Paper DS09

Selecting GeARS: Analysis Results Standard Implementation Options for Accelerated Analysis and TFL Automation

Richard Marshall, Clymb Clinical, Burlington, MA, United States
Bhavin Busa, Clymb Clinical, Burlington, MA, United States

ABSTRACT

Published as a new foundational CDISC standard in April 2024, the Analysis Results Standard (ARS) is complex; the model contains multiple classes, attributes and relationships. Most available ARS training focuses on understanding and use of the components of the model that describe analyses and outputs. However, the model contains additional components to support the generation of submission metadata and the automation of analyses and TFL creation. These additional classes are designed to allow different implementation strategies such as integration with existing macro libraries or fully integrated automation solutions.

Understanding how ARS metadata can be interpreted and how the additional classes can be used in different implementation scenarios is not a walk in the

- **P** **park**. Steering through the complexities of the ARS model, we demonstrate
- **R** **reverse**-compatibility with existing macro libraries, and show how the vendor-
- **N** **neutral**, software-agnostic metadata attributes can be leveraged to
- **D** **drive** the automation of analysis and TFL generation.

INTRODUCTION

In April 2024, CDISC published version 1 of the Analysis Results Standard (ARS) to support automation, consistency, traceability, and reuse of analysis results data. The basis of this new foundational standard is a logical model that defines classes that are used to:

1. Prospectively define analyses to be performed,
2. Describe outputs that display related sets of analysis results, and
3. Provide a standard, machine-readable format for the storage of individual analysis result values that are linked to the metadata definitions of the analyses that produced them.

The ARS Model documentation [1] describes the structure of the ARS classes (i.e., the attributes they contain) and the relationships between them, and the ARS User Guide [2] describes the use of the model. There have been multiple presentations and workshops describing the ARS model [3,4,5] and CDISC ARS Training is now available via the CDISC Learning System.[6]

Included in the ARS Model are several classes that are designed to accommodate programming code, either directly or via reference to external programming code files:

Class Name	Each Instance Represents
AnalysisOutputProgrammingCode	Programming statements and/or a reference to the program used to perform a specific analysis or create a specific output.
AnalysisOutputCodeParameter	A parameter whose value is used in programming code for a specific analysis or output.
AnalysisProgrammingCodeTemplate	Programming statements and/or a reference to a used as a template for creation of a program to perform method operations for a specific analysis.
TemplateCodeParameter	A replacement parameter whose value is substituted in template programming code to create statements required for a specific analysis.

This paper looks in detail at these programming code classes and describes how the stored or referenced programming code may be used in different implementation scenarios, ranging from the simple (e.g., identifying, for reference, the program file used to generate each output) to the complex (e.g., automated generation of analysis results).

In this paper, examples of ARS metadata are represented in YAML [7] format for compactness and readability. Other representations, such as JSON, are equally valid. Similarly, while “SAS Version 9.4” is shown as the programming language version used for many of the programming code examples in this paper, other programming languages are equally valid (e.g., R, Python, etc.).

1 OUTPUT PROGRAMMING CODE

In the simplest implementation scenarios, an instance of the AnalysisOutputProgrammingCode class may be used in the *programmingCode* attribute of the Output class to represent or reference the programming statements used to create the output. The *context* attribute of the AnalysisOutputProgrammingCode class is always used to indicate the programming language version used for the programming statements, and then either:

- The *code* attribute is used to represent the programming statements themselves as text, or
- The *documentRef* attribute is used to reference an external file, or the part of an external file, that contains the programming statements. In this case, the *documentRef* attribute contains an instance of the DocumentReference class, in which:
    - the *referenceDocumentId* attribute indicates the identifier value of a defined reference document (the external program file), and
    - the *pageRefs* attribute may optionally be used to specify a part of the referenced document.

Depending on the sponsor's implementation requirements, the output programming code may include just the statements needed to generate the analysis results represented in the output. Alternatively, the output programming code may also include the statements necessary to create the whole display.

In an implementation scenario where the sponsor simply wishes to indicate which program file was run to generate each output, the relevant program file can be identified as a reference document which is then referenced by identifier value (*referenceDocumentId*) in the *documentRef* attribute of the output's *programmingCode* attribute:

```
referenceDocuments:
- id: table140701_sas
  location: ./programs/table140701.sas
  name: table140701.sas
  description: Program for generation of table 14.7.01
…
outputs:
- id: Out14-7-01
  name: Table 14.7.01
  label: Summary of TEAE by System Organ Class and Preferred Term
  programmingCode:
    context: SAS Version 9.4
    documentRef:
      referenceDocumentId: table140701_sas
```

In the above example, all of the programming statements contained in the "table140701.sas" file are associated with the "Table 14.7.01" output. If only a subset of the "table140701.sas" were applicable for the "Table 14.7.01" output, it would be possible to reference the applicable parts of the file by page number(s), page range(s), or named destination(s) by using the *pageRefs* attribute in addition to the *referenceDocumentId* attribute. For more information about specifying page references, refer to the DocumentReference section of the ARS user guide.

