Paper SM06

An Intuitive Way to Derive the Last Known Alive Date in R

Marckenley Mercie, Bristol Myers Squibb, Madison, NJ, USA
Diana Tai, Bristol Myers Squibb, Madison, NJ, USA

ABSTRACT

Accurate derivation of the Last Known Alive Date (LSTALVDT) is critical for the creation of CDISC-compliant ADaM datasets, as it directly influences key efficacy endpoints such as progression-free survival (PFS), overall survival (OS), and long-term follow-up metrics in clinical trials. Traditionally, SAS has been the predominant tool for generating analysis datasets. On the other hand, there is a growing trend in the industry towards adopting open-source solutions. This paper introduces a comprehensive R-based methodology leveraging tidyverse packages to derive LSTALVDT. Furthermore, we present a Shiny application designed to streamline this derivation process, thereby enhancing the efficiency and accuracy of dataset production and validation.

INTRODUCTION

The last known alive date (LSTALVDT) is a critical variable in clinical trials, particularly for time-to-event analyses. For example, when deriving overall survival (OS), LSTALVDT is often used as the analysis date (ADT) if the actual date of death is not available. In time-to-event adverse event (AE) analyses, LSTALVDT also helps define the ADT for censored participants by taking the minimum of the last known alive date and the treatment end date. Although there are many other ways in which LSTALVDT is vital to clinical data analysis, we will focus on why R is an excellent choice for calculating this variable. R is an open-source language with a rich ecosystem of packages that make data manipulation and statistical analysis straightforward. Powered by libraries like dplyr, tidyr, purrr, and more, R enables analysts to write clear, reusable functions that can handle the complexities of clinical data. These capabilities lend themselves well not only to static workflows but also to dynamic, interactive applications—thanks to Shiny, R's web application framework.

In the following sections, we will explore two approaches to deriving the last known alive date:

- An R function that can be incorporated into standard R scripts for analysis datasets.
- An interactive Shiny app that allows the user to calculate and review LSTALVDT on-the-fly.

ASSUMPTIONS AND CONSIDERATIONS

The methodology to derive the last known alive date (LSTALVDT) can vary across the industry. Our approach is outlined as follows:

"For subjects who died, and death date is complete, set LSTALVDT to the numeric value of the date part of the death date. For all other subjects, collect complete dates from all study related or follow-up procedures where a subject is determined to be alive. Derive LSTALVDT as the maximum date among all complete dates. If derived LSTALVDT > database cutoff/lock date, then set LSTALVDT to the database cutoff/lock date."

Based on this algorithm, two key assumptions were made:

1. SDTM-compliant input data: By adhering to SDTM standards, potential issues arising from unforeseen data inconsistencies are mitigated.
2. All date variables involved in the calculation are assumed to follow ISO 8601 format.

It is important to note that the algorithm relies exclusively on complete dates. For instance, certain domains, such as adverse events (AE), may include records with partial dates. This solution excludes such partial dates from the calculation. Considerations for handling date imputations will be discussed in the later sections. With these assumptions in place, the next step is to build the R code that will calculate LSTALVDT.

DERIVING LSTALVDT

This algorithm can be divided into three main steps. First, parse the date values and convert them into a numeric format. Second, for each domain, determine the maximum date for each subject. Third, consolidate the resulting datasets and identify the overall maximum among the merged date columns. In SAS, these tasks can be automated using macro functions. Likewise, in R, a comparable approach is achieved via user-defined functions. By leveraging the tidyverse ecosystem, one can write functions that are both flexible and easily readable.

While the steps outlined above may seem straightforward when dealing with only a few domains, the complexity increases when dates related to the last known alive date are distributed across 10 or more domains. One potential solution is to organize the required inputs into a file. However, manually creating this file introduces an additional burden. Fortunately, this file can be generated programmatically, allowing it to be used as input for subsequent functions.

