PharmaSUG 2026 - Paper ET-279

Goodbye SAS, Hello R: Practical Workflows for CDISC Standards

Madeleine Penniston, Alyssa Wittle, Atorus Research

ABSTRACT

The adoption of open-source tools in the pharmaceutical industry is rapidly transforming how clinical data is managed, analyzed, and delivered for regulatory review. Among these tools, R has become the leading choice, but what does programming in R really mean for today's statistical programmer? This paper presents an end-to-end CDISC workflow demonstrating how statistical programmers can manage the lifecycle of clinical trial data using R as the primary analytical engine. Beginning with SDTM development, the process illustrates techniques using CDISC standards and demonstrates controlled-terminology mapping using packages such as sdtm.oak. Using a consistent programming approach, raw CRF inputs are transformed into standardized datasets across a variety of domain types. The paper then transitions into ADaM dataset creation, highlighting how R can easily accommodate the Subject-Level Analysis (ADSL), Basic Data Structure (BDS), and the Occurrence Data Structure (OCCDS) datasets. Within each example, practical functions in the admiral package are introduced to derive baseline values, flags, and other key analysis variables. Each coding stage highlights reproducibility strategies and environment control through the use of renv. The advantages of R are also explored, including easier transposing techniques, simple row-wise operations, and the absence of fixed length variable requirements. These allow programmers to reduce the risk of data truncation and other structural errors. Together, these components provide a comprehensive view of how R can support a submission ready clinical programming workflow for SDTM and ADaM development.

INTRODUCTION

R is a programming language and environment for statistical computing and graphics that are widely adopted across academic settings. It is often praised for syntax that allows a user to perform complex statistical procedures in one line of code; for instance, fitting a linear regression model using a concise readable statement. As the pharmaceutical industry increasingly embraces open-source technologies, R is on track to become the leading language for clinical trial data analysis.

Traditional statistical programming environments, like SAS, have historically been valued for their stability and standardization in regulatory submissions. However, because SAS is developed within a single commercial ecosystem, it offers a more centralized approach to programming. In contrast, R's open-source foundation supports a more collaborative, community driven environment, with diverse programming approaches and continuous public feedback.

With R's rapid growth in popularity, important questions arise:

- Is developing SDTM and ADaM more efficient in R?
- How can CDISC standards be implemented in a structured, reproducible, and regulatory aligned manner?

This paper presents an end-to-end CDISC workflow introducing how statistical programmers can manage the lifecycle of clinical trial data using R as the primary analytical engine.

TRANSITIONING FROM SAS TO R

The transition from SAS to R can be daunting. While there is a vast collection of resources designed to support clinical programmers, the most significant challenge is adapting to different programming mindsets.

MINDSET CHANGES

The largest conceptual difference is understanding how data is processed. SAS uses row-oriented and stepwise data processing while R uses column-oriented, object-based programming. In R, datasets are treated as objects, where each column is a vector, resulting in operations being applied to entire columns

at once, as opposed to each row. For example, instead of the SAS data step, an assignment operator is used in R. As a result, programming tasks are expressed differently in R than in SAS. For instance, creating a new dataset:

```
data new;
    set old;
run;
```

Program 1. SAS dataset creation

The same operation is performed with one simple assignment in R:

```
new <- old
```

Program 2. R Data Frame Creation

R completes the process in fewer lines of code, in a readable format for newer users. The assignment operator (<-) creates a new object named “new” and assigns the contents of the old object “old”.

FUNCTIONS AND PACKAGES

Another important transition for SAS programmers is moving from macros to functions. A function is a piece of code that takes some input and performs a predefined task. Functions are very similar to SAS macros and SAS procedures such as PROC SORT, PROC TRANSPOSE, etc. The majority of programming in R is done using functions. A collection of functions is called a package. The open-source nature of R allows anyone to create a package and publish it to an online repository for community use. For clarity, package names are written with curly brackets using the syntax {packagename}.

