameworks and tools to facilitate the adoption of CDISC standards. The validator is a Java application which checks a set of data, formatted according to a CDISC standard, for compliance with the rules for the chosen standard. The rules are implemented using XML which allows the user to define additional rules if so desired. Checking the compliance of a set of data against a standard is an important task, both in general and specifically during the submission process. As the models are very complex, use of such an application is more efficient than coding the compliance checks oneself.

In this paper we are primarily concerned with the process of running the OpenCDISC validator and not with the results of the runs. Just as the validator provides an efficient way to check a set of data, the aim of this paper is to present a relatively simple R function which simplifies the task of running the validator on a large number of sets of data and repeating runs on new versions of sets of data. R [2] is a free software environment for statistical computing and graphics maintained by the R Foundation for Statistical Computing [3].

We also briefly compare an alternate implementation of the function in the Python [4] programming language.

In developing the R function we used two different versions of the OpenCDISC Validator: 1.2.1 and 1.3 both of which can be downloaded from the OpenCDISC web site.

3 Limitations of the OpenCDISC GUI

While the OpenCDISC GUI is relatively comfortable to use, there are a number of drawbacks, especially when many validation runs on many different sets of data are required. In particular:

- the GUI does not allow the user to choose the directory in which the report is to be placed, but attempts to write to the *reports* sub-directory of the installation directory. On server systems, this is generally neither possible nor desirable. In fact one will generally want to store the report in a directory containing other study-related files;
- the GUI only remembers the settings for the last run. So a user who is responsible for validation runs for multiple sets of data has to set the model standard, navigate to the source files, choose the *define.xml* file and the standards version for each run. Given the iterative nature of building SDTM or ADaM datasets, this is a significant shortcoming;
- In certain circumstances it can be of interest to run the validation on a specific subset of the datasets as execution times for large datasets can be quite long. While the GUI provides such a feature, it does not provide a way to save this selection across sessions;
- Batch or scheduled runs are not possible with the GUI.

4 OpenCDISC Command Line Interface

The OpenCDISC validator also includes a command line interface (CLI) *validator-cli-1.3.jar* located in the *lib* sub-directory of the installation. The CLI supports options for specifying

- whether a validation or generation task is to be run;
- which type of standard to use (SDTM, ADaM, Define.xml, SEND, Custom);
- the location of the files to be validated;
- the OpenCDISC configuration file, define file and codelists to use; and
- the location and format of the report to produce.

Full details of the options supported can be obtained by executing the command

```
java -jar /path/to/OpenCDISC/lib/validator-cli-1.3.jar -help
```

The CLI clearly allows us to overcome the limitations of the GUI. The command to execute a specific run is, however, quite long and a large number of options must be specified. Some of these options are constant across calls, for example, the location of the configuration file for a particular standard is always the same. Furthermore, the CLI accepts only a directory name, a file specification of the form *path/*.xpt* or the path to a single dataset as input. Wildcards must specify the entire dataset name, i.e., a wildcard of the form *path/a*.xpt* results in only the first matching dataset being validated. So in this respect the CLI is inferior to the GUI.

5 Solution

To overcome the drawbacks of both GUI and CLI, and also to provide some error checking in terms of existence of required files, we designed and implemented a script which reads the options from a parameter file. The parameter file allows the specification of multiple calls to the validator and contains all the necessary and sufficient information required to construct the calls. The syntax for calling the script should be

```
script [-log log-file] [-version version] parameter-file
```

The log file is a record of the script's actions, along with any messages produced by the validator. We will see some examples in section 6.4 below. The current implementation derives the log file name from the parameter file name.

5.1 The parameter file

The parameter file read by the script is a simple text file obeying the following rules:

- arguments for a single call of the validator are entered as a white space separated list on a single line
- there are four arguments:
  1. specification of the file(s) to validate. The validator CLI accepts only directory names, individual file names or wildcards such as *.xpt. A wild-card such as a*.xpt results in only the first file matching the specification being processed;
  2. the standard to use. This can be one of *sdtm-3.1.1, sdtm-3.1.2, adam-1.0* or *define-1.0* and is used to construct the path to the validator's configuration files using the pattern */path/to/OpenCDISC/config/config-standard.xml*. You may also explicitly specify a configuration file by prefixing the path to the file with the $\sim$ character;
  3. the location of the report file;
  4. the type of report to produce;
- blank lines are ignored;
- a line on which the first non-white space character is a # sign is ignored and may be used for comments;
- file specifications may be absolute or relative. If the latter the paths must be relative to the location of the parameter file.