PARAMETER FILE

For simplicity, the parameter file can be organized as a table with three columns. The *Domain* column lists each SDTM domain; the *Dates* column holds a comma-separated string of date variables for that domain; and the *Condition* column specifies any row filtering criteria that applies to its corresponding domain.

```
# Create a parameter file
create_param_file <- function(data_path) {
  datafiles <- list.files(data_path, pattern = "\\.sas7bdat$", full.names = TRUE)
  sdtm_list <- rlang::set_names(lapply(datafiles, haven::read_sas),
                                tools::file_path_sans_ext(basename(datafiles)))

  # Extract the date columns from an SDTM domain
  extract_dates <- function(domain_name, domain_data) {
    if ("QLABEL" %in% colnames(domain_data)) {
      # Supplemental domains
      filtered_data <- domain_data |>
        dplyr::filter(grepl("date", QLABEL, ignore.case = TRUE))
      date_columns <- unique(filtered_data$QNAM)
    } else {
      # Regular domains
      date_columns <- grep("(DT|DTC|DTM)$", colnames(domain_data), value = TRUE)
    }

    if (length(date_columns) > 0) {
      return(tibble::tibble(Dataset = domain_name,
                            Dates = paste(date_columns, collapse = ", ")))
    } else {
      return(NULL)
    }
  }

  date_info <- purrr::map2(names(sdtm_list), sdtm_list, extract_dates)

  sdtm_dates <- dplyr::bind_rows(purrr::compact(date_info)) |>
    dplyr::mutate(Condition = NA)

  writexl::write_xlsx(sdtm_dates, "lstalvdt_param_file.xlsx")

  list(sdtm_list = sdtm_list,
       sdtm_dates = sdtm_dates)
}
```

Sample Code 1: Function to Generate Parameter File

To generate the parameter file, a function can be created that specifies the directory containing the SDTM datasets. One of the key strengths of R lies in its capacity to efficiently handle repetitive tasks through the use of base R's apply family of functions and purrr's map functions, both of which enable concise solutions for iterating over data structures. As illustrated in Sample Code 1, the function, when combined with haven's *read_*sas, facilitates the concurrent loading of all SDTM datasets into a list. To identify the date columns, the *extract_dates* function returns a tibble, an R data frame object, with the domain name and the associated date columns. Given that the data follows the SDTM format, *extract_dates* searches for column names containing the suffixes "DTC", "DT", or "DTM". For supplementary domains, which are structured in long format, the function searches the Qualifier Variable Label (i.e., QLABEL) for the term 'date', then extracts the unique values from the Qualifier Variable Name (i.e., QNAM). While this function is designed to work with a single input dataset, we can apply it to all data frames in the list using purrr's *map2* function. Finally, to compile the results into a single table, we can use *dplyr::bind_rows* to set the list of data frames together, while removing any NULL values with the *purrr::compact* function.

Notice that the function returns a single list containing both the named list of SDTM domains and the parameter data frame. This ensures that the datasets do not need to be loaded into the environment again and provides the flexibility to use the parameter data frame directly to derive LSTALVDT. However, this assumes that no filter conditions are required and that every date in SDTM contributes to the last known alive date, which is often not the case. To address this, the static solution also outputs the parameter data frame as an .xlsx file, as shown in Figure 1, allowing the programmer to remove any dates or domains that do not contribute to the calculation and add any necessary filter conditions.

	A	B	C
1	Dataset	Dates	Condition
2	ae	AESTDTC, AEENDTC	
3	ce	CESTDTC	
4	cm	CMSTDTC, CMENDTC	
5	cv	CVDTC	
6	dd	DDDTC	
7	dm	RFSTDTC, RFENDTC, RFXSTDTC, RFICDTC, RFPENDTC, DTHDTC	
8	ds	DSSTDTC	
9	eg	EGDTC	
10	ex	EXSTDTC	
11	faae	FADTC	

Figure 1: Snippet of Parameter File Excel Output

DATE OBJECTS IN R

In SAS, date operations require conversion to the SAS numeric date type, a practice that is similarly applied in the R ecosystem, albeit with some important distinctions in date-time representations. In R, the primary representations for dates are the Date type, which represents calendar dates without any associated time information; POSIXct, which stores date-time values as the number of seconds since the Unix epoch (1970-01-01); and POSIXlt, which represents date-time as a list of individual components such as year, month, day, hour, minute, and second. The SDTM standards require date variables to be in ISO 8601 format, specifically YYYY-MM-DDThh:mm:ss. Since the focus is solely on the calendar date portion, the R Date type is selected for date representation. In Sample Code 2, a custom function is created to handle complete date values, since the *as.Date* function would throw an error when parsing ISO 8601 date strings with a partial calendar date. While the lubridate package is generally recommended for working with ISO 8601 date-time strings, the *as.Date* function is sufficient in this case, as it extracts only the date portion from ISO 8601-compliant strings, provided the date part is not incomplete.

```
# Convert a string to a Date
convert_date <- function(x) {
  ifelse(nchar(x) >= 10, as.Date(x, origin = "1960-01-01"), NA)
}
```