The {pharmaverse} is an ecosystem of R packages that are designed to support CDISC data standards and clinical trial workflows. It consists of curated R packages for the intention of generating FDA ready submissions. These are not validated packages, but curated packages submitted by the community to aid clinical programming.

There are two methods of calling a function in R. The first is a package can be loaded into the working environment and then any function within that package can be called directly in a program:

```
library(packagename)
functionname()
```

Program 3. library() Call Syntax

Alternatively, functions can be called using the double colon (::) operator, which tells R to use a function from a specific package without first loading the entire package into the session. This is similar to referencing a dataset using a SAS library name (e.g., libname.dataset), where packages act like the library and the function is the specific object being referenced.

```
packagename::functionname()
```

Program 4. Function Call Syntax

For clarity in this paper, functions will be referenced using the double colon syntax, however it is not a required method of calling a function, and in many cases, not the preferred method of coding.

KEY PROGRAMMING TOOLS

The most used tool within R is the pipe operator (|>). The pipe feature allows for multiple data transformations to be written in a step-by-step sequence without creating intermediate objects. Conceptually, the pipe can be read as “take that object and do this to it”.

For example:

```
# Read as, take dm and create the column STUDYID
ADSL <- dm |>
    dplyr::mutate(STUDYID = “abc123”)
```

Program 5. Pipe Operator Example

In this code, the output of dm is passed into the dplyr::mutate() function, and creates a new variable named STUDYID, then stores it within the object ADSL.

The pipe allows transformations to be written in the order they are programmed. Instead of breaking logic across multiple data steps, programmers can build transformations sequentially, reducing the intermediate code that must be tracked.

Another helpful feature is the help operator (?). Typing:

```
?packagename::functionname
```

Program 6. Help Operator Example

will immediately display the function syntax for the function, providing quick access to reference source materials for all functions within the working environment. For example, the R documentation for dplyr::mutate() would be displayed as shown in Figure 1.