A sample parameter file follows, illustrating the rules listed above:

```
# Purpose: Sample parameter file. This line is a comment

sdtm/ae.xpt sdtm-3.1.2 report/sdtm_ae.xls Excel
sdtm/dm.xpt sdtm-3.1.2 report/sdtm_dm.xls Excel
adam/*.xpt adam-1.0 report/adam.xls Excel
adam/*.xpt ~config/cfg-adam-1.0_plus.xml report/adam_alt_cfg.csv Csv
```

This parameter file will cause the validator to be called four times: once each for the SDTM AE and DM datasets, once for all ADaM datasets using the configuration file supplied by OpenCDISC and once using a customized configuration file. The directories *adam*, *sdtm*, *config* and *report* are all in the same parent directory as the parameter file itself.

Allowing the parameter file to contain options for multiple calls of the validator solves the problem of validating a selection of the datasets in a directory.

5.2 Structure of the script

The script should carry out the following actions:

1. read the parameter file into a two-dimensional data structure with one dimension representing the call and the other dimension representing the parameters for the call. The parameter dimension has to store values for the validator options *-source, -config, -report, -report:type, -log, -type* and, optionally, *-config:define*.

2. check the existence of any input files (source, configuration, directory to contain the report) and the validity of the report type;

3. if a *define.xml* file is present in the source directory, use this as argument to the *-config:define* option[1]. This causes the validator to carry out additional checks which rely on information in the *define.xml* file;

4. execute the validator for each call which passed the checks, capturing any messages to standard output or standard error.

We also decided to parse the entire file before beginning execution as this allows us to specify that no calls should be executed if any errors are found. This feature, however, is not implemented in the current release which executes all calls that pass the checks.

5.3 Choice of scripting language

In choosing a scripting language, we wanted to be independent of operating system, so shell scripts were not an option. At the same time, the scripting language should be close to the operating system as essentially the script is building a command to execute, given that certain pre-conditions are met. The most common such scripting languages are Perl and Python. Both languages provides arbitrary anonymous data structures through combining lists and associative arrays[2] and the ability to easily run operating system commands and collect their output. Because identical versions of Python are easily available on both UNIX and Microsoft operating systems, we concentrate on Python. Python provides functions for checking the existence of files, converting constructed file paths to the canonical form for the operating system, reading files into lists and using so-called file *globbing* which will turn out to be useful for checking the existence of the source files. It also provides extensive support for regular expressions. So all the actions which the script needs to carry out can be implemented fairly easily in Python.

One other possible language for implementation is the R language. R provides, among other data structures, lists and associative arrays[3], file manipulation functions including globbing and regular expression support. It also provides a function for executing operating system commands and retrieving their output. Furthermore, it provides all these features in the same way across operating systems. Additionally, R is becoming popular in the biostatistics and statistical programming fields so that support and maintenance is less of an issue than for Python or Perl. Finally, operating R in a validated environment is well supported by the R Foundation which provides an extremely detailed description of the development methodology and quality control [5] and a comprehensive set of installation and operational qualification scripts which are easy to run. R is also used by FDA reviewers [6]. As a result of these considerations we decided to implement the script in R.

[1] For our purposes we decided to always use the define.xml if present, but it would be easy to make this optional

[2] in Perl hashes; in Python: dictionaries

[3] vectors with named elements

6 Details of the R implementation

The R implementation consists of two files, both of which may be downloaded from the PhUSE Wiki [7]. The file *runocdv.r* defines the function *runocdv()* which parses the parameter file, checks the inputs and executes the validator if checks are passed, writing a log file of the actions and results. The syntax of the function is

```
runocdv(file,
        ocd.path=c("1.3"="/opt/OpenCDISC/V1.3","1.2.1"="/opt/OpenCDISC/V1.2.1"),
        ocd.def.ver="1.3",
        ocd.java="/usr/lib/java/bin/java",
        log=NA
        )
```

This function can only be called from another R program, so to allow the function to be called from the command line or from a file manager such as the Microsoft Windows Explorer, we provide a second file *call.runocdv.r* which serves as an example and which can be called directly from the operating system. This loads the *runocdv()* function and executes it passing the parameter file specified in the call. The script uses a magic number pointing to *Rscript* so on UNIX-like systems, if this file is made executable, it can be called directly:

```
call.runocdv.r parameter-file
```

On Microsoft Windows it may be necessary to call *Rscript* explicitly:

```
\path\to\R\bin\Rscript call.runocdv.r parameter-file
```

In the following sections we describe the salient features of the *runocdv()* function.