Sample Code 2: Date Conversion Function

Next, utilizing dplyr's column-wise operations, the second function, *convert_all_dates*, illustrated in Sample Code 3, applies the *convert_date* function across all specified date columns. To enhance reliability, a validation step is incorporated into *convert_all_dates* to alert the user if any date columns listed in the *date_vars* argument are missing from the domain. Now that we have a reliable method for converting the dates, we can proceed to find the maximum date value across these columns.

```
# Convert all dates columns in a data frame to Date format
convert_all_dates <- function(data, date_vars) {
  # Split the date columns from the list, ensuring they are in the right format
  dates <- unlist(strsplit(date_vars, ",|(, )"))
  valid_dates <- intersect(dates, names(data))
  missing_col <- setdiff(dates, valid_dates)

  # Check for missing columns
  if (length(missing_col) > 0) {
    message(paste(
      "The following date columns are not in the dataset and will not be used to calculate
LSTALVDT:",
      paste(missing_col, collapse = ", ")
    ))
  }
  # Convert date columns in the data
  data |>
    dplyr::mutate(across(all_of(valid_dates), ~convert_date(.)))
}
```

Sample Code 3: Function to Convert Multiple Date Columns

MAXIMUM BETWEEN DATES

To determine the maximum date for each subject, we developed a custom R function*, find_max_date*, as shown in Sample Code . This function dynamically computes the maximum date across user-specified date columns and optionally applies row-level filters. It accepts the following parameters:

- *data*: The SDTM dataset containing the USUBJID and the relevant date columns.
- *domain_name*: A string that specifies the domain name for traceability.
- *date_vars*: A comma-separated list of date variable names, analogous to the argument used in *convert_all_dates*.
- *filter_cond*: An optional R-syntax filter expression (e.g., *"ARMCD == 'XYZ' "*) that can be applied to subset the data before computing the maximum date.

The function leverages the rlang package to dynamically construct and evaluate user-specified column and filter expressions at runtime. The *date_vars* input is parsed into individual column names, converted into symbols via *rlang::syms*, and then spliced into an expression that calculates the maximum date among these columns (with *na.rm = TRUE* to ignore missing values). If a filtering condition is supplied, it is transformed into an R expression using *rlang::parse_expr* and applied using *dplyr::filter* to include only the relevant rows.

```
# Find the maximum date in a data frame
find_max_date <- function(data,
                          domain_name,
                          date_vars,
                          filter_cond = NULL) {

  dates <- unlist(strsplit(date_vars, ",|(, )"))
  valid_dates <- intersect(dates, names(data))
  max_expr <- rlang::expr(max(!!!rlang::syms(valid_dates), na.rm = TRUE))

  if (!is.null(filter_cond) & !is.na(filter_cond)) {
    filter_expr <- rlang::parse_expr(filter_cond)
    data <- data |>
      dplyr::filter(!!filter_expr)
  }

  data |>
    dplyr::group_by(USUBJID) |>
    dplyr::summarise(max_date = !!max_expr) |>
    dplyr::filter(!is.infinite(max_date)) |>
    dplyr::mutate(source_data = stringr::str_to_upper(domain_name))
}
```

Sample Code 4: Function to Find the Maximum Between Dates

Following any optional filtering, the function groups the dataset by *USUBJID* and uses *dplyr::group_by* and *dplyr::summarise* to compute the maximum date. It then appends a *source_data* column to record the originating domain, stored in uppercase. This design allows users to specify an arbitrary number of date columns for a given domain, rather than computing the maximum for each date column individually. An important consideration is the handling of missing values. Although *na.rm = TRUE* instructs the max function to ignore missing dates, questions remain regarding the scenario in which *DATE1*, *DATE2*, *DATE3*, or any additional date columns are all missing for the same record.

	USUBJID	max_date	source_data
1	MODELSTUDY-12-016-26106	-Inf	PP

Figure 2: Negative Infinity Value for *max_date*

To illustrate, consider the Pharmacokinetic Parameters domain in which the goal is to determine the maximum between the Date/Time of Parameter Calculations (PPDTC) and the Date/Time of Reference Point (PPRTDTC). In the scenario depicted in Figure 2, the subject in question has missing values for both date variables. When using the *max* function with *na.rm = TRUE* under these circumstances, R assigns negative infinity (-Inf) as the result. To mitigate potential downstream issues, the implementation can be adapted to discard columns or rows containing negative infinity, thereby ensuring the integrity of subsequent calculations.