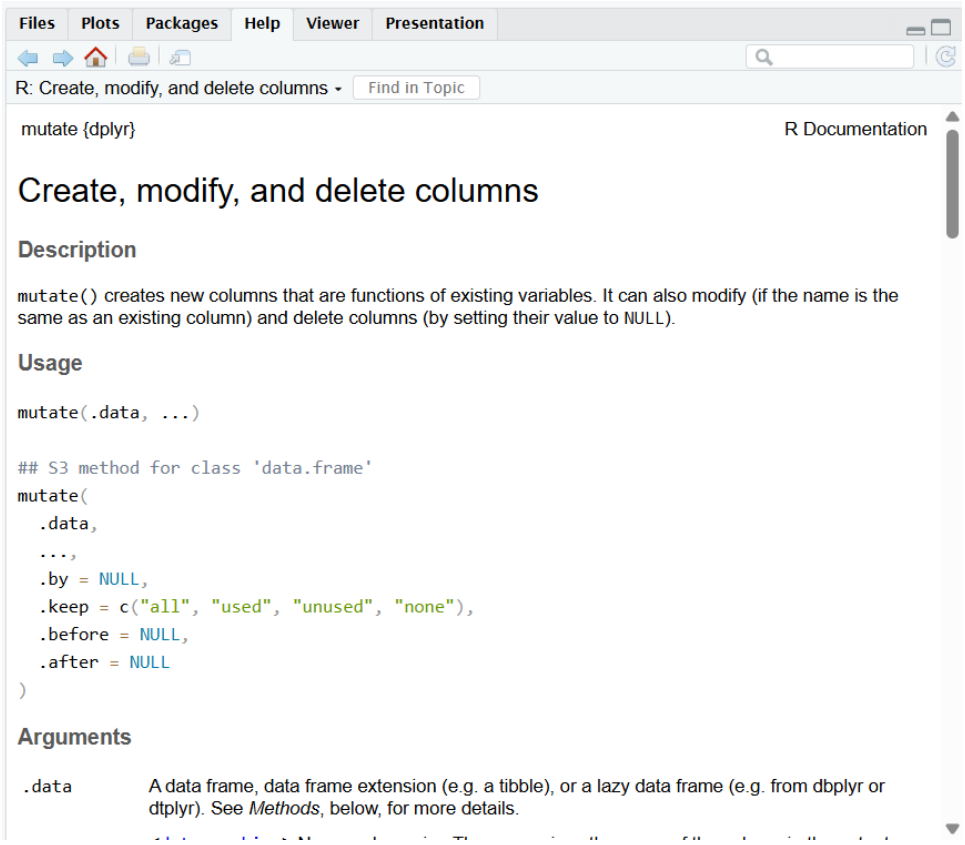

Figure 1: Example of R Help Documentation for dplyr::mutate

This built in R feature provides instant access to reference materials directly within the programming environment, allowing users to quickly learn and apply new functions without the need to search externally.

The use of functions simplifies common data manipulation tasks. For example, transposing data or performing row-wise calculations can be implemented using fewer lines of code when compared to the SAS equivalent. Consider the SAS code to transpose ADVS from long to wide format.

```
proc sort data = advs;
    by USUBJID AVISIT AVISITN;
run;

proc transpose data=advs out=advs_wide;
    by USUBJID AVISIT AVISITN;
    id PARAMCD;
    var AVAL;
run;
```

Program 7. Transposing ADVS in SAS

Then consider the R code to transpose ADVS from long to wide format.

```
advs_wide <- advs |>
    tidyr::pivot_wider(
        id_cols = c(USUBJID, AVISIT, AVISITN),
        names_from = PARAMCD,
        values_from = AVAL)
```

Program 8. Transposing ADVS in R

Above, the tidyr::pivot_wider() function performs the same operation without the need to sort ADVS prior to performing the transpose.

In addition to eliminating the need to sort data prior to the transformation, R offers an additional advantage for character values. Character values do not have fixed lengths, eliminating the risk of truncation.

REPRODUCIBILITY AND PACKAGE CONTROL

SAS programs are typically executed with a controlled validated environment where software versions remain consistent throughout the course of a study. R packages are sourced from an ecosystem that is constantly evolving to improve on itself. Constant updates to R-packages within a study can cause unknown results, however, to solve this challenge, the package {renv} allows programmers to create reproducible controlled project environments. {renv} creates a project specific library folder and records the exact package versions of all packages used within the project in a file called renv.lock. This file is a snapshot of your programming environment.

When the project is reopened at a later time, the function call,

```
renv::restore()
```

Program 9. Restore Programming Environment

will install the exact package versions in renv.lock. This ensures all data manipulation occurs using the same controlled environment, providing a similar workflow to SAS.

Finally, package validation should be considered when utilizing any R package. It is important to note, {pharmaverse} packages themselves are not validated software, however organizations often implement internal validation frameworks to assess package risk and suitability for regulated workflows.

TRANSITIONING INTO SDTM DEVELOPMENT

With the fundamental ideas established, the next step is to apply these concepts to the first stage of the clinical data standardization lifecycle: Study Data Tabulation Model (SDTM) development.

The first step to any R program is the setup of the programming environment. Unlike SAS where the setup consists of log configuration and system options, the R workflow typically begins with library() statements. For example:

```
library(packagename1)
library(packagename2)
```

Program 10. library() Statements

Loading in the necessary libraries in the beginning will improve reproducibility and readability within a program. It clearly documents the dependencies required to run the program and ensures all functions are available within the working environment.

At a high-level breakdown:

SAS Programming workflow	R Programming workflow
Log setup and system options	Library statements
Data import statements	Data import statements
Data steps and procedures	Preprocessing, derivations and validation

SDTM DEVELOPMENT IN R

SDTM development begins with importing raw Case Report Form (CRF) data into the programming environment. CRF data are commonly delivered using SAS transport files (*.xpt) or as SAS datasets (*.sas7bdat). Importing these file types is made simple using the package {haven}, which offers helpful functions for reading in SAS files into R. The package is designed to preserve labels, variable types, and precision of numbers during the import process, ensuring nothing gets lost along the way. To read in a CRF dataset such as dm.sas7bdat, located in a folder named rawdata, simply call:

```
haven::read_sas("rawdata/dm.sas7bdat")
```

Program 11. Importing dm.xpt file

Once all necessary CRF data are imported into R, data manipulation can begin. An important consideration at this stage is how R and SAS handle missing values differently. Unlike SAS, which uses a period (.) and empty quotes ("") to symbolize a missing value, R has one unified missing symbol denoted "NA". When importing SAS datasets into R, variables may not be automatically stored as NA.