6.1 Initialisation

This section of the program sets up a vector of valid values for the report type, defines functions for writing a usage message and for writing to the log file. It then checks whether a parameter file was specified and whether the specified parameter file exists. If either of these conditions are not met, the function terminates with a non-zero return code. Finally, if no log file name is specified, a log file name is constructed from the parameter file name by replacing the file type *ocd* with *log* and the log file is opened:

```
## connection for log file and function to write to it
if (is.na(log)) {
  log <- sub("ocd$","log",ocdfile)
}
ocdlog <- file(log,"wt")
```

The *is.na()* function checks whether a variable has a value or not. To actually use a file, the handle returned by the *file()* function must be used. Finally, we write a message to the log indicating the time, the name of server on which we are running and the name of the parameter file:

```
writeLog(c(paste("NOTE: runocdv.r Revision: 1.0 on server ",
                 Sys.info()["nodename"]),
           paste("NOTE: processing job list",
                 basename(ocdfile),"at",Sys.time()),
           ""))
```

Note that operating system details are hidden from us - the *Sys.info()* function returns appropriate information and we do not have to bother testing the operating system and the checking for information in a particular way. This makes for simple, more robust programs which still run on multiple operating systems.

6.2 Reading the parameter file

This section reads the parameter file and loads it into a two-dimensional data structure. It illustrates how a single function call in R can operate on an entire data structure. For example, the *readLines()* function:

```
fid <- file(ocdfile, "r")
fid.contents <- readLines(fid,-1,TRUE)
```

reads the entire file into a character vector with each line of the file becoming an element of the vector. After this the *grepl* function is used to construct a logical vector specifying which elements of *fid.contents* are neither blank nor comments, and this logical vector is used to subset *fid.contents* itself:

```
fid.jobs <- fid.contents[!grepl("^([[:space:]]*#|[[:space:]]*$)",fid.contents)]
```

The resulting character vector *fid.jobs* contains one element for each of the lines defining calls to the validator. The next step is to parse each of these vector elements into a vector where each element contains one parameter value. This is achieved using the *strsplit()* function:

```
ocd.param <- strsplit(fid.jobs,"[[:space:]]")
```

The resulting object *ocd.param* is a list with one element per call. The list elements are the vectors of parameters for each call. In our example above the list (numbered items) would have four elements, each of which is a character vector with four elements:

1. ('sdtm/ae.xpt', 'sdtm-3.1.2', 'report/_sdtm_ae.xls', 'Excel')
2. ('sdtm/dm.xpt', 'sdtm-3.1.2', 'report/_sdtm_dm.xls', 'Excel')
3. ('adam/*.xpt', 'adam-1.0', 'report/_adam.xls', 'Excel')
4. ('adam/*.xpt', '~config/cfg-adam-1.0_plus.xml', 'report/adam_alt_config.csv', 'Csv')

Note that the syntax for elements of lists and vectors use double and single square brackets, respectively: *ocd.param[[i]]* is the i^{th} element of the list *ocd.param*, while *ocd.param[[i]][j]* is the j^{th} element of the vector *ocd.param[[i]]*. In our example, ocd.param[[1]][4] has the value *Excel*.

6.3 Constructing the arguments

This section of the function checks the correctness of parameter values and the existence of required inputs. Informative messages are written, both in the case of passing or failing the checks. The section is a loop over the list of parameter sets *ocd.param*. In order to build the arguments for the validator CLI and the messages to be written to the log, we use two additional two-dimensional data structures, both lists of vectors set up using the *vector()* function:

```
jobs.n <- length(ocd.param)
ocd.msg <- vector("list",jobs.n)
ocd.args <- vector("list",jobs.n)
```

ocd.args for the command and *ocd.msg* for messages, both with a length equal to the number of calls (*jobs.n*). This enables us to have access to the original values in the parameter list. We use a further vector to record the result of the checks for each call.

In any case where a file path is constructed, the *normalizePath()* function is used to convert the result to the appropriate syntax for the operating system on which the program is running. In addition to converting between forward and backward slashes, it also simplifies relative addresses and collapses multiple directory separator characters to produce a file path in canonical form.

The input files, i.e., the value for the *-source* argument of the validator, may be specified with wildcards. To simplify checking the existence of these files we use the *Sys.glob()* function:

```
if (length(Sys.glob(ocd.param[[i]][1]))==0) {
  # write error message
} else {...}
```

which allows us to avoid parsing file specifications with wildcards. This function returns a vector of files matching the specification so we only have to test that the length of this vector is non-zero to know that at least one file matches the specification.

When checking for the existence of the directory to which the report and log files will be written, we use the *dirname* function:

```
if (!file.exists(dirname(ocd.param[[i]][3]))) {
  # write error message and stop processing this job
} else {...}
```

which simplifies parsing of the directory part of a path and again removes the burden of having to have a different set of parsing code for each operating system. We also use the related *basename* function:

```
  if (basename(ocd.param[[i]][1])=='define.xml') {
```

to check if the source file is *define.xml* in order to know if we should set the *-type* parameter to *Define* rather than the default *SDTM*. Finally, for checking against a list of valid values which we have stored in a vector, we can use the *is.element* function:

```
if (!is.element(tolower(ocd.param[[i]][4]),ocd.reptype)) {
```

6.4 Executing the validator

We could have placed the code for executing the validator in the same loop as the one constructing the arguments and checking for existence and correctness. We chose not to do that in order to allow a future option of executing the validator only if inputs for all calls pass the checks.