LSTALVDT FUNCTION

In the preceding sections, we examined various date conversion methods and demonstrated how to determine the maximum value across date columns. The next step involves deriving LSTALVDT. While the functions introduced thus far were designed to handle only a single input dataset, it is often necessary to process multiple datasets iteratively without invoking each function individually. To address this requirement, we propose an overarching function, *lstalvdt*, which operates on the following inputs:

- *param_list*: A list of SDTM datasets and the parameter data frame (assuming that *create_param_file* has already been executed).
- *paramfile_path*: The directory containing the updated parameter file if it exists.
- *cuttoff_date*: The cutoff date for the study.
- *output_path*: The directory where the final dataset will be exported.

To facilitate the iterative processing of multiple datasets, the *imap* function from the purrr package was employed. This function provides the ability to operate on both the value and the position (or name) of each list element, mirroring the behavior of *purrr::map2(x, names(x), ...).* Such functionality proves especially advantageous when list names carry meaningful information—such as dataset-specific column names or filtering criteria. Sample Code 5 shows this approach in detail. In the first step, the *convert_all_dates* function is applied to each SDTM dataset, transforming the relevant date columns. Here, the dataset name is automatically used to identify and extract the corresponding date columns from *param_df$Dates*, ensuring that each dataset is processed according to its specific requirements. The second step applies the *find_max_date* function to compute the maximum date across the date columns for each dataset, using filtering conditions derived from *param_df$Condition* with the dataset name as a key. Finally, once all datasets have been processed, *dplyr::bind_rows* combines the outputs into a single dataset, thereby consolidating the maximum date information across all subjects and all datasets.

```
dates_formatted <- sdtm_list |>
  purrr::imap(~ {
    dates <- param_df$Dates[param_df$Dataset == .y]
    convert_all_dates(.x, dates)
  })

max_dates <- dates_formatted |>
  purrr::imap(~ {
    dates <- param_df$Dates[param_df$Dataset == .y]
    filter_cond <- param_df$Condition[param_df$Dataset == .y]
    find_max_date(.x, .y, dates, filter_cond)
  }) |>
  dplyr::bind_rows()
```

Sample Code 5: *imap* calls in *lstalvdt* Function

Following the creation of the dataset containing maximum dates, derivation of the Last Alive Date (LSTALVDT) is finalized by incorporating both the Death Date (DTHDTC) and the database lock or cutoff date. The procedure begins by calculating an intermediate date, defined as the minimum of the maximum date and the cutoff date, i.e., computing *min(max(max_date), cutoff_date)* within a *dplyr::group_by* and *dplyr::summarise* step. Subsequently, a left join is performed with the dataset that contains the death date (commonly the Demographics domain), and any non-missing DTHDTC values are used to replace the corresponding entries in the intermediate date step. This approach is illustrated in Sample Code 6. The *lstalvdt* function returns a list object that contains both the traceable *max_dates* dataset and the final lstalvdt data frame. Furthermore, the lstalvdt data frame can be exported as an .xpt file for downstream use in SAS, facilitating seamless integration into production and validation workflows.

```
lstalvdt_df <- max_dates |>
  dplyr::group_by(USUBJID) |>
  dplyr::summarise(INTERMDT = min(max(max_date), cutoff_date, na.rm = TRUE),
                   .groups = "keep") |>
  dplyr::left_join(dates_formatted[[dataset_with_dthdtc]] |>
                     dplyr::select(USUBJID, DTHDTC), by = "USUBJID") |>
  dplyr::mutate(LSTALVDT = as.Date(ifelse(!is.na(DTHDTC), DTHDTC, INTERMDT))) |>
  dplyr::select(USUBJID, LSTALVDT) |>
  dplyr::arrange(USUBJID)
```

Sample Code 6: Final Derivation of LSTALVDT

FURTHER AUTOMATION

A Shiny application was developed to further streamline the process of LSTALVDT derivation. With this application, even users without any knowledge of R programming can interact with the functions that have been detailed in this paper. The app is split into two sections, which have been implemented as two R modules. The first module creates the parameter file, while the second module derives last known alive date.