From a CDISC standpoint, missing values must be represented consistently as null values (SDTMIG 4.2.5). If empty strings are retained when performing data manipulation in R, these are not treated as null and therefore will not be interpreted as missing.

The best practice is to ensure consistency when importing data. This can be implemented using dplyr::mutate() and dplyr::across() functions:

```
dm <- haven::read_sas("rawdata/dm.sas7bdat") |>
    dplyr::mutate(
        dplyr::across(where(is.character), ~replace(.x, .x == "", NA))
    )
```

Program 12. Creating Consistent NULL Values

The function haven::read_sas is used to read the raw DM dataset into R. The pipe operator (|>) then passes that dataset directly into the dplyr::mutate(), allowing the transformation step to follow in sequence. Within dplyr::mutate(), dplyr::across() applies the same transformation to multiple variables at once. The statement, where(is.character) limits the transformation to character variables only, as this is where empty strings occur.

The function ~replace() is then applied to each selected variable. Here .x represents the current variable being processed, .x == "" identifies blank character values, and replace(…, NA) converts the blank values into NA.

Once the CRF and missing values are consistently represented as null, the next step is ensuring controlled terminology validation and compliance checks are incorporated into the programming workflow. These checks can be implemented manually or the package {sdtm.oak} provides a framework for modular SDTM programming. Instead of manually programming compliance checks, {sdtm.oak} incorporates these directly into the derivation of a variable. The package creates unique identifiers to map topic variables, followed by qualifier and timing variables to generate domains.

Useful functions such as sdtm.oak::assign_ct(), allow a variable to be derived while simultaneously applying controlled terminology validation. The workflow in R advances beyond simple data manipulation, and transforms into structured, metadata-driven SDTM development.

SDTM DEVELOPMENT: DM EXAMPLE

The SDTM implementation in R follows the same general principles; however, variable derivations vary by

domain class. The Demographics (DM) Domain is the primary subject level dataset for all other observations within a study. Containing one record per subject, the domain is commonly sourced from multiple CRF forms.

In SAS merging requires a PROC SORT of both datasets, that carefully align BY variables and then navigating different merging relationships. This can be seen within Program 13. Failure to execute these can result in warnings, dropped records, or incorrect merging. In contrast, merging in R is simplified through join functions, demonstrated in Program 14. The {dplyr} package provides easy to use merging functions that allow the user to easily pick which merge relationship best fits the data.

```
proc sort data = dm;
    by USUBJID;
run;
proc sort data = ex;
    by USUBJID;
run;
data dm_ex;
    merge dm(in = a) ex(in = b);
    by USUBJID
    if a;
run;
```

Program 13. Merging in SAS

```
dm_ex <- dplyr::left_join(dm, ex, by = "USUBJID")
```

Program 14. Merging in R

In Program 13 and Program 14, two datasets dm and ex are being merged by USUBJID, keeping all records within dm. In R, there are four {dplyr} merging functions that allow a programmer to control how the datasets are merged.