In a different implementation scenario, where the sponsor wishes to associate executable code directly within the output's metadata definition, the executable programming statements could be specified as text in the *code* attribute of the output's programming code.

```
outputs:
- id: Out14-7-01
  name: Table 14.7.01
  label: Summary of TEAE by System Organ Class and Preferred Term
  programmingCode:
    context: SAS Version 9.4
    code: '%include "./programs/table140701.sas";'
```

The example above shows the SAS statement necessary to include and execute the contents of "table140701.sas" program file, which is similar to the defined reference to the program file shown in the first example. If the sponsor did not want to rely on any reference to external program files, the programming statements contained within the "table140701.sas" program file could instead be represented directly in the *code* attribute of the output's programming code.

```
outputs:
- id: Out14-7-01
```

```
  name: Table 14.7.01
  label: Summary of TEAE by System Organ Class and Preferred Term
  programmingCode:
    context: SAS Version 9.4
    code: 'data aesum;
           set adam.adae;
           where … '
```

Similarly, if the programming statements were available for execution from the sponsor's existing macro library, the call to the relevant macro could be included in the *code* attribute:

```
outputs:
- id: Out14-7-01
  name: Table 14.7.01
  label: Summary of TEAE by System Organ Class and Preferred Term
  programmingCode:
    context: SAS Version 9.4
    code: '%aesum;'
```

2 PARAMETERIZED OUTPUT PROGRAMMING CODE

Some sponsors may have existing macro libraries that contain generic programs used to generate outputs, where the specific characteristics of each output are determined by parameter (or argument) values passed to the selected generic program at the time of execution. To support implementation scenarios involving the use of parameterized programs, the AnalysisOutputProgrammingCode class also contains a *parameters* attribute. When referencing or providing the parameterized programming statements used to generate an output, the parameter values used at the time of execution of the specified or referenced programming code statements are recorded in the *parameters* attribute of the output programming code. Within the *parameters* attribute, each parameter is represented as an instance of the AnalysisOutputCodeParameter class, in which:

- the *name* attribute holds the name of the parameter,
- the *value* attribute holds the single value for the parameter, and
- optionally, the *label* and *description* attributes may either or both be used to provide, respectively, a short label or longer description for the parameter.

The following example shows ARS metadata that might be created to indicate that the "Table 14.7.01" output is generated using a parameterized Python program that accepts 3 runtime arguments:

```
referenceDocuments:
- id: aesum_py
  location: ./programs/aesum.py
  name: aesum.py
  description: Python program for generation of AE summary tables
…
outputs:
- id: Out14-7-01
  name: Table 14.7.01
  label: Summary of TEAE by System Organ Class and Preferred Term
  programmingCode:
    context: Python 3.12
    documentRef:
      referenceDocumentId: aesum_py
    parameters:
    - name: bysoc
      label: Summarize by SOC (Y/N)
      value: Y
    - name: bypt
      label: Summarize by PT (Y/N)
      value: Y
    - name: popvar
      label: Population variable
      description: >
        Records are selected for inclusion when the named variable has a value of Y.
      value: SAFFL
```

The metadata in the example above can be interpreted to indicate that the output is generated through execution of the Python program with the specified argument values, e.g.:

```
> python ./program/aesum.py --bysoc=Y --bypt=Y --popvar=SAFFL
```

Alternatively, parameter values might be specified for substitution into programming statements provided in the *code* attribute. In an implementation scenario where the sponsor has a SAS macro library containing an "aesum" macro that performs the same functions, and takes the same parameters, as the previous example Python program, the programming code might be represented as follows (where the parameter labels and description have been excluded for brevity):

```
outputs:
- id: Out14-7-01
  name: Table 14.7.01
  label: Summary of TEAE by System Organ Class and Preferred Term
  programmingCode:
    context: SAS Version 9.4
    code: '%aesum(bysoc=&bysoc, bypt=&bypt, popvar=&popvar);'
    parameters:
    - name: bysoc
      value: Y
    - name: bypt
      value: Y
    - name: popvar
      value: SAFFL
```

