Provenance metadata for statistical data: An introduction to Structured Data Transformation Language (SDTL)

George Alter[1], Darrell Donakowski[1], Jack Gager[2], Pascal Heus[2], Carson Hunter[2], Sanda Ionescu[1], Jeremy Iverson[3], H V Jagadish[1], Carl Lagoze[1], Jared Lyle[1], Alexander Mueller[1], Sigbjorn Revheim[4], Matthew A. Richardson[1], Ornulf Risnes[4], Karunakara Seelam[1], Dan Smith[3], Tom Smith[5], Jie Song[1], Yashas Jaydeep Vaidya[1], Ole Voldsater[4]

Abstract

Structured Data Transformation Language (SDTL) provides structured, machine actionable representations of data transformation commands found in statistical analysis software. The Continuous Capture of Metadata for Statistical Data Project (C^{2}Metadata) created SDTL as part of an automated system that captures provenance metadata from data transformation scripts and adds variable derivations to standard metadata files. SDTL also has potential for auditing scripts and for translating scripts between languages. SDTL is expressed in a set of JSON schemas, which are machine actionable and easily serialized to other formats. Statistical software languages have a number of special features that have been carried into SDTL. We explain how SDTL handles differences among statistical languages and complex operations, such as merging files and reshaping data tables from “wide” to “long”.

Keywords

Metadata, provenance, statistical data

Acknowledgment

Acknowledgement: The Continuous Capture of Metadata for Statistical Data Project is funded by National Science Foundation grant ACI-1640575.

Introduction

Structured Data Transformation Language (SDTL) is a language for representing data transformation commands found in statistical analysis and data management software. SDTL describes changes to a dataset at both the file- and variable-level. Since SDTL is structured and machine actionable, it can be queried to produce histories of each variable in a dataset and to answer questions like:

- Which original variables were used to construct this derived variable?
- Which commands were used in the construction of this derived variable?
- Which derived variables were affected by this original variable?

SDTL can be translated into natural language, so that researchers do not need to understand the specific software used to process the data. SDTL can also be incorporated into versions of the PROV model that have been extended to describe provenance at the variable- and command-level, like ProvONE (Cuevas-Vicenttín et al., 2015).

SDTL extends the capabilities of tools used by data repositories and data producers to document and describe data. Data repositories specializing in the social sciences maintain documentation in the Data Documentation Initiative (DDI) metadata standard (Vardigan, 2008). Online data catalogs draw upon content stored in DDI XML, and codebooks are translated from XML into PDF or other formats. DDI includes features for recording data provenance, but there was no systematic way to describe data provenance before SDTL. Variable histories expressed in SDTL can be formatted, searched, and queried. Data users who download DDI XML from repositories can use automated tools to create new codebooks reflecting their changes to the data. By building SDTL into their workflows, data producers can use SDTL to create documentation and to audit the command scripts that manage their data. In the future, SDTL may also be used to translate command scripts from one statistical analysis package to another.

SDTL was developed to work with five leading statistical packages: SPSS, Stata, SAS, R, and Python (IBM Corp., 2019; Python Software Foundation, 2019; R Core Team, 2013; SAS Institute, 2015; StataCorp., 2020). The Continuous Capture of Metadata for Statistical Data (C^2Metadata) Project, which created SDTL, set out to automate the creation of variable-level provenance metadata by translating scripts used by statistical analysis software into a format compatible with metadata standards like the Data Documentation Initiative (DDI) (Vardigan, 2008) and Ecological Metadata Language (EML) (E.H. Fegraus, 2005). (See Alter et al., 2020.) Our goal was to create a history for each variable showing its derivation from earlier variables and all of the ways that it has been modified. SDTL serves as an intermediate language that represents other languages in a more convenient format. Since SDTL is expressed in a structured format (e.g., JSON) with tags and delimiters, its syntax is obvious and unambiguous, and SDTL is easily read by computer programs without elaborate parsing algorithms. SDTL may be used to translate between statistical languages, but it is designed for documentation and description and not as an operational language.

The DDI Alliance, which maintains international standards for metadata, has adopted SDTL as one of its suite of products (DDI Alliance, 2020). DDI metadata is widely used by data repositories serving the social sciences for data discovery tools, catalogs, and codebooks. SDTL provenance descriptions can be inserted into existing data derivation fields in the DDI metadata standards. The DDI Alliance will assure that the SDTL is maintained and expanded in an orderly way.

We provide here an introduction to SDTL focusing on important features of the source languages that it can represent and ways that SDTL handles differences among them. A User Guide and detailed descriptions of SDTL commands are available at C2Metadata Project (2020b).

Statistical Packages as Data Processing Platforms

SDTL inherits a number of assumptions about how data are transformed from the languages used in statistical analysis packages, and it is helpful to understand how those programs work. Statistical packages differ in important ways from two other tools often used for managing data, spreadsheets and relational databases, such as SQL. All three tools encourage users to think of data as rectangular matrices, "tables," but they each have different capabilities and limitations.

1. Rows and Columns
   In statistical packages, each row is an individual/entity or an observation of an individual/entity, and each column is a variable describing an attribute of an individual/entity.   As in a relational database, columns are named and are referenced by their variable names. Statistical packages typically do not allow users to put more than one type of information in each column as a spreadsheet does.

2. Metadata
   Statistical packages attach more metadata to each variable than either a spreadsheet or a relational database, even if it is less metadata than most researchers need. Users control the data type (numeric, string, date, etc.) and display format of every variable. Columns can have both variable and value labels that appear on output. A variable label is a brief description of its content. Value labels are text descriptions of the categories in variables. Statistical packages encourage the use of integer codes for categorical information, but they will display the corresponding value label in output tables. For example, a variable may be coded as 1 for ages 0 to 15, 2 for ages 15 to 65, and 3 for ages 65 and above, but tables can show these categories with labels "Children," "Working ages," and "Older ages."

3. Variable lists and variable ranges
   One of the most common features of statistics software packages is the use of "variable lists" and "variable ranges" to simplify the application of data transformation commands to multiple variables. For example, common value labels (e.g. 1="Yes", 2="No", 3="NA") may be applied to hundreds of variables with a single command. Variable ranges refer to a group of adjacent columns by identifying the first and last variable in the range. For example, "VAR01 TO VAR04" in SPSS or "VAR01-VAR04" in Stata or SAS will apply a command to VAR01, VAR02, VAR03, and VAR04, assuming that the columns appear in that order in the data. A variable list may include both individual variables and variable ranges, such as "VAR01 VAR05 VAR11-VAR32 VAR51-VAR72".

4. Order of rows and columns matters
   Commands in statistical packages can take advantage of the order of columns and rows. The meaning of a variable range, such as “Age TO Income” depends upon the order of the columns. Statistical packages also process rows in sequential order, which is often used in data processing scripts. Commands that merge files or aggregate within groups may only operate on data sorted in advance. Statistical packages can also use values from earlier or later rows in computing variables. For example, if the data consist of annual observations of a country or region, the SPSS LAG function can be used to access the value in the previous year. In Stata, a command can test whether the current row applies to the same person or place as the preceding row by using syntax like “if districtID == districtID[_n-1]”. This is not possible in SQL relational databases, which do not permit operations that depend on the sequential ordering of rows, but it is possible in spreadsheets, which allow both absolute and relative cell references.