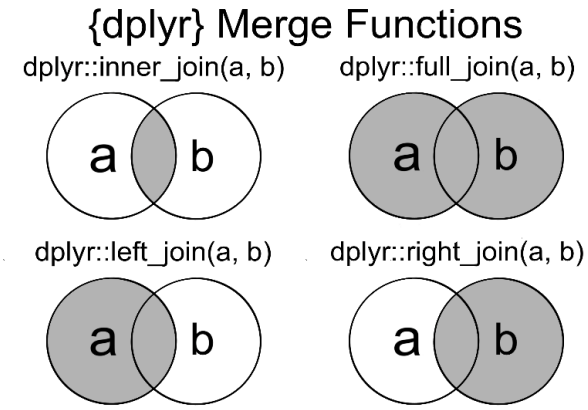

Figure 2: {dplyr} Merge Functions

The merging functions can be utilized with or without specific BY key variables and without prior sorting. This allows the user to focus more on the relationships between datasets. For example, merging DM (Figure 3.) with EX (Figure 4.), using left join methodology without explicitly defining a BY variable:

dm

Filter

	USUBJID	SEX	AGE
1	101	M	65
2	102	F	72

Figure 3: DM Input

ex

Filter

	USUBJID	EXTRT	EXDOSE
1	101	DrugA	50
2	101	DrugA	75
3	102	DrugB	100

Figure 4: EX Input

```
dm_ex <- dplyr::left_join(dm, ex)
```

Program 15. No BY Key Merge in R

The output of Program 15 is displayed in Figure 5.

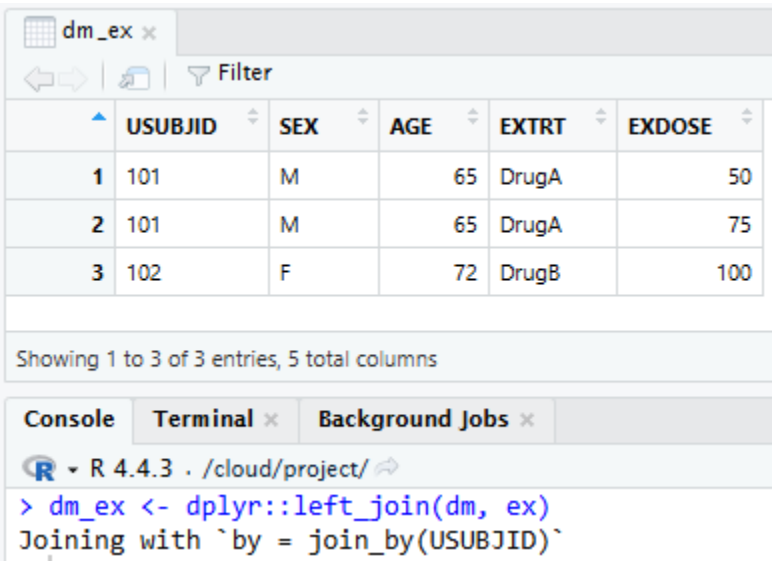

Figure 5: Output: left_join(dm,ex)

In this example, R automatically identifies the common variable (USUBJID) and preforms the join without requiring it to be explicitly defined, unlike traditional SAS programming that requires prior sorting and a BY statement.

Once the raw data are initially combined, the data may collapse and be sorted using specific group_by() functions, or selection logic to ensure compliance with SDTM structure requirements.

Factors are another useful R feature in the development of DM. Factors are used for categorical data with a fixed set of values such as SEX, RACE, or ETHNIC. They allow variables to be sorted with predefined levels. By explicitly defining factor levels, programmers can enforce standardized values and maintain consistent ordering.

```
dm <- dm |>
    dplyr::mutate(
        SEX = factor(SEX, levels = c("M", "F"))
    )
```

Program 16. Creating Factors in R

This converts SEX to a factor variable, using the dplyr::mutate() and the base::factor().The levels argument within the base::factor() function, takes a vector of the unique character values and ensures “M” always appears before “F” when sorting or printing. This reduces sorting logic and aligns derived variables with CDISC expectations for controlled terminology.

SDTM DEVELOPMENT: SE EXAMPLE

The Subject Elements (SE) domain is a “A special-purpose domain that contains the actual order of elements followed by the subject, together with the start date/time and end date/time for each element” (SDTMIG 5.3). The structure defined within SE highlights a subject's journey through the EPOCH variable.