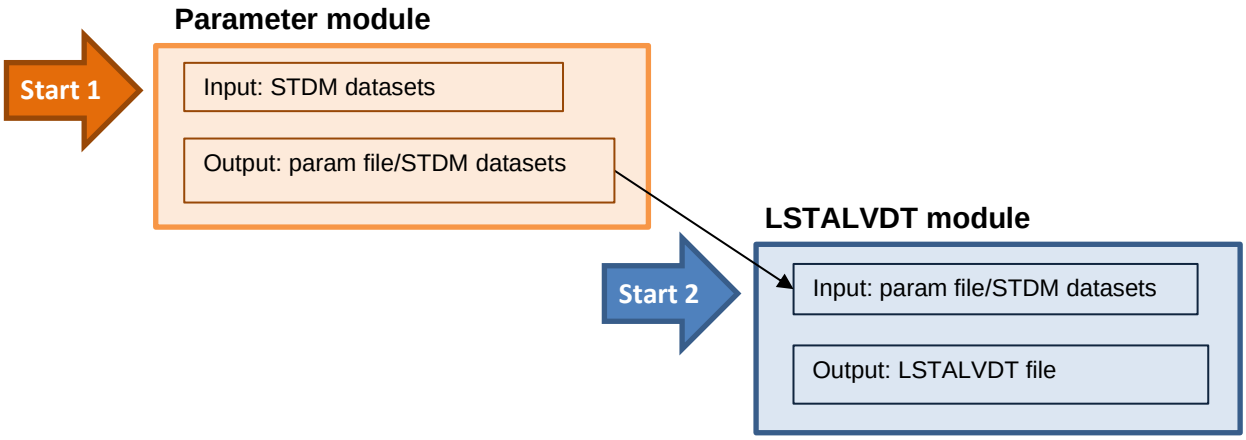

Figure 3: Diagram of the Shiny Application

The interaction between the modules is detailed in Figure 3. The first parameter module takes in STDM files as input and outputs a parameter file as well as the original STDM files. The second module takes in those outputs as the input and returns the final LSTALVDT file. These two modules correspond to two tabs in the application: “Parameter File Creation” and “LSTALVDT Derivation”. The user has two options: to either start in the first parameter tab, or to go directly to the LSTALVDT tab and supply those inputs directly.

To utilize the entire application, the user should start in the “Parameter File Creation” tab and upload the STDM datasets there. This is done with help from the shinyFiles package, which contains functions that aid in file directory navigation. After a folder with the required files is selected, the *create_param_file* function, which has been modified to fit within the Shiny framework, will run. The parameter data frame will be displayed in a datatable (using the DT package) as shown in Figure 4 while the list of SDTM domains will be saved in the background.

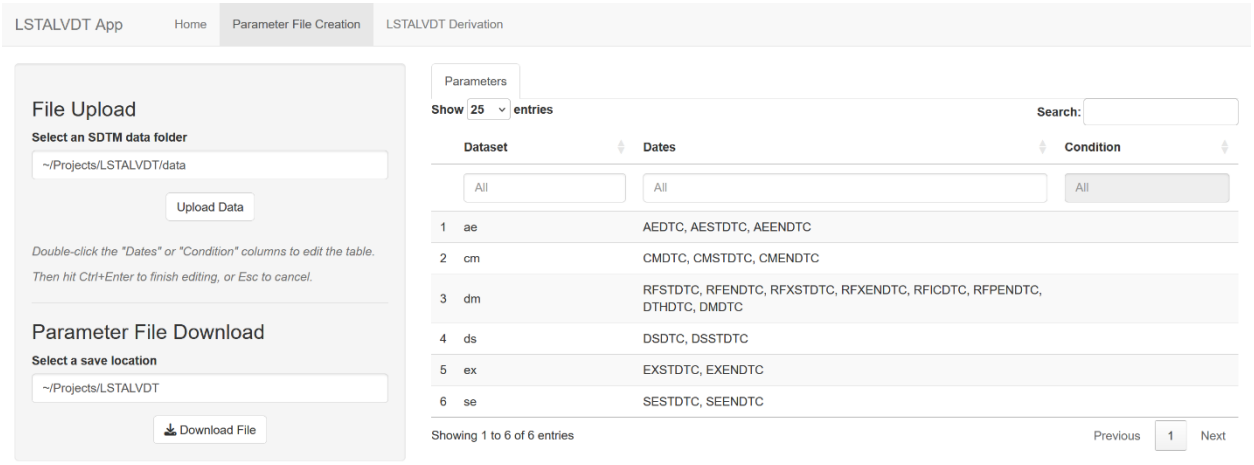

Figure 4: Screenshot of the “Parameter File Creation” tab

The parameter file can then be downloaded to a user-specified location as an .xlsx file. This can be edited to modify any domains and corresponding date columns or to add filter conditions. However, there is another option facilitated by the nature of an interactive Shiny application, because the DT package can produce editable tables. By taking advantage of this option, the *Dates* and *Condition* columns can be modified within the app itself, removing the need to move to a secondary program for editing.