   SDTL includes a command (SortCases) to change the order of rows, but it does not currently support a command to change the order of columns. When we tested commands that sort columns in SPSS and Stata, we discovered that they apply different sort sequences to variable names. Stata is case sensitive and sorts variable names in ASCII order. SPSS is not case sensitive for variable names, but it sorts names beginning with lowercase before uppercase of the same letter. For example,

   SPSS sort order: aa8 aA9 Aa7 AA6 id xx3 xX4 Xx2 XX1 XX5
   Stata sort order: AA6 Aa7 XX1 XX5 Xx2 aA9 aa8 id xX4 xx3

5. Missing values
   The value of a variable may not be available for all rows, and all statistical packages have features for handling these “missing values.” Statistical calculations may exclude cases with missing values on any variable, or they may adjust for missing values in some way. Some statistical packages also allow users to identify more than one type of missing value. In survey research some questions do not apply to all respondents (e.g. “How many years have you been married?”), and respondents may respond “don’t know” or simply refuse to answer. Researchers need to distinguish between “does not apply”, “don’t know”, and “no response”.

   There are also important differences among statistical packages in the ways that missing values in logical expressions are processed. SPSS and R use three-valued logic in which a logical expression may be true, false or missing. SAS and Stata use two-valued logic (true or false) by processing missing values as either negative or positive infinity. Thus, if the value of varX is missing, the logical expression “varX > 0” will be false in SAS but true in Stata.

6. Dataframes and files
   When a statistical package is in operation, data may exist only in computer memory or in temporary storage space. A data transformation script may create any number of temporary instances of the data and save only a few of them for later use. The C^2Metadata Project adopted the convention of

using "dataframe" for working versions of data stored in memory or temporary storage to distinguish them from data in files that will persist after the transformation script is completed.

Elements of SDTL

The elements of SDTL, called "types," are divided into groups as shown in Figure 1. Commands are found in *CommandBase*, which is divided into two parts: *TransformBase* for commands that change data or metadata and *InformBase* for types that generate messages or comments. *ExpressionBase* consists of elements used to construct numeric, text, or logical expressions within commands. Types that describe variables are in *VariableReferenceBase*, which is a sub-category of *ExpressionBase*. The last group in Figure 1, "types for complex properties," is used when a property of a type has more than one sub-property. The "bases" shown in Figure 1 are hierarchical, and types inherit properties from higher levels. For example, the *messageText* property is available to all types in *CommandBase*.[6] Tables 1-5 list the SDTL types under the headings shown in Figure 1.

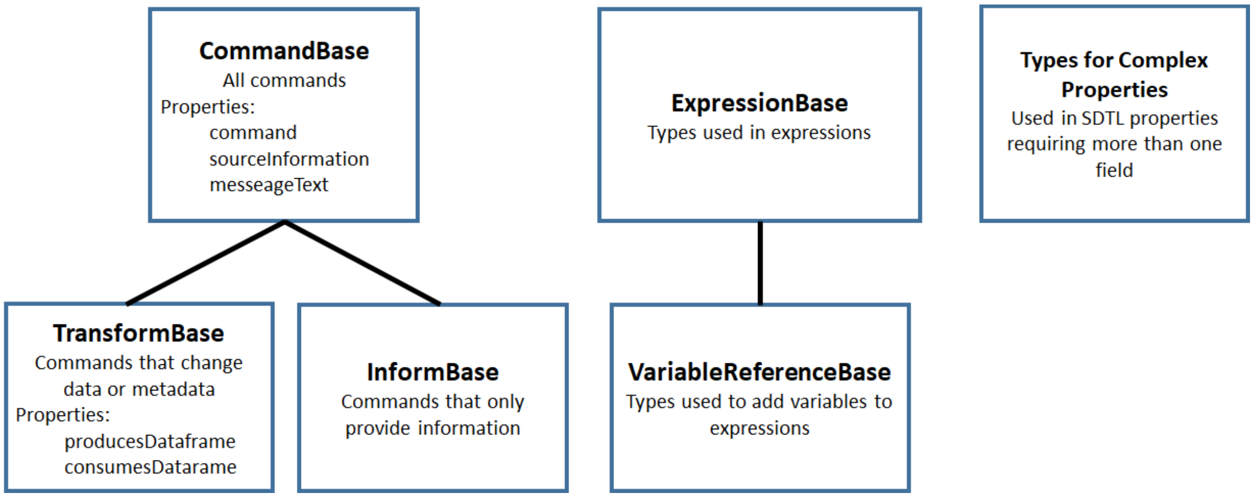

CommandBase
All commands
Properties:
command
sourceInformation
messeageText
ExpressionBase
Types used in expressions
Types for Complex Properties
Used in SDTL properties requiring more than one field
TransformBase
Commands that change data or metadata
Properties:
producesDataframe
consumesDatarame
InformBase
Commands that only provide information
VariableReferenceBase
Types used to add variables to expressions

FIGURE 1. SDTL TYPES HIERARCHY

TransformBase

The types belonging to *TransformBase* are commands that change data or provide information about the data to a user. All commands in *TransformBase* also inherit properties from *CommandBase*:

Properties inherited from *CommandBase*:

command	The name of a command
sourceInformation	Information about the source of the transform command.
message	Adds a message that can be displayed with the command.

Properties inherited from *TransformBase*:

producesDataframe	Identifies the dataframe which this transform produces.
consumesDataframe	Identifies the dataframe which this transform acts upon.

In Table 1 the commands in *TransformBase* are arranged into six functional sub-groups. There are only four SDTL commands that add or modify variables without changing the structure of a dataframe (group A), and *Compute* and *Recode* are by far the most frequently used in data transformation scripts. *Compute* assigns the value of an expression to a variable. *Recode* converts a continuous variable into categories.

Commands in group B operate only on metadata (names, labels, data type, display properties).

Load and *Save* in group C read and write data from files into dataframes.

The commands in group D modify the structure of a dataframe by changing the number of rows or columns.

Commands that control the execution of a script (group E) are discussed below.

Table 1

TransformBase: SDTL Types that Change a Dataframe

A. Commands that create variables or change the values of a variable	
Aggregate	An aggregation summarizes data using aggregation functions applied to data that may be grouped by one or more variables. The resulting summary data is added to each row of the existing dataset. The SDTL Collapse command is used when the summary data is used to create a new dataframe with one row per group..
Compute	Assigns the value of an expression to a variable.
Recode	Describes recoding values in one or more variables according to a specified mapping. The Recode command can either describe a recoding of one or more individual variables, or a range of variables. When one or more individual variables are described, a new variable name can be specified. In this case, the original variable is left alone, and a new variable is created with the recoded values.
SetMissingValues	Defines values that are treated as missing values for a list of variables.
B. Commands that change the metadata associated with a variable or dataframe	
Rename	Rename changes the name of a variable or list of variables.
SetDatasetProperty	Changes a property of a dataframe.