The EPOCH variable helps define the phase of the study in which the observation occurred, such as SCREENING, TREATMENT, or FOLLOW-UP. Using conditional logic in R, these phases can easily be approached using dplyr::mutate() and dplyr::case_when() functions, allowing observations to be mapped to the appropriate study phase.

```
se <- se |>
    dplyr::mutate(
        EPOCH = dplyr::case_when(
            SESTDTC < TRTSDTC ~ “SCREENING”,
            SESTDTC >= TRTSDTC & SESTDTC <= TRTENDTC ~ “TREATMENT”,
            SESTDTC > TRTENDTC ~ “FOLLOW-UP”)
    )
```

Program 17. EPOCH Creation using {sdtm.oak}

Above, the EPOCH variable is created based on a series of conditions representing each treatment phase. Within dplyr::case_when(), the tilde (~) separates the condition from the value to be returned. If a condition on the left-hand side of the tilde is met, then the expression on the right-hand side is assigned to the EPOCH variable.

SDTM DEVELOPMENT: VS EXAMPLE

Vital Signs (VS) domain captures all measurements taken throughout the study. For domains of this type, it is often required to identify baseline measurements, which represent the last observation collected prior to subject treatment.

Additionally, variables such as VSORRESU must comply with the CDISC controlled terminology (CT). CT can be applied directly with the variable derivations using sdtm.oak::assign_ct(), given a controlled terminology file is uploaded into the work environment.

```
vs <- sdtm.oak::assign_ct(
    raw_dat = vs_raw,
    tgt_var = “VSORRESU”,
    raw_var = “orresu”,
    ct_spec = ct_file,
    ct_clst = "C71620"
)
```

Program 18. VSORRESU Creation using {sdtm.oak}

Within Program 18, the raw_dat argument represents the raw source data, with the raw_var (orresu) to be sourced from. The CT file is supplied through the ct_spec argument while ct_clst specifies the CDISC codelist (C71620). The tgt_var argument defines the name of the output variable (VSORRESU), which is created with CT applied and returned within the resulting dataframe (vs).

Once CT is applied baseline flags can be derived from helper functions available within {sdtm.oak}. The sdtm.oak::derive_blfl() function can be used to assign the baseline flag.

```
vs <- sdtm.oak::derive_blfl(
  sdtm_in = vs,
  dm_domain = dm,
  tgt_var = “VSBLFL”,
  ref_var = “RFSTDTC”,
  baseline_visits = character(“SCREENING”,”BASELINE”)
)
```

Program 19. VSBLFL Creation using {sdtm.oak}

The sdtm.oak::derive_blfl() derives a baseline flag by identifying records that meet predefined baseline criteria. In Program 19, the function references the treatment start date (RFSTDTC) from the dm_domain

(dm) and evaluates records within the sdtm_in dataset (vs) to determine which observations occur prior to or at the treatment start date.

The baseline_visits argument further restricts the selection to specific visits, in this case “SCREENING” and “BASELINE”. Records that meet these conditions are assigned a value of “Y” for VSBLFL, while all other records remain unflagged.

Additional specifications for identifying baseline records can be further defined using the baseline_timepoints argument, which specifies timepoints with the --TPT variable to identify more precise baseline measurements. This approach allows baseline records to be consistently identifiable while maintaining a transparent programming workflow.

SDTM DEVELOPMENT: AE EXAMPLES

Adverse Events (AE) domain records adverse events experienced by a subject. A key focus within an events domain is capturing the relationship of the event to the treatment exposure.

Dates play an important role in this process. In R, sdtm.oak::assign_datetime() simplifies the derivation of AESTDTC, deriving the ISO8601 date.

```
ae <- sdtm.oak::assign_datetime(
    tgt_dat = ae,
    tgt_var = "AESTDTC",
    raw_dat = raw_ae,
    raw_var = "startdate_raw",
    raw_fmt = "%Y-%m-%d",
    raw_unk = c("UN", "UNK"),
    id_vars = oak_id_vars()
)
```

Program 20. AESTDTC Creation using {sdtm.oak}

The function converts the raw start date values into ISO8601 format with the consideration of unknown date components. Each argument within this function plays a specific role. The tgt_dat parameter specifies the input dataset to be modified, which is merged with the raw_dat dataset containing the raw source values. The tgt_var defines the name of the SDTM variable to be created, in this case AESTDTC, and the raw_var is the original source variable containing the unformatted date values (startdate_raw).