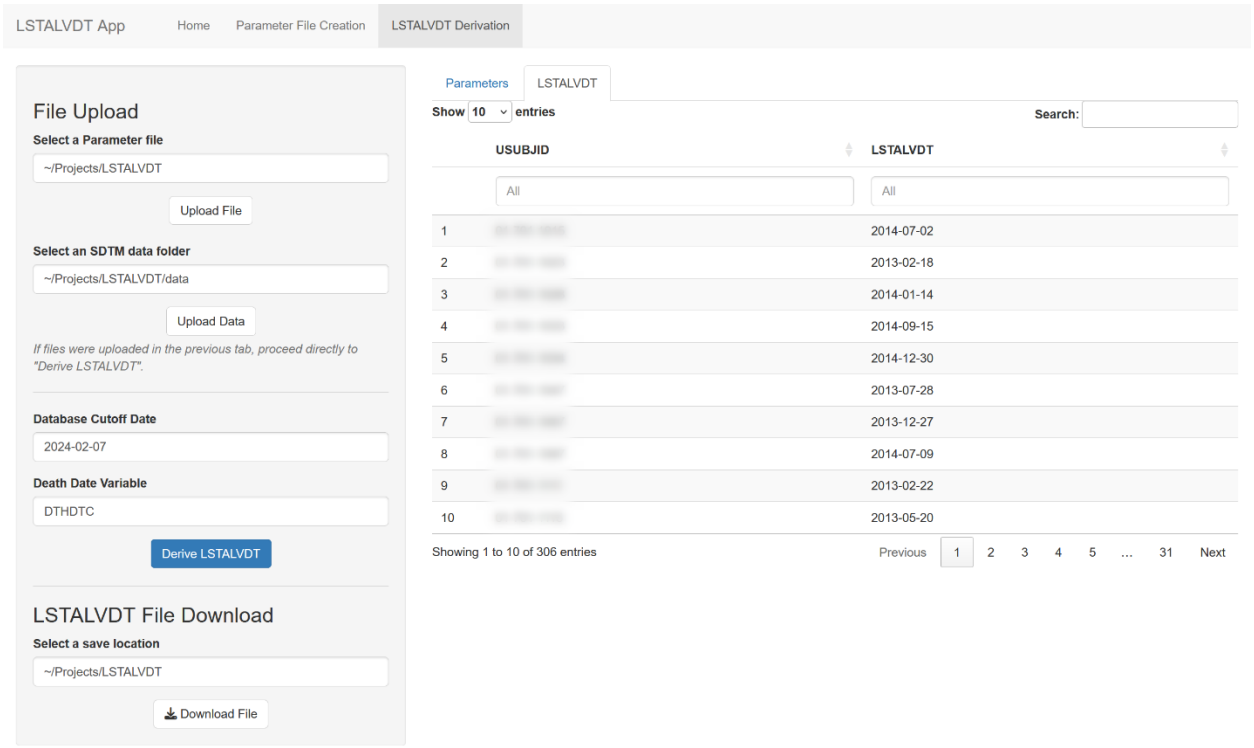

Figure 5: Screenshot of the “LSTALVDT Derivation” tab

Once the parameter file is ready, the user can move to the “LSTALVDT Derivation” tab. If a parameter file was created in the previous tab, then that data frame (and any edits made), as well as the saved STDM domains will be ready to be used in the next step. The user can simply click the “Derive LSTALVDT” button, and the *lstalvdt* function, with app-specific modifications, will run. The final data frame will be displayed as depicted in Figure 5 and can be downloaded as an .xpt file.

The other option is to start directly in the “LSTALVDT Derivation” tab. This route should be chosen if an external parameter file has been prepared, which is usually the case if the file was downloaded from the parameter tab in a prior session. In this case, both the parameter file as well as the SDTM datasets must be uploaded. Then the user can proceed as previously detailed and click the button to finish the derivation. Overall, the Shiny application provides an alternate and interactive way of deriving the last known alive date.