SetDataType	Sets the data type of a variable or list of variables.
SetDisplayFormat	Sets the display or output format for a variable or list of variables.
SetValueLabels	Describes the assignment of labels to categorical values.
SetVariableLabel	Describes the assignment of a label to a variable.
C. Commands that read or write files	
Load	Load data from a file.
Save	Writes a dataset to a file.
D. Commands that change the structure of a dataframe	
AppendDatasets	Combines datasets by concatenation for datasets with the same or overlapping variables.
Collapse	A collapse command summarizes data using aggregation functions applied to data that may be grouped by one or more variables. The resulting summary data is represented in a new dataset. See Aggregate for adding summary variables without changing the number of rows.
DropCases	Rows that match the selection condition are deleted in the dataset. Other rows are retained.
DropVariables	Deletes variables from the dataset.
KeepCases	Rows that match the selection condition are retained in the dataset. Other rows are deleted.
KeepVariables	Variables to be retained in the dataset. Variables not on the list are deleted.
MergeDatasets	Merges datasets holding overlapping cases but different variables. The merge may be controlled by keys or grouping variables.
NewDataframe	Creates a new empty dataframe. Numbers of rows or columns may be specified. All values are assumed to be missing.
ReshapeLong	Creates a new dataset with multiple rows per case by assigning a set of variables in the original dataset to a single variable in the new dataset.

ReshapeWide	ReshapeWide is not supported in the current version of SDTL, because it depends on values in the data. However, it may be useful when values of the index variable are available in the metadata file or the data can be processed.
SortCases	Sorts rows in the dataframe in a specified order.
E. Commands that control the flow of operations in a script	
DoIf	A set of commands that are performed when a logical expression is true. May also include ElseCommands to be performed if the logical expression is false. The commands in DoIf are performed once, and it expects a logical condition that applies to the entire dataframe. Use IfRows for commands that are performed on each row depending upon values on those rows.
Execute	This command causes the system to execute preceding commands before continuing to process the command script.
IfRows	A set of commands that are performed on each row in the dataframe when a logical expression is true for that row. May also include ElseCommands to be performed if the logical expression is false. Use DoIf for a logical condition that applies to the entire dataframe and commands that are performed once.
LoopOverList	A loop creates multiple versions of a set of commands by iterating over a list of variables, numbers, or strings.
LoopWhile	LoopWhile iterates over a set of commands under the control of one or more logical expressions. Since the logical conditions typically depend upon values in the data, commands executed in a LoopWhile cannot be anticipated and expanded in SDTL.

InformBase

Table 2 shows informational commands that do not describe changes to the data. Although SDTL does not include commands that analyze data, these commands can be transcribed verbatim in an SDTL script with the *Analysis* command. *Unsupported* is used for commands that our Parser cannot translate into SDTL. *Invalid* is used when the parser recognizes a command in the source language but its syntax does not conform to expectations. *NoTransformOp* was created for commands in the source language that do not play a role in SDTL. For example, R and Python install libraries that may change the operation of commands. Even though the Parser has translated these commands into SDTL, the library may be relevant information for some data users.

Table 2.

InformBase: Commands that provide information

Analysis	Describes an analysis command. An analysis command does not result in any data transformation.
Comment	Describes a source code comment.
Invalid	Describes an invalid command. A command is invalid if it uses incorrect syntax, or is otherwise not allowed by the executing system.
Message	Inserts message text in the SDTL file.
NoTransformOp	*NoTransformOp* is used for a command in the original script that provides important information but does not have a function in SDTL. For example, "library()" in R loads a package of R functions. Since the Parser detects the library, the SDTL will reflect the library that is used, and commands derived from the library will be translated in the SDTL script. However, it is useful to know which library is active for auditing the R script, even if it does not perform any data transformations.
Unsupported	Describes an unsupported command. An unsupported command is valid syntax, but not supported by the parsing application.

ExpressionBase

The SDTL types in Table 3 (*ExpressionBase*) are used in expressions, which may be numeric, text, date-time, or logical. The most powerful of these types is *FunctionCallExpression*, which is a reference to the Function Library discussed below.

VariableReferenceBase (Table 4) is a subcategory of *ExpressionBase* used to describe the variables used in an expression.

Table 3.

ExpressionBase: SDTL Types Used in Expressions

BooleanConstantExpression	BooleanConstantExpression takes values of TRUE and FALSE.
DateTimeConstant	Describes a date or date-time combination using an ISO 8601 compliant string.
FunctionCallExpression	An expression evaluated by reference to the Function Library.
GroupedExpression	A group of expressions to be evaluated before expressions outside of the group. Used to control the order of operations in a formula.
IteratorSymbolExpression	The name of an iterator symbol used as an index in describing the actions of a loop.
MissingValueConstantExpression	A missing value constant. Some languages allow multiple missing value constants.
NumberRangeExpression	Defines a range of numeric values.
NumericConstantExpression	A numeric constant.
NumericMaximumValueExpression	Represents the largest numeric value supported by a system.
NumericMinimumValueExpression	Represents the smallest numeric value supported by a system.
StringConstantExpression	A text string.
StringRangeExpression	Defines a range of string values.
TimeDurationConstant	Describes a duration of time using an ISO 8601 compliant string.
UnhandledValuesExpression	Represents any values not previously handled (for example, in a set of recode rules).
ValueListExpression	Wraps a list of other expressions.
VariableReferenceBase	SDTL types used to describe variables. See Table 4.

Table 4.

VariableReferenceBase: SDTL Types Used to Describe Variables in Expressions

AllNumericVariablesExpression	An expression that represents all numeric variables in the dataset, similar to `_all` in SPSS or Stata.
AllTextVariablesExpression	An expression that represents all text variables in the dataset, similar to `_all` in SPSS or Stata.
AllVariablesExpression	An expression that represents all variables in the dataset, similar to _all in SPSS or Stata.
CompositeVariableNameExpression	A composite variable name is used to describe a variable name that is computed.
VariableListExpression	A list of variables, which may include variable names (VariableSymbolExpression) and variable ranges (VariableRangeExpression).
VariableRangeExpression	A list of variables in adjacent columns defined by the variable names of first and last columns.
VariableSymbolExpression	A reference to a variable.

Table 5 includes types that were created to represent complex properties of other commands. For example, every type in *CommandBase* uses *sourceInformation* to show the original language of each command and its location in the command script. *AppendDatasets* and *MergeDatasets*, which operate on more than one file use types *AppendFileDescription* and *MergeFileDescription* to capture a number of properties associated with each file.

Table 5.