In this example, the value specified for each parameter could be substituted into the specified programming code statement, at the places indicated by the ARS parameter name preceded by an ampersand (&), to give an executable SAS (version 9.4) statement of:

```
%aesum(bysoc=Y, bypt=Y, popvar=SAFFL);
```

In an alternative implementation scenario, instead of substituting the parameter values into the programming code prior to execution, the defined parameter values could be instantiated within the execution environment before the unmodified programming code is executed. For example:

```
%let bysoc = Y;
%let bypt = Y;
%let popvar = SAFFL;
%aesum(bysoc=&bysoc, bypt=&bypt, popvar=&popvar);
```

At present, the ARS does not specify any conventions or requirements for the representation or referencing of parameter values with or within referenced or specified programming code statements. Sponsors may define or choose conventions that best facilitate their implementation strategy. For example, if a sponsor were using a Python-based engine to interpret ARS metadata to produce executable SAS code, they might choose an alternative convention for the representation of substitution parameters that facilitates Python string manipulation. In this case, the positioning of the replacement strings within specified programming code might be indicated by braces ({}) instead of a leading ampersand:

```
    code: '%aesum(bysoc={bysoc}, bypt={bypt}, popvar={popvar});'
```

Any convention used should be described in the documentation related to the implementation and must be communicated with any recipient if the ARS metadata is to be shared.

3 ANALYSIS PROGRAMMING CODE

Each output may contain the results of several individual analyses (e.g., summary statistics for different groupings of data, total values used as denominators, associated p-values). The ARS also supports the provision of programming code at the analysis level and sponsors may choose an implementation strategy that involves referencing or representing programming code at the analysis level instead of, or as well as, at the output level.

Like the Output class, the Analysis class also contains a *programmingCode* attribute in which the programming code for the analysis is referenced or represented, as described above for the Output class, as an instance of the

AnalysisOutputProgrammingCode class. However, while the programming code associated with an output might include programming statements needed both to generate result values and to position and format them within the output display, the programming code associated with an analysis will generally only include the statements used to produce result values for the analysis.

The following example shows the specification of the SAS version 9.4 programming statements used to generate the results for the "Comparison of Age Group by Treatment" analysis:

```
analyses:
- id: An03_02_AgeGrp_Comp_ByTrt
  name: Comparison of Age Group by Treatment
  programmingCode:
    context: SAS Version 9.4
    code: 'proc freq data=ADSL;
      table TRT01A * AGEGRP / chisq;
      exact pchi;
      run;'
```

As shown for outputs in section 1 above, programming code for analyses may also be specified by reference to a program file that has been defined as a reference document, or by including a call to an existing macro.

Provision of programming code for analyses is optional. When providing programming code for individual analyses, sponsors may choose to provide code for all analyses or only for a subset of analyses. An implementation scenario that might involve provision of programming code for a subset of analyses is one in which the sponsor needs to represent Analysis Results Metadata – as defined in the CDISC Analysis Data Model Version 2.1 document [9] (ADaM v2.1) – for the key or critical analyses included in a submission (e.g., the analyses performed to obtain either the main results of efficacy and safety, or the results that provide the rationales for the setting of dosage and administration specifications).

ARS SUPPORT FOR ARM FOR DEFINE-XML

In 2015, CDISC published the Analysis Results Metadata Specification Version 1.0 for Define-XML Version 2 [8] (ARM for Define-XML) as a standard format to support the submission of Analysis Results Metadata as defined in the metadata model described in the ADaM v2.1 specification. According to the ADaM v2.1 specification, best practice is that Analysis Results Metadata be provided to assist the reviewer by identifying the critical analyses, providing links between results, documentation, and datasets, and documenting the analyses performed.

At the time of publication of this paper, the ARS is not considered to be a replacement for ARM for Define-XML. Instead, the ARS contains components to support the creation of ARM for Define-XML. In an ARS implementation scenario where the sponsor wishes to include Analysis Results Metadata with a submission, the sponsor might populate the ARS components corresponding with each of the ARM for Define-XML components – including the *reason*, *purpose*, and *programmingCode* attributes of the Analysis class – and then use this information to generate information in ARM for Define-XML format for inclusion with the submission. The ARS user guide contains more information about the mapping between ARS and ARM for Define-XML components.