OTHER CONSIDERATIONS

In the algorithm used to derive the last known alive date, only complete dates are considered. However, there are scenarios where date imputation may be necessary. For instance, as detailed in Table 1, a subject in the Adverse Event (AE) domain has both a complete and a partial date. Specifically, the subject has start dates for two separate AE events, with one date being partial. While it is clear that the later event occurred sometime in 2023, our current implementation would fail to capture this event. Depending on the risk associated with these data points, imputing dates in such cases should be considered to ensure more accurate results.

USUBJID	**AEDECOD**	**AESTDTC**
MODELSTUDY-01-100101	Prostate Cancer	2022-01-05
MODELSTUDY-01-100101	Wound	2023-02

Table 1: Partial Dates in AE domain

There are areas where improvements can be made in the derivation of the last known alive date discussed in this paper. One key area is improving traceability. In the solution presented earlier, we included a traceability variable, *source_data*, in the intermediate dataset. However, this only indicates the domain from which the maximum date is derived. Additionally, the intermediate dataset can contain multiple records, making manual validation challenging. To enhance the implementation, we propose processing additional traceability columns for inclusion in the final derived dataset. These columns would not only indicate the domain but also specify the name of the date column from which the date was derived, along with any associated sequence numbers for the subject.

For those looking to avoid implementing traceability columns and date imputations from scratch, the admiral package—an R package designed for ADaM dataset generation—offers built-in options to derive the last known alive date. Further details on these options can be found in the package reference page, which will be omitted here for brevity.

CONCLUSION

The derivation of the last known alive date (LSTALVDT) is a cornerstone of time-to-event analyses in clinical trials, with wide-reaching implications for endpoints like progression-free survival and overall survival. In this paper, we introduced two approaches for calculating LSTALVDT in R: a static method leveraging tidyverse packages for flexibility and reproducibility, and an interactive Shiny application for enhanced usability and efficiency. The static R scripts demonstrate the power of tidyverse in handling complex data transformations, while the Shiny application bridges the gap between technical and non-technical users by providing a user-friendly interface to streamline dataset production and validation. We also explored areas for potential enhancement, including the incorporation of traceability columns and handling partial dates through imputation strategies. These considerations highlight opportunities for improving accuracy, transparency, and reliability in LSTALVDT derivation workflows. In conclusion, this work underscores the value of open-source solutions in advancing clinical data programming.

APPENDIX

```
# Create a parameter file
create_param_file <- function(data_path) {
  datafiles <- list.files(data_path, pattern = "\\.sas7bdat$", full.names = TRUE)
  sdtm_list <- rlang::set_names(lapply(datafiles, haven::read_sas),
                                tools::file_path_sans_ext(basename(datafiles)))

  # Extract the date columns from an SDTM domain
  extract_dates <- function(domain_name, domain_data) {
    if ("QLABEL" %in% colnames(domain_data)) {
      # Supplemental domains
      filtered_data <- domain_data |>
        dplyr::filter(grepl("date", QLABEL, ignore.case = TRUE))
      date_columns <- unique(filtered_data$QNAM)
    } else {
      # Regular domains
      date_columns <- grep("(DT|DTC|DTM)$", colnames(domain_data), value = TRUE)
    }

    if (length(date_columns) > 0) {
      return(tibble::tibble(Dataset = domain_name,
                            Dates = paste(date_columns, collapse = ", ")))
    } else {
      return(NULL)
    }
  }

  date_info <- purrr::map2(names(sdtm_list), sdtm_list, extract_dates)

  sdtm_dates <- dplyr::bind_rows(purrr::compact(date_info)) |>
    dplyr::mutate(Condition = NA)

  writexl::write_xlsx(sdtm_dates, "lstalvdt_param_file.xlsx")

  list(sdtm_list = sdtm_list,
       sdtm_dates = sdtm_dates)
}
```

Appendix 1: Creating the Parameter File

```
# Convert a string to a Date
convert_date <- function(x) {
  ifelse(nchar(x) >= 10, as.Date(x, origin = "1960-01-01"), NA)
}

# Convert all dates columns in a data frame to Date format
convert_all_dates <- function(data, date_vars) {
  # Split the date columns from the list, ensuring they are in the right format
  dates <- unlist(strsplit(date_vars, ",|(, )"))
  valid_dates <- intersect(dates, names(data))
  missing_col <- setdiff(dates, valid_dates)

  # Check for missing columns
  if (length(missing_col) > 0) {
    message(paste(
      "The following date columns are not in the dataset and will not be used to calculate
LSTALVDT:",
      paste(missing_col, collapse = ", ")
    ))
  }
  # Convert date columns in the data
  data |>
    dplyr::mutate(across(all_of(valid_dates), ~convert_date(.)))
}
```