Types for Complex Properties in SDTL Commands

AppendFileDescription	Describes files used in an AppendDatasets command.
DataframeDescription	Describes a dataframe in the consumesDataframe or producesDataframe types. Provides the name of the data frame and a list of variables (columns). DataframeDescription can also define dimensions in dataframes that have hierarchical indexes, data cubes, or multi-indexes.
FunctionArgument	Describes the arguments in a function as specified in the SDTL Function Library.
IteratorDescription	Describes an iteration process consisting of an IteratorSymbolExpression and a list of values it takes.
MergeFileDescription	Describes files used in a MergeDatasets command.
RecodeRule	Describes how values will be recoded.
RecodeVariable	Describes a variable that will have its values recoded.
RenamePair	Variable names before and after a variable is renamed.
ReshapeItemDescription	Describes a new variable created by reshaping a dataset from wide to long.
SortCriterion	Describes a criterion by which cases are sorted, including the variable name and whether to sort ascending or descending.
SourceInformation	SourceInformation defines information about the original source of a data transform.
ValueLabel	Associates a label with a value in a categorical variable.

Conditional Execution by Row or by File/Dataframe

Statistical languages have some commands that are executed sequentially on every row and other commands that apply to an entire file or dataframe. The *Compute* command illustrated above is an example of the first type. *Compute* creates or modifies a variable that will appear on every row in the data. New variables are usually computed from other variables on the same row, but we describe calculations that aggregate across rows in our discussion of the "Function Library" below. In contrast, commands that load or save files or modify metadata, such as data type and display format, do not change the number or contents of rows in the dataframe.

The difference between row-level and file/dataframe-level commands becomes very important when the action is conditional on the value of a variable or other parameter. Consider these commands in the Stata language:

```
replace varY=3 if varX>5   /*** Version 1 *****/

if varX>5 replace varY=3   /*** Version 2  ****/
```

Although they appear to be the same, they have very different outcomes. The condition in Version 1, “if varX>5”, is a qualifier within a Stata command (“replace”) that is executed sequentially on each row in the dataframe. Some rows will be set to 3 and others will not be changed, depending upon the value of “varX” on each row. In Version 2 the “replace” command is nested in an “if” command, which is a program flow command designed for use in Stat scripts (“do-files”). The “if” command is not evaluated separately for each row; it is evaluated only once using the value of “varX” on the first row in the dataframe. Consequently, if “varX>5” is true for row one, “varY” is set to 4 for all rows, and if “varX>5” is false for row one, no rows are changed regardless of the value of “varX” on other rows. Table 6 illustrates the results of these commands where only row 1 satisfies the condition “varX>5”.

Table 6. Examples of Conditional Execution by Row and Dataframe in Stata

	Initial values		Version 1 (SDTL *IfRows*): replace varY=3 if varX>5		Version 2 (SDTL *DoIf*): if varX>5 replace varY=3	
Row	varX	varY	varX	varY	varX	varY
1	9	11	9	3	9	3
2	4	11	4	11	4	3
3	1	11	1	11	1	3

SDTL includes two ways of applying conditions to commands. The SDTL command *IfRows* is used for conditions that should be evaluated sequentially on every row. *DoIf* in SDTL is used for flow control in scripts where the condition is evaluated once before executing a command or group of commands. Both *IfRows* and *DoIf* can be applied to a group of commands, and both include an *elseCommands* property for commands to be performed if the condition is false.

Function Library

Although the number of data transformation commands in statistical packages is small, the power of these commands is magnified by “functions,” which are available in every language. Functions are available in most computer languages as a convenient way to invoke common operations. In the same way that a mathematical equation may use “sine(x)” to refer to the corresponding trigonometric function, “sine(varX)” may be used in a statistical package to insert the sine of variable “varX” in a computation or comparison. There are thousands of functions in statistical packages, and programming C^{2}Metadata parsers and updaters to reproduce all of them would have been an enormous job. Fortunately, our goal is to describe data transformations not to perform them. We devised a simple way to add an unlimited number of functions to SDTL with minimal impact on the code required to translate a script into SDTL. This was accomplished by creating a Function Library, which serves as a crosswalk between SDTL and the various statistical packages. The Function Library is a file that can be maintained in a spreadsheet and accessed as a JSON file.

Functions in computer languages normally have two parts: a function name followed by parameters enclosed in parentheses. The function invokes program code that replaces the function with a value computed from the parameters. The computed value of a function may be a number, text, or logical (Boolean) constant. For example, sine(varX) will return the sine of an angle equal to the value of varX, and gt(varX, varY) will return TRUE if varX is greater than varY and FALSE otherwise. Each parameter is used in a specific way by the computer code that evaluates the function.

Parameters may be specified in two ways. Sometimes, parameters are given in a defined order separated by a delimiter, usually a comma, which is included even if the parameter is omitted. Parameters may also be identified by name. For example, in Stata “std(varX), mean(10) std(3)” will standardize the values of varX so that the transformed values have mean=10 and standard deviation=3. In this case the first parameter (varX) is given by position, but the other two parameters (“mean” and “std”) are specified by name. In SDTL parameters may be specified by position or by name. We currently use EXP1, EXP2, EXP3, … as parameter names in SDTL, but these names are arbitrary and meaningful mnemonics could be used.

Some functions operate on a list items of the same type, which makes them appear to have an indeterminate number of parameters. For example, mean(varX, varY, varZ) would compute the mean of three variables. To avoid parameter lists of indefinite length, the SDTL Function Library uses the *VariableListExpression* and *ValueListExpression* types in SDTL. A *VariableListExpression* packages a list of variables into a single SDTL type that is treated as one parameter in an SDTL function. A *VariableListExpression* has a single property (*variables*) defined as a JSON array that can consist of any combination of individual variables (*VariableSymbolExpression*) or variable ranges (*VariableRangeExpression*).

The Function Library maps the names and parameters of SDTL functions to functions in other languages. Every function is described with the SDTL name of the function and the order and names of its parameters. The SDTL function is also mapped to the same function in SPSS, Stata, SAS, R, and Python. This table compares functions that compute a random number from a uniform distribution in SDTL and five other languages:

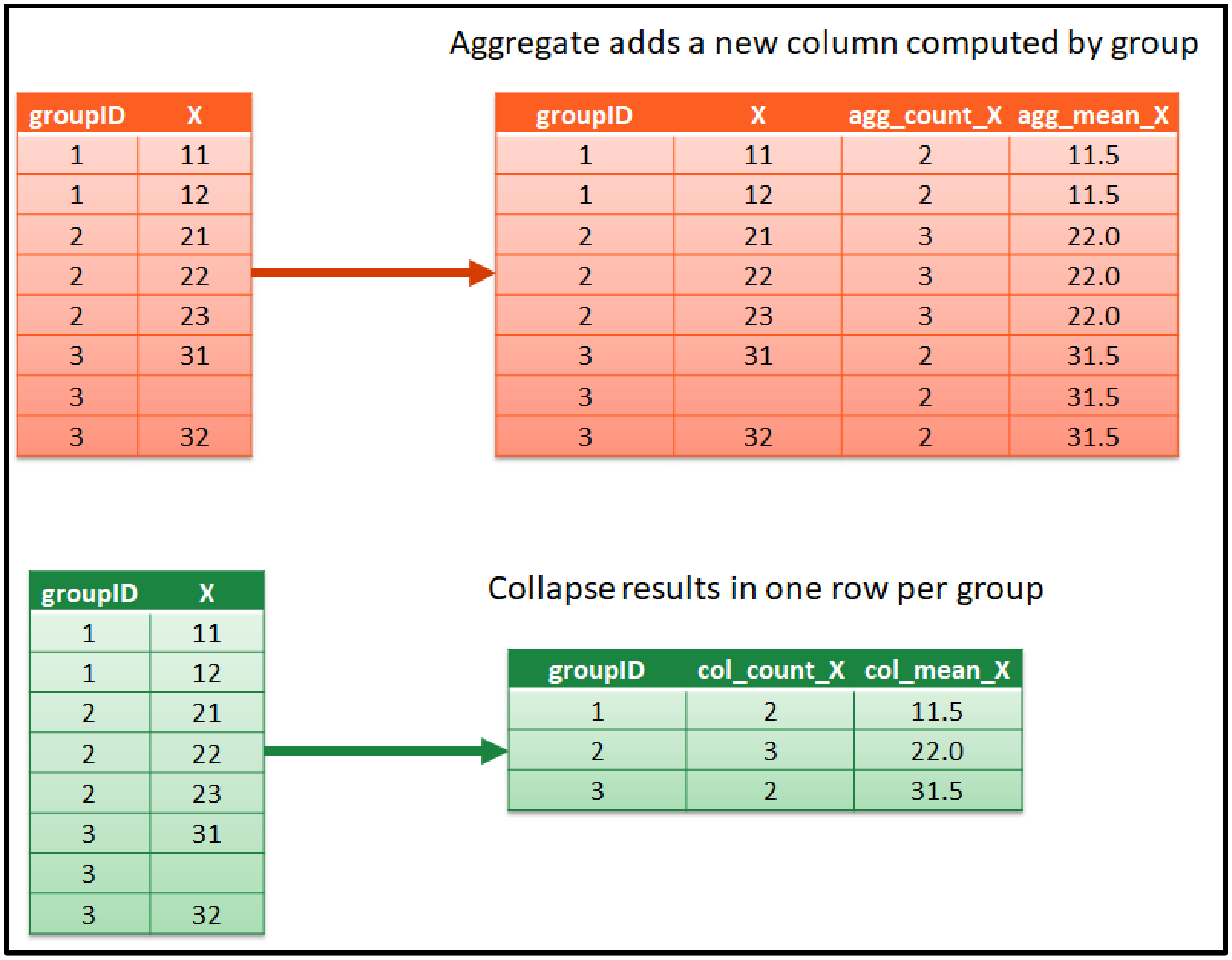
SDTL	random_variable_uniform(EXP1, EXP2)
SPSS	RV.UNIFORM(EXP1, EXP2)
Stata	runiform(EXP1 EXP2)
SAS	RANUNI(seed)
R	runif(n, min=EXP1, max=EXP2)
Python	numpy.random.uniform(low= EXP1, high= EXP2)

SDTL and most of these languages specify the minimum (EXP1) and maximum (EXP2) of the range of the random number. In SAS the range is always 0 to 1, which is the default range in other languages. The Function Library entry for SAS specifies that 0 and 1 are passed to SDTL as values for parameters EXP1 and EXP2. Computer programs often use mathematical formulas to approximate random numbers, and the SAS version of this function allows users to specify a "seed" for its random number generator. Since the seed is specific to the implementation in SAS, it is not included in SDTL. In R the "runif" function creates a vector of random numbers of length "n". We assume that the random number will be either a single number used in an expression or a vector added to the dataframe as a new variable, which makes this parameter unnecessary in SDTL.

The Function Library partitions functions into four groups corresponding to different SDTL commands:

Function Library group	SDTL command	Meaning
Horizontal	*Compute*	Calculates a value from variables on the same row. Rows are processed sequentially.
Vertical	*Aggregate*	Calculates a new variable by aggregating across rows in a group. Every row in the group has the same value.
Collapse	*Collapse*	Calculates a new variable by aggregating across rows in a group in a new dataframe with one row per group.
Logical	*DoIF* *IfRows* *KeepCases* *DropCases*	Functions used in logical conditions.

Horizontal functions operate sequentially by row using variables appearing on each row. Vertical and Collapse functions operate on groups of rows by aggregating values within columns (see Figure 2). Vertical functions used in an SDTL *Aggregate* command add new variables (columns) to a dataframe by applying the result of a computation to every row in a group. The *Collapse* command does the same computation, but it reduces the number of rows by creating one row per group. For example, suppose that groups are defined by variable “YearsOfEducation,” and we compute mean(AnnualIncome). The *Aggregate* command will add mean AnnualIncome to every row, and the *Collapse* command will create one row for every value of YearsOfEducation including both YearsOfEducation and mean AnnualIncome. Thus, Vertical functions do not change the number of rows in the dataframe, and Collapse functions create a new dataframe with fewer rows.



Aggregate adds a new column computed by group

groupID	X
1	11
1	12
2	21
2	22
2	23
3	31
3	
3	32

groupID	X	agg_count_X	agg_mean_X
1	11	2	11.5
1	12	2	11.5
2	21	3	22.0
2	22	3	22.0
2	23	3	22.0
3	31	2	31.5
3		2	31.5
3	32	2	31.5

Collapse results in one row per group

groupID	X
1	11
1	12
2	21
2	22
2	23
3	31
3	
3	32

groupID	col_count_X	col_mean_X
1	2	11.5
2	3	22.0
3	2	31.5

Figure 2. Illustrations of Aggregate and Collapse

All functions have unique names in SDTL, but other languages sometimes use the same function name in different contexts with different outcomes. A good illustration is a function for computing means, which has three different meanings in both SPSS and Stata.

	Compute	Aggregate	Collapse
SDTL	mean(EXP1)	agg_mean(EXP1)	col_mean(EXP1)
SPSS	mean(EXP1) Context: COMPUTE	mean(EXP1) Context: AGGREGATE with MODE=ADDVARIABLES	mean(EXP1) Context: AGGREGATE
Stata	rowmean(EXP1) *Context: generate, replace*	mean(EXP1) Context: *egen*	"(mean)" statistic option Context: *collapse*

Flow Control, Loops, and Macros

Statistical software packages include extensive programming capabilities. Stata and SAS have powerful macro features, and R and Python are very capable programming languages. There are two ways of handling these programming features in SDTL.

First, whenever possible the Parser will expand macros and other programming code into simpler commands. For example, if a loop applies a *Compute* command to four variables, it can be converted into four *Compute* commands. This may make the SDTL long and verbose, but it simplifies the work of finding which commands affect every variable.

Second, SDTL does include types for describing loops (*LoopOverList, LoopWhile*), which are the most common kind of flow control, and *IteratorSymbolExpression* was created to describe an index used in a loop.