PARAMETERIZED ANALYSIS PROGRAMMING CODE

As described above for output programming code, parameter values may be specified in the *parameters* attribute within the *programmingCode* attribute of an analysis. When programming code for an analysis is specified directly within the *programmingCode* attribute of an analysis – either by reference to an external file in the *documentRef* attribute or by inclusion of programming statements in the *code* attribute – any specified parameter values represent the values used at the time of execution of the programming code.

For example, as an alternative to using a specific set of programming statements (as shown in the previous example), the results for the "Comparison of Age Group by Treatment" analysis could be generated via a call to a generic macro that includes substitution parameters:

```
analyses:
- id: An03_02_AgeGrp_Comp_ByTrt
  name: Comparison of Age Group by Treatment
  programmingCode:
    context: SAS Version 9.4
    code: '%fisher(ands=&ANDS, grpvar=&GRPVAR, anvar=&ANVAR);'
```

```
parameters:
- name: ANDS
  label: Analysis dataset
  value: ADSL
- name: GRPVAR
  label: Grouping variable
  value: TRT01A
- name: ANVAR
  label: Analysis variable
  value: AGEGRP
```

4 TEMPLATE ANALYSIS PROGRAMMING CODE

While the programming code for an output may only be specified directly within the *programmingCode* attribute of the output, the programming code for an analysis may instead be provided via template programming code that is associated with the analysis method specified for the analysis. In the ARS model, each defined analysis is associated with an analysis method that represents the set of statistical operations that are performed to generate individual result values for the analysis. Each analysis method is defined as an instance of the AnalysisMethod class and may be referenced by identifier value in the *methodId* attribute of 1 or more analyses.

Template programming code may be specified in an analysis method's *codeTemplate* attribute as an instance of the AnalysisProgrammingCodeTemplate class. When specified, template programming code represents the programming statements used to perform the statistical operations defined for the method and generate results for any analysis that uses the analysis method. This means that, in implementation scenarios that use template programming code, the template programming code defined once for a method will be automatically associated with any analysis that uses the method.

The AnalysisProgrammingCodeTemplate class has the same attributes as the AnalysisOutputProgrammingCode class, and the attributes are used in the same way:

- The *context* attribute always used to indicate the programming language version used for the template programming statements,
- Either the *code* attribute is used to represent the template programming statements themselves as text, or the *documentRef* attribute is used to reference an external file, or the part of an external file, that contains the template programming statements.

Although not required, parameters will usually be defined with template programming code to allow analysis-specific requirements to be applied to the generic template code when it is used for individual analyses. As with analysis or output programming code, template programming code parameters are defined in the *parameters* attribute. However, there are some differences between the parameters defined for output or analysis programming code and those defined for template programming code:

- Each parameter defined for output or analysis programming code:
  - is defined as an instance of the AnalysisOutputCodeParameter class,
  - indicates the parameter value used at the time of execution of the programming code, and
  - must have a single value specified.
- Each parameter defined for template programming code:
  - is defined as an instance of the TemplateCodeParameter class,
  - indicates a parameter that determines the functionality of the template code when it is used for an analysis, and
  - may have:
    - no value specified, which indicates that an analysis-specific value must be specified when the template code is used in an analysis,
    - a list of values, which generally represents the set of values available for use with the parameter, one of which must be chosen when the template code is used in an analysis, or
    - a specification of a single value, which generally indicates the parameter's "default" value that applies unless it is overridden when the template code is used in an analysis. The single default value may be specified either as the actual value to be used (in the *value* attribute) or as a reference to the metadata element(s) containing the parameter value(s) (in the *valueSource* attribute – as discussed in the next section).

The following hypothetical example method has been created for use in any analysis that needs to generate counts of treatment-emergent adverse events (TEAE) that may optionally be grouped by system organ class (SOC) and/or preferred term (PT). Template programming code is defined as a call to an existing macro that accepts 3 parameters. The "bysoc" and "bypt" parameters determine whether counts will be grouped by SOC and/or PT (respectively), and the "popvar" parameter indicates the population flag variable used to filter records of inclusion (where the variable has

a value of “Y”). Both the “bysoc” and “bypt” parameters each have 2 values specified (“Y” and “N”) and, in this example implementation scenario, this indicates that:

- each parameter may only have a value of “Y” or “N”, and
- the appropriate analysis-specific value should be assigned for each parameter when the method is used for a specific analysis.