The section for executing the validator is very simple: we first write the messages collected while constructing the arguments. If the check were passed, we then concatenate the arguments and execute the resulting command using the *system()* function:

```
ocd.jcmd.out <- system(ocd.java.cmd,intern=TRUE)
writeLog(ocd.jcmd.out)
```

The *intern=TRUE* option returns anything written by the command to standard error or standard output and we write this to the log file.

The log file from a run of the example parameter file above starts with a header announcing the program and the name of the parameter file:

```
NOTE: runocdv.r Revision: 1.0 on server  green
NOTE: processing job list src2_1.3.ocd at 2012-07-29 13:44:53
```

For each call, a report of the input parameters, expanded to complete paths in the case of file names, is produced. Additionally, any messages produced by the validator itself are written to the log.

```
   Source: /data/projects/runocdv_v1.0/sdtm/ae.xpt
   Define: /data/projects/runocdv_v1.0/sdtm/define.xml
 Standard: /opt/OpenCDISC/V1.3/config/config-sdtm-3.1.2.xml
   Report: /data/projects/runocdv_v1.0/report/sdtm_ae.xls
     Type: Excel
OpenCDISC: 1.3
------------------------------------------------------------
         OpenCDISC Validator Messages
------------------------------------------------------------
 Beginning validation, please wait...
 The validation has completed.
------------------------------------------------------------
      Log: no log was created.
```

The final message that no log was created refers to a log file which may be created by the validator. The location of this file is specified by the (undocumented) validator option *-log*. In our tests it has never been created.

7 Comparison with a Python implementation

Since the original intention was to implement the script in Python, after completing the implementation in R we wrote the script in Python. Given the availability of similar data structures and operating system level functions as in R, the structure of the Python script is identical to that of the R script. Where we used lists of vectors for the two-dimensional data structures in R, we used lists of dictionaries in Python. This allowed us to use names instead of indexes for the parameter elements which makes the code more readable. In fact, we could have used named elements for the vectors and achieved the same result.

From a resource usage point of view, there is no difference as far as CPU usage is concerned - in both cases it is minimal and actually zero while the validator is running. *call.runocdv.r* running under R 2.14.2 on on a GNU/Linux system with a 2.6.37 kernel uses about 22MB of memory while *runocdv.py* running under Python 2.6.6, also in line mode on the same system, uses only 5MB. In comparison to the OpenCDISC validator itself which uses at least 400MB even when processing a single dataset and consumes as much CPU as is available, the resource usage of either script is negligible.

8 Conclusion

The OpenCDISC validator is an important tool not only in preparing submissions for FDA but also in general for checking that a set of data conforms to a CDISC model. For high-volume use, however, a number of limitations of the validator make it labour-intensive and therefore error-prone. The GUI does not allow the user to choose where the output files should be written and will actually write them to the installation directory. Furthermore

the GUI remembers only some settings from the previous run so the user is required to reselect options and, in particular, to navigate to the input files. The CLI allows full control but requires that the user type very long commands. We have presented an approach which allows the user to overcome all of these limitations: the user writes the parameter values for an arbitrary number of calls to the validator in a file and executes our runocdv() function to generate and execute the call. This approaches allows automation, reduces error, improves traceability and is furthermore easy to implement. The choice of the R language to implement the function shows that R is not only a language for data analysis but is also a useful scripting language for controlling the execution of other programs. The availability of arbitrary anonymous data structures allows the programmer to represent the data required for the problem with ease. In particular, the way in which R hides operating system details significantly reduces the amount of effort involved for such a task and also reduces the complexity of the code. The function is available for download from the PhUSE Wiki. We plan to build a package including installation tests, documentation and examples.

References

[1] http://www.opencdisc.org accessed on 30 July 2012

[2] http://www.r-project.org accessed on 30 July 2012

[3] http://www.r-project.org/foundation/main.html accessed on 2 August 2012

[4] http://www.python.org/ accessed on 30 July 2012

[5] http://www.r-project.org/doc/R-FDA.pdf accessed on 24 August 2012

[6] http://www.r-project.org/conferences/useR-2007/program/presentations/soukup.pdf accessed on 24 August 2012

[7] http://www.phusewiki.org/wiki/index.php?title=Controlling_OpenCDISC_Validator_using_R accessed on 2 August 2012

Contact Information

Martin Gregory
Data Integration and Statistical Systems
Merck Serono
Frankfurterstr 250
64293 Darmstadt
Germany
Martin.Gregory@merckgroup.com