The raw_fmt argument defines the expected format of the raw date values using the standard R date formatting conventions, where %Y represents a four-digit year, %m a two-digit month, and %d a two-digit day. These formats are part of base R and are used consistently across functions that handle dates.

The raw_unk argument specifies values that represents unknown or partial dates, such as “UN” and “UNK”, which are provided in a vector using the base::c() function. Finally, id_vars defines the key variables used to uniquely identify records, ensuring the derivation is applied correctly at the subject level.

As a result, AESTDTC is created into ISO8601 format and is compliant with CDISC standards. With SDTM datasets providing the standardized foundation, the next stage in workflow is the creation of Analysis Data Model (ADaM) datasets to support statistical analysis.

ADAM CREATION IN R

The ADaM datasets are the bridge between the standardized domains to producing tables, listings, and figures (TFLs). There is no single package that fully automates ADaM creation, but {admiral} provides a comprehensive framework that assists in programming. {admiral} is a collection of three different packages: one containing core functions for creating ADaM datasets, therapeutic area extensions, and company-specific package extensions.

{admiral} includes built in programming checks that support validation and consistency across datasets. Many functions confirm datatypes, verify required variables, and validate if variables are meeting expected formats before performing derivations. These internal checks help detect programming issues early and allow consistency across development, promoting a more standardized approach.

CREATION OF ADSL

The first step of ADaM development is the creation of the subject-level analysis dataset (ADSL). ADSL contains one record per subject and includes key demographic information, and a variety of population and analysis flags. These flags or indicator variables are critical because they define the analysis populations used in statistical summaries.

Using {admiral} these flags can easily be developed with the use of helper functions. admiral::derive_var_merged_exist_flag() can create subject-level flags based on the specific records in another dataset:

```
adsl <- admiral::derive_var_merged_exist_flag(
    dataset = adsl,
    dataset_add = ex,
    by_vars = USUBJID,
    new_var = SAFFL
)
```

Program 21. SAFFL Creation using {admiral}

This function checks if a subject exists within EX, then sets SAFFL to “Y”, otherwise it sets SAFFL to NA. Internally, admiral::derive_var_merged_exist_flag() preforms the merge by the specified key variables (USUBJID) and applies the conditional logic required to generate the flag.

This eliminates the need for manual merge steps, and explicit IF-THEN logic typically required in traditional programming workflows. As a result, complex subject-level flag derivations can be implemented in a concise, standardized manner, improving code readability and efficiency while reducing the risk of programming errors.

CREATION OF BDS DATASETS

BDS datasets contain one record per subject, parameter, and timepoint, resulting in multiple derived analysis variables such as changes from baseline and analysis flags. The standard calculation for CHG is AVAL – BASE, and this derivation can be manually implemented using dplyr::mutate():

```
advs <- advs |>
    dplyr::mutate(CHG = AVAL - BASE)
```

Program 22. CHG Manual Creation

Program 22 takes the advs dataset and creates CHG by calculating the difference between AVAL and BASE. This approach allows for greater flexibility in customization if needed.

Within the {admiral} framework, this derivation of CHG can also be performed in one function call using the admiral::derive_var_chg() function.

```
advs <- admiral::derive_var_chg(advs)
```

Program 23. CHG Creation using {admiral}

Similar to Program 22, Program 23 calculates the difference between the analysis value (AVAL) and the baseline value (BASE) within the dataset (advs). The resulting dataset includes the CHG variable, representing the change from baseline.

Internally, the function applies the standard calculation across all applicable records, eliminating the need for explicit programming of the calculation and conditional logic. This provides a concise and standardized approach to deriving commonly used analysis variables. By applying standardized derivation functions, programmers can consistently derive analysis values across multiple parameters and visits. This approach promotes reproducibility and ensures that commonly used analysis variables follow a consistent methodology across datasets.