The “popvar” parameter has a single value of “SAFFL” and, in this example implementation, this indicates that the default value for the parameter is “SAFFL” unless overridden when the method is used for a specific analysis.

```
methods:
- id: MTH_SOCPT_COUNT
  name: Count TEAE by SOC/PT
  label: Count TEAE by System Organ Class and/or Preferred Term
  operations:
  - id: MTH_SOCPT_COUNT_1
    name: Count
    order: 1
    resultPattern: XXX
  codeTemplate:
    context: SAS Version 9.4
    code: '%socpt_count(bysoc=&bysoc, bypt=&bypt, popvar=&popvar);'
    parameters:
    - name: bysoc
      value:
      - Y
      - N
    - name: bypt
      value:
      - Y
      - N
    - name: popvar
      value: SAFFL
```

Depending on a sponsor’s operational requirements, there are several ways in which the template programming code for a method such as this could be implemented for any analysis that uses the method:

a. The selected parameter values could be used to modify the template programming code before it is stored as part of the analysis definition. Even though the selected parameter values are represented in the programming code, the sponsor might choose to represent them for reference in the analysis.

```
analyses:
- id: EXAMPLE1
  name: Count of TEAE by System Organ Class and Preferred Term (Safety Population)
  methodId: MTH_SOCPT_COUNT
  programmingCode:
    context: SAS Version 9.4
    code: '%socpt_count(bysoc=Y, bypt=Y, popvar=SAFFL);'
    parameters:
    - name: bysoc
      value: Y
    - name: bypt
      value: Y
    - name: popvar
      value: SAFFL
```

b. As in the previous example, the selected parameter values could be substituted into the template programming code before it is stored as part of the analysis definition. In this case, as the selected parameter values are already represented in the programming code, the sponsor might choose not to represent them in the analysis.

```
analyses:
- id: EXAMPLE2
  name: Count of TEAE by System Organ Class (Safety Population)
  methodId: MTH_SOCPT_COUNT
  programmingCode:
    context: SAS Version 9.4
    code: '%socpt_count(bysoc=Y, bypt=N, popvar=SAFFL);'
```

c. The template programming code could be copied unchanged to the analysis definition with the analysis-specific parameter values represented with the programming code. In this case, the sponsor chose not to re-represent unmodified default values (e.g., for the “popvar” parameter).

```
analyses:
- id: EXAMPLE3
  name: Count of TEAE by Preferred Term (Safety Population)
  methodId: MTH_SOCPT_COUNT
  programmingCode:
    context: SAS Version 9.4
    code: '%socpt_count(bysoc=&bysoc, bypt=&bypt, popvar=&popvar);'
    parameters:
    - name: bysoc
      value: N
    - name: bypt
      value: Y
```

d. Template programming code may be used for the analysis by “implied reference” only – i.e., as the programming code is already defined for the method referenced in the *methodId* attribute, it is not re-represented in the analysis definition. However, analysis-specific parameter values still need to be represented if there is no prespecified single default value, or if the default value is overridden for the analysis:

```
analyses:
- id: EXAMPLE4
  name: Count of TEAE (ITT Population)
  methodId: MTH_SOCPT_COUNT
  programmingCode:
    context: SAS Version 9.4
    parameters:
    - name: bysoc
      value: N
    - name: bypt
      value: N
    - name: popvar
      value: ITTFL
```

5 METADATA-BASED PARAMETERIZATION

As mentioned in the previous section, the *valueSource* attribute of a template programming code parameter can be used to reference the metadata element(s) from which the parameter value(s) can be obtained. This means that, not only can generic template programming code be defined just once per method, but the source of analysis-specific parameter values can also be defined just once per method. Through the use of template programming code and metadata-based parameterization, it is possible for the programmatic generation of analysis results to be automated based simply on the analysis method selected for each analysis.

In the following example, the analysis method is defined to calculate counts of subjects, optionally grouped by one or more grouping factors. The method definition includes template programming code represented as a call to an existing macro (“CatVar_Count_ByGrp”) that accepts 2 parameter values:

- **anid**: the id value for the analysis that uses the method, which is obtained from the metadata element referenced as “#/id”.
- **gfids**: the id value(s) for any grouping factors referenced by the analysis, which are obtained from the metadata elements referenced as “#/orderedGroupings/[*]/groupingId”.

Using a dataset created based on the definition of the analysis identified by the “anid” parameter value (i.e., in which records from the input analysis dataset specified for the analysis have been filtered according to the analysis set and any data subset specified for the analysis, and group id values have been applied according to the grouping factors defined for the analysis), the macro calculates counts of records grouped by any grouping factors specified in the value of the “gfids” parameter.

```
methods:
- id: Mth01_CatVar_Count_ByGrp
  name: Count by group for a categorical variable
  description: Count across groups for a categorical variable, based on