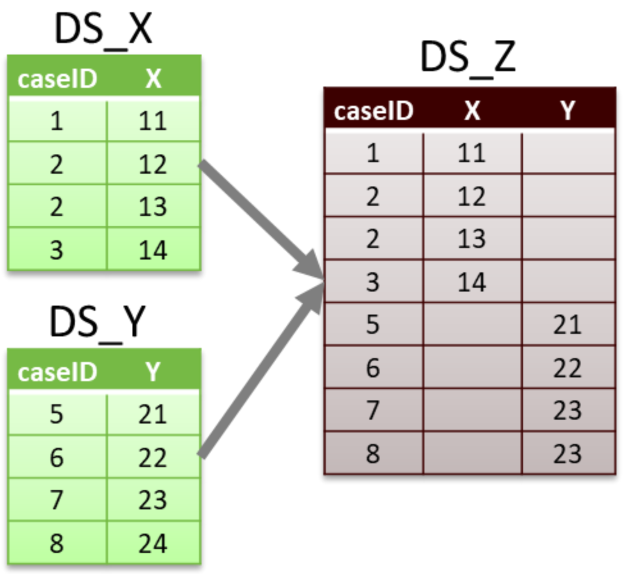
SDTL does not have arrays, which may be used in loops, but it does have functions that operate like arrays. The *VariableArrayDereference* and *ValueArrayDereference* functions allow an SDTL script to use an expression to select an entry in a list. The first parameter of each function points to the position of an entry in a variable or value list given as the second parameter. The operation of these functions can be illustrated by this simplified example, in which "[Age, Sex, Education, Income]" is a list of variable names:

VariableArrayDereference(3, [Age, Sex, Education, Income])

The value of this function would be "Education", which is the third item in the list. Since the contents of the list is not stored anywhere, the full list must be repeated every time that the function is used. However, the index parameter could be an *IteratorSymbolExpression* in a loop.

Appending, Merging, and Updating Datasets

The five languages covered by the C^{2}Metadata Project offer a wide variety of ways of combining datasets. *AppendDatasets* is used to concatenate rows (cases) from datasets that have the same columns (variables) (Figure 3). *MergeDatasets* combines columns from datasets that have the same rows. These operations are complicated by features that resolve conflicts, such as merging files with overlapping column names or unmatched rows. Datasets are usually merged by joining rows with the same keys, but some statistical packages will merge rows sequentially when keys are not specified. *MergeDatasets* can also be used to update a dataset by replacing its current values with values from a different dataset. Both *AppendDatasets* and *MergeDatasets* use subtypes (*AppendFileDescription*, *MergeFileDescription*) to describe actions that apply to specific input datasets. For example, the merge commands in SPSS and SAS allow users to rename variables, select variables, and select cases at the time of the merge without changing the input dataset.


DS_X

caseID	X
1	11
2	12
2	13
3	14

DS_Y

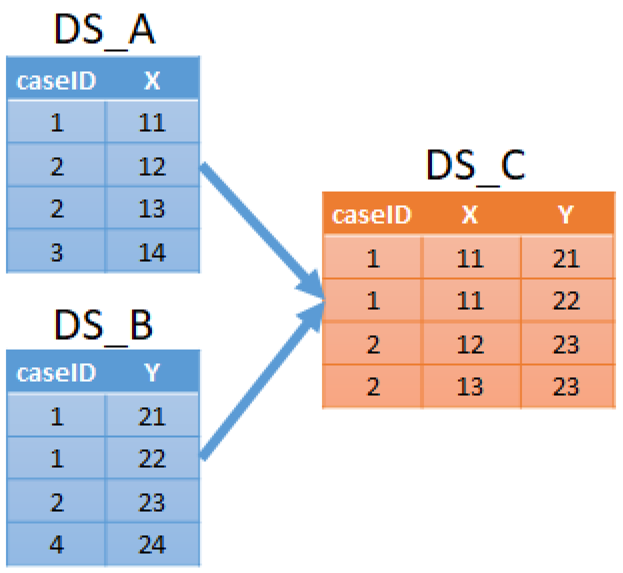
caseID	Y
5	21
6	22
7	23
8	24

DS_Z

caseID	X	Y
1	11	
2	12	
2	13	
3	14	
5		21
6		22
7		23
8		23

Figure 3. Appending Datasets

R (dplyr) and Python (Pandas) use "joins" like those in SQL to merge dataframes. Rows in the output dataset are created by comparing one or more key variables specified in a "by" parameter. Joins in R and Python are implicitly Cartesian joins that create every possible combination of rows with the same keys. For example, caseID=2 is repeated in DS_A and caseID=1 is repeated in DS_B. The Cartesian join of DS_A and DS_B by caseID is DS_C (Figure 4), in which there are two rows for both caseID=1 and caseID=2. Note that caseID=3 and =4 are not included in DS_C, because they do not exist in both input datasets. DS_C is the result of an "inner" join, and the unmatched rows can be included by specifying "outer", "left", or "right" joins. Following the model of SQL, R and Python are agnostic about the order in which the data are sorted, and all joins are Cartesian.



DS_A

caseID	X
1	11
2	12
2	13
3	14

DS_B

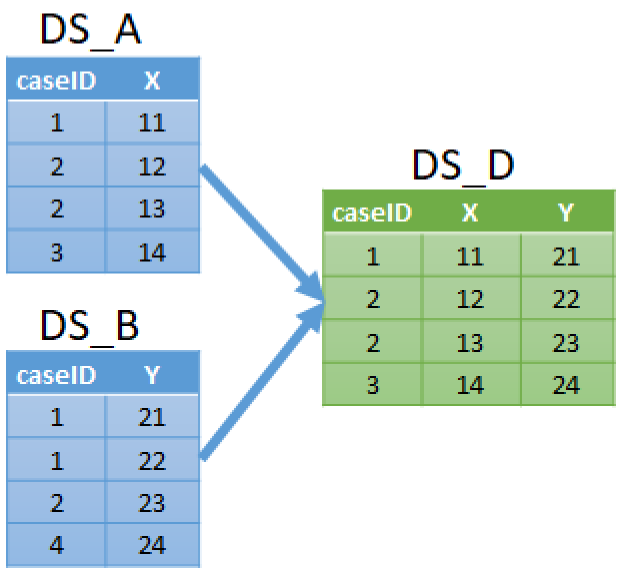
caseID	Y
1	21
1	22
2	23
4	24

DS_C

caseID	X	Y
1	11	21
1	11	22
2	12	23
2	13	23

Figure 4. Cartesian Inner Join

In SPSS, SAS, and Stata merging is often a sequential process on files that are sorted before they are merged. Even when the merge involves matching on key variables, SPSS, SAS, and Stata require the input files to be sorted before they can be merged, and the user must determine whether keys are unique (one-to-one) or repeated (one-to-many or many-to-many). A sequential merge of DS_A and DS_B without keys produces DS_D (Figure 5), which is very different from the result of a Cartesian join (Figure 4).



DS_A

caseID	X
1	11
2	12
2	13
3	14

DS_B

caseID	Y
1	21
1	22
2	23
4	24

DS_D

caseID	X	Y
1	11	21
2	12	22
2	13	23
3	14	24

Figure 5. Sequential Merge

SDTL uses three properties (*mergeType*, *newRow*, and *update*) to represent all of these possibilities. These properties are found in *MergeFileDescription*, which means that they are specified for each input dataset.

The *mergeType* property describes how rows from the input datasets are combined in the output data. Most merge types (e.g. OneToOne, OneToMany) involve matching rows on key variables, which are specified with *mergeByVariables* (in *MergeDatasets*) and *mergeByNames* (in *MergeFileDescription*). Sequential merges assume that the input files are already sorted.

The *newRow* property determines when the rows contributed by an input file generate a row in the output file. When *newRow* is TRUE, all rows in this dataset are included in the output dataset, regardless of whether they were matched to another input dataset on the *mergeByVariables*. When *newRow* is FALSE, only rows that have been matched are included. An inner join is represented in SDTL by setting *newRow* to FALSE on all input datasets, and *newRow* is TRUE for all input datasets to describe an outer join. Left and right-joins are created by using TRUE and FALSE on different inputs.

mergeType	
Sequential	Match rows from each input dataframe in the order in sequential order.
OneToOne	Create one row for each value of the MergeByVariables. If a combination of the MergeByVariables is repeated, only one row is matched. Rows with repeated combinations of the MergeByVariables may or may not be included in the output file depending on the NewRow property.
OneToMany	Create a row in the output dataframe by matching rows in this dataframe to every row in other dataframes with the same value of MergeByVariables. Note that OneToMany implies that one of the other input datarames is set to ManyToOne.
ManyToOne	Create a row in the output dataframe by matching all rows in this dataframe to the one row in the other dataframe with the same value of MergeByVariables.
Cartesian	Create a new row in the output dataframe for every possible combination of rows having the same value of MergeByVariables. This is equivalent to a many to many merge.
Unmatched	Create a new row for every row that cannot be matched on the MergeByVariables
SASmatchMerge	SAS uses a merging approach that combines matching keys and sequential merges within groups.

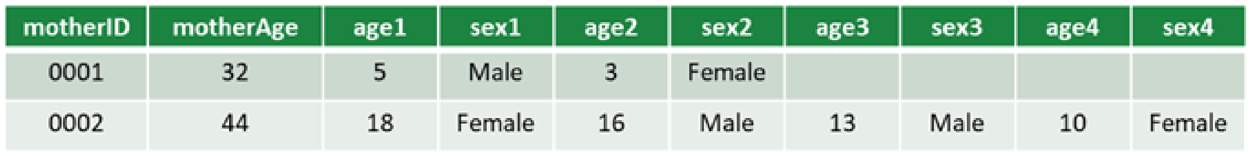
newRow	
TRUE	Always include rows from this dataframe, even if the MergeFileVariables do not match a row in any other dataframe.
FALSE	Only include rows from this dataframe, if the MergeFileVariables match a row in another dataframe.

There is even more diversity in the responses of different languages when the datasets to be merged contain a variable (column) with the same name. R and Python follow SQL by including both variables with modified names, which can be handled by using the *renameVariables* property in the *MergeFileDescription*. However, SPSS, Stata, and SAS will include only one variable in the output data, and they may use the omitted variable to update values in the included variable. The *update* property of *MergeFileDescription* is used to specify how values from the omitted version of the variable will be handled. If *update* is set to Ignore, a variable that is also found in the Master dataset will have no effect on the output dataset. If *update* is set to FillNew, values from the repeated variable will only appear on new rows not found in the Master dataset. UpdateMissing replaces only missing values in the Master dataset, and Replace changes all values on matched rows in the Master dataset.

update	
Master	This dataframe is the Master dataframe.
Ignore	If a column with the same name exists in the Master dataframe, ignore the values in this dataframe.
FillNew	If a column with the same name exists in the Master dataframe, use the values from this dataframe only in new rows created from this dataframe.
UpdateMissing	If a column with the same name exists in the Master dataframe, use values from this dataframe when the value in the Master dataframe is missing.
Replace	If a column with the same name exists in the Master dataframe, use values from this dataframe.

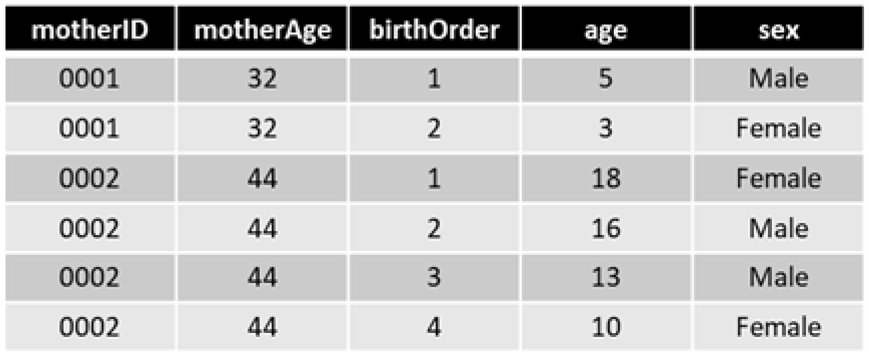
ReshapeLong, ReshapeWide, and CompositeVariableNameExpression

All of the statistical packages covered by the C2Metadata project have commands to reshape files between “wide” and “long” formats. Figures 6 and 7 illustrate the difference between wide and long format for data describing a mother and her children. In the wide format (Figure 6) there is one row for each mother, and each child is described by two variables, age and sex. Data for each child are identified by including birth order in the variable name, e.g. age1, age2, etc. The wide format requires a column for every variable for each child, and we must allow enough variables to describe the largest family in the data. If one woman had 20 children, the dataset in Figure 6 would have 40 columns: age1, sex1, …, age20, sex20. Consequently, datasets in wide format usually have many empty cells. In long format, Figure 7, there is a row for each child and information about mothers is repeated on the rows for each of their children. The long format includes an additional variable, birthOrder that uniquely identifies children within each family. Since the information in each format is identical, the choice between wide and long depends upon the types of analysis to be performed and convenience.



motherID	motherAge	age1	sex1	age2	sex2	age3	sex3	age4	sex4
0001	32	5	Male	3	Female				
0002	44	18	Female	16	Male	13	Male	10	Female

Figure 6. Wide Format



motherID	motherAge	birthOrder	age	sex
0001	32	1	5	Male
0001	32	2	3	Female
0002	44	1	18	Female
0002	44	2	16	Male
0002	44	3	13	Male
0002	44	4	10	Female

Figure 7. Long Format

The information in Figures 5 and 6 can also be stored in separate datasets for mothers and children by using the motherID variable as a key for linking children to their mothers. In a relational database the two-table approach would be used to remove repetition and "normalize" the database. However, unlike SQL, most statistical analysis software cannot compute results on data contained in more than one table. Data from the mothers table and the children table would need to be merged before any analysis is performed.

SDTL includes features for operating on wide and long format data. The *CompositeVariableNameExpression* is used to describe repeated variable names in wide-format data, such as age1, age2, etc. Composite names consist of a "stub" (e.g. "age", "sex") and an index value. Composite names are described in a *ReshapeItemDescription*, which is a complex property used in the *ReshapeWide* and *ReshapeLong* SDTL commands.

The C^{2}Metadata Project has implemented the *ReshapeLong* but not the *ReshapeWide* command. When we convert data from wide to long, we know how many rows to create, because each row corresponds to a set of identical variables described in the metadata file. But we cannot reshape data from long to wide format without knowing the maximum number of columns to create, which is not included in the metadata file of a long format dataset. Since the scope of the C^{2}Metadata Project has been limited to metadata-only operations, *ReshapeWide* is not currently supported.

Pseudocode Library and Translator

The Pseudocode Library is a simple and extensible way to create natural language versions of SDTL scripts. Every type in SDTL consists of a set of properties. Each of these properties can be resolved into text -- a variable name, a number, or a string. The Pseudocode Library is a set of templates for SDTL types with text to include before and/or after each property in a command. Templates look like this:

“starting text {property1} more text {property2} even more text”

The translation involves inserting text created from each property into the corresponding space in the template, where property names surrounded by curly brackets. For example, the pseudocode templates for the *Rename* command and the *RenamePair* type are:

Rename	Rename variables: {renames}
RenamePair	\n\t from {oldVariable} to {newVariable};

In this case, *RenamePair* is a complex type used to fill the *renames* property of the *Rename* command. If we rename varA to varAlpha, the *RenamePair* becomes.

```
\n\t from varA  to varAlpha;
```

and the *Rename* command becomes

```
Rename variables: \n\t from varA  to varAlpha;
```

If we evaluate \n as a new line and \t as a tab, we get

```
Rename variables:
        from varA  to varAlpha;
```

Pseudocode templates for functions are included in the Function Library.

Limitations and Future Developments

The DDI Alliance has created an SDTL Working Group to manage SDTL as one of its suite of standards. Modifications and additions to the SDTL standard will follow an orderly process with opportunities for community review and a published calendar for new versions. This framework assures that SDTL will evolve in response to new developments in source languages and new applications of the language.

The Function Library provides a simple way to expand the reach of SDTL without changing the language itself. The Function Library can be updated to include new functions in SDTL or to map additional functions in source languages to existing SDTL functions. In most cases, additions to the Function Library do not require changes to the program code in applications that translate source languages into SDTL.

Version 1.0 of SDTL is being released with two limitations that are due to the limited scope of the C^{2}Metadata Project. First, the C^{2}Metadata Project adopted a metadata-only approach. We assume that the pre-transformation data are well described in a standard metadata schema, such as DDI or EML, and we do not access the data at any time. For this reason, SDTL can describe reshaping data from long to wide, but C^{2}Metadata parsers do not support that command. When data are changed from long to wide, the number of columns in the new dataframe depends upon the values of the index variables in the original dataframe. The only way to know the range of these index variables is to inspect the actual data, and this requires integration of SDTL into statistical analysis software. We hope that this integration will happen in the future, especially for the open source packages R and Python.

Second, SDTL does not yet describe variables created by statistical analysis commands. SDTL was created to describe data and not tables, graphs or other analytical results. Since statistical analysis packages have many more analysis commands than data transformation commands, representing analysis commands was not on the agenda of the C^{2}Metadata Project. However, we acknowledge that analysis commands can also produce data. For example, estimated regression models are often used to construct predicted values and residuals. In view of the number and diversity of analytical commands, SDTL may be linked to an external ontology of statistical tests, such as the STATO ontology (ISA Commons, 2020).

SDTL was designed to document data transformations, and it is not intended to be an executable language. SDTL provides enough information for a human to understand changes to a file or a variable, but this may not be sufficient for a computer to perform these operations. In addition, a command script may be translated into SDTL in more than one way. SDTL, like other complex languages, often provides several methods for accomplishing a specific result. For example, the functions performed by the SDTL Recode command can also be achieved by IfRows and Compute statements or by the cut() function found in some languages.

Discussion

SDTL provides a new level of transparency for data processed and managed by statistical analysis packages. SDTL was created to simplify the automated creation of provenance metadata at the variable level. The C^{2}Metadata Project is providing open-source code for translating SPSS, SAS, Stata, R, and Python into SDTL, as well as code for translating SDTL into natural language (C2Metadata Project, 2020a). Software applications that create data catalogs, codebooks, and tools to reconstruct data provenance can read SDTL rather than interpreting each of the different statistical languages. For data producers, these tools simplify the process of describing the steps in preparing raw data for publication. Data repositories will receive more detailed machine-actionable metadata to improve the documentation their collections. Researchers will be able to understand how variables were created regardless of the software used in their production.

References

Alter, G., Donakowski, D., Gager, J., Heus, P., Hunter, C., Ionescu, S., . . . Voldsater, O. (2020). *Automating the Capture of Data Transformation Metadata from Statistical Analysis Software*. ICPSR. University of Michigan. Ann Arbor MI. Retrieved from http://hdl.handle.net/2027.42/156014

C2Metadata Project. (2020a). Gitlab Repository: c2metadata. Retrieved from https://gitlab.com/c2metadata

C2Metadata Project. (2020b). Structured Data Transformation Language. Retrieved from http://c2metadata.gitlab.io/sdtl-docs/

Cuevas-Vicenttín, Víctor, B Ludäscher, P Missier, K Belhajjame, F Chirigati, Y Wei, and B Leinfelder. 2016. "Provone: A Prov Extension Data Model for Scientific Workflow Provenance." Retrieved from http://jenkins-1.dataone.org/jenkins/view/Documentation%20Projects/job/ProvONE-Documentation-trunk/ws/provenance/ProvONE/v1/provone.html

DDI Alliance. (2020). Structured Data Transformation Language. Retrieved from https://ddialliance.org/products/sdtl/1.0

E.H. Fegraus, S. A., M.B. Jones, M. Schildhauer. (2005). Maximizing the value of ecological data with structured metadata: an introduction to ecological metadata language (EML) and principles for metadata creation. *Bulletin of the Ecological Society of America, 86*, 158–168.

IBM Corp. (2019). IBM SPSS Statistics for windows, version 26.0. Armonk, NY: IBM Corporation.

ISA Commons. (2020). STATO: an Ontology of Statistical Methods. Retrieved from http://stato-ontology.org/

Python Software Foundation. (2019). Python Language Reference, version 3.8. Beaverton, OR. Retrieved from https://www.python.org/

R Core Team. (2013). R: A Language and Environment for Statistical Computing. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from http://www.R-project.org/

SAS Institute. (2015). SAS®9.4 Product Documentation. Cary, NC: SAS Institute Inc. Retrieved from http://support.sas.com/documentation/94/index.html

StataCorp. (2020). Stata Statistical Software: Release 16.1. College Station, TX: StataCorp LP.

Vardigan, M. (2008). *Beyond the codebook: Documenting survey research on the Web.* Paper presented at the International Conference on Survey Methods in Multinational, Multiregional, and Multicultural Contexts (3MC), Berlin, Germany.

Endnotes

[1] University of Michigan
[2] Metadata Technologies North America
[3] Algenta Technologies
[4] Norwegian Centre for Research Data
[5] NORC
[6] We show SDTL types in italic font beginning with uppercase, like *Compute*. Properties within types are in italic font beginning with a lowercase letter, like *sourceInformation*.