CREATION OF OCCDS DATASETS

OCCDS datasets capture subject-level occurrences such as adverse events, where a record represents an event that was experienced by the subject. In an R based ADaM workflow, OCCDS datasets can be programmed using a combination of general data manipulation and functions available in {admiral}.

One of the most common derivations within an OCCDS dataset is the creation of a treatment emergent flag, the most common being the treatment emergent adverse event flag (TRTEMFL). This can be derived using derive_var_trtemfl():

```
adae <- adae |>
    admiral::derive_var_trtemfl(
        new_var = TRTEMFL,
        start_date = ASTDTM,
        end_date = AENDTM,
        trt_start_date = TRTSDTM
)
```

Program 24. TRTEMFL Creation using {admiral}

This function evaluates whether an event occurred during the treatment period and assigns TRTEMFL accordingly. Internally, the function applies conditional logic comparing the event start and end dates (ASTDTM, AENDTM) to the treatment start date (TRTSDTM) to determine if the event is treatment emergent.

Similar to a SAS macro, admiral::derive_var_trtemfl() uses predefined logic, where the input parameters define how the derivation is applied and then the function executes the programming steps automatically. This allows programmers to perform complex derivations without explicitly writing the full conditional logic in individual programs.

Like SAS, programmers still have the flexibility to define their own functions within a program and reuse them across programs when more customized logic is required. Additionally, a key benefit of open-source is that many commonly used functions have already been developed and shared by the community, reducing the need to build these derivations from scratch.

These examples demonstrate how packages in the {pharmaverse} allow clinical programmers to implement a fully reproducible CDISC workflow in R, from raw CRF through SDTM domains and into the analysis ready ADaM datasets.

CONCLUSION

The rapid growth of open-source technologies within the pharmaceutical industry is transforming how clinical trial data is prepared and analyzed for regulatory review. While the transition from SAS to R initially presents a learning curve, programming in R enables clean and reproducible clinical programming workflows.

Unlike traditional programming environments, R offers packages specifically designed to support CDISC standards. Instead of being a general-purpose tool like SAS, open-source offers packages intentionally developed for the pharmaceutical industry and its regulatory requirements. This provides a significant advantage, as many standardized solutions are already developed.

These packages and their functions can be customized to align with industry, company, or project specific standards. Additionally, package documentation in R is typically more comprehensive and accessible than macro documentation, providing clear examples and parameter descriptions. Packages within the {pharmaverse} such as {sdtm.oak} and {admiral} support common statistical programming tasks, allowing the user to focus back in on the data.

For experienced programmers, R offers modernization, streamlined data transformations, and advanced visualization capabilities. For early career programmers, it's function-based programming approach provides clear and structured code, allowing a user to quickly develop skills. Additionally, packages like {renv} allow users to control package versions and recreate programming environments, supporting reproducibility throughout the study lifecycle.

Together, R supports the full CDISC workflow from SDTM development through ADaM dataset creation.

REFERENCES

CDISC. Analysis Data Model Team "Analysis Data Model Implementation Guide (ADaMIG), Version 1.3." Clinical Data Interchange Standards Consortium, 29 November 2021. https://www.cdisc.org/standards/foundational/adam/adamig-v1-3

CDISC. Analysis Data Model Team "Analysis Data Model Structure for Occurrence Data Implementation Guide (OCCDS) Version 1.1." Clinical Data Interchange Standards Consortium, 29 November 2021. https://www.cdisc.org/standards/foundational/adam/adam-structure-occurrence-data-implementation-guide-v1-1

CDISC. Submission Data Standards Team. "Study Data Tabulation Model Implementation Guide: Human Clinical Trials Version 3.4" Clinical Data Interchange Standards Consortium, 29 November 2021 https://www.cdisc.org/standards/foundational/sdtmig/sdtmig-v3-4

CONTACT INFORMATION

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

Madeleine Penniston
Maddie.Penniston@AtorusReaserch.com

Alyssa Wittle
Alyssa.Wittle@AtorusResearch.com