Appendix 2: Date Conversions

```
# Find the maximum date in a data frame
find_max_date <- function(data,
                          domain_name,
                          date_vars,
                          filter_cond = NULL) {

  dates <- unlist(strsplit(date_vars, ",|(, )"))
  valid_dates <- intersect(dates, names(data))
  max_expr <- rlang::expr(max(!!!rlang::syms(valid_dates), na.rm = TRUE))

  if (!is.null(filter_cond) & !is.na(filter_cond)) {
    filter_expr <- rlang::parse_expr(filter_cond)
    data <- data %>%
      dplyr::filter(!!filter_expr)
  }

  data |>
    dplyr::group_by(USUBJID) |>
    dplyr::summarise(max_date = !!max_expr) |>
    dplyr::filter(!is.infinite(max_date)) |>
    dplyr::mutate(source_data = stringr::str_to_upper(domain_name))
}
```

Appendix 3: Maximum Between Dates

```
lstalvdt <- function(param_list,
                     paramfile_path = NULL,
                     cutoff_date = as.Date("2024-02-07", origin = "1960-01-01"),
                     output_path) {
  # Load all datasets
  sdtm_list <- param_list$sdtm_list

  if (!is.null(paramfile_path)) {
    param_df <- readxl::read_excel(paramfile_path) |>
      dplyr::arrange(Dataset)
  }
  else {
    param_df <- param_list$sdtm_dates |>
      dplyr::arrange(Dataset)
  }
  # Find death date dataset
  contains_dthdtc <- sapply(sdtm_list, function(df) "DTHDTC" %in% names(df))
  dataset_with_dthdtc <- names(sdtm_list)[contains_dthdtc]

  # Only one domain should contain DTHDTC, if more than one contains DTHDTC throw error
  if (length(dataset_with_dthdtc) > 1) {
    stop("Error: More than one dataset contains the 'DTHDTC' variable.")
  }
  # Get domains only specified in param_file
  sdtm_list <- sdtm_list[names(sdtm_list) %in% param_df$Dataset]

  # Transpose Supp Domains
  transpose_supp <- function(domain, data) {
    if (grepl("^SUPP", domain, ignore.case = TRUE)) {
      data |>
        tidyr::pivot_wider(id_cols = c(STUDYID, USUBJID, IDVAR, IDVARVAL),
                           names_from = QNAM,
                           values_from = QVAL)
    }
    else {
      data
    }
  }
  sdtm_list <- purrr::imap(sdtm_list, ~transpose_supp(.y, .x))

  dates_formatted <- sdtm_list |>
    purrr::imap(~ {
      dates <- param_df$Dates[param_df$Dataset == .y]
      convert_all_dates(.x, dates)
    })

  max_dates <- dates_formatted |>
    purrr::imap(~ {
      dates <- param_df$Dates[param_df$Dataset == .y]
      filter_cond <- param_df$Condition[param_df$Dataset == .y]
      find_max_date(.x, .y, dates, filter_cond)
    }) |>
    dplyr::bind_rows()

  lstalvdt_df <- max_dates |>
    dplyr::group_by(USUBJID) |>
    dplyr::summarise(INTERMDT = min(max(max_date), cutoff_date, na.rm = TRUE),
                     .groups = "keep") |>
    dplyr::left_join(dates_formatted[[dataset_with_dthdtc]] |>
                       dplyr::select(USUBJID, DTHDTC), by = "USUBJID") |>
    dplyr::mutate(LSTALVDT = as.Date(ifelse(!is.na(DTHDTC), DTHDTC, INTERMDT))) |>
    dplyr::select(USUBJID, LSTALVDT) |>
    dplyr::arrange(USUBJID)

  haven::write_xpt(lstalvdt_df,
                   file.path(output_path, "lstalvdt.xpt"),
                   version = 5)
  list(Max_dates = max_dates,
       LSTALVDT_df = lstalvdt_df)
}
```

Appendix 4. LSTALVDT Function

ACKNOWLEDGEMENTS

Many thanks to the Bristol Myers Squibb Cell-Therapy, Hematology, and RWE statistical programming teams for their support and encouragement.

CONTACT INFORMATION

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

Marckenley Mercie
Bristol Myers Squibb
Email: marckenley.mercie@bms.com

Diana Tai
Bristol Myers Squibb
Email: diana.tai@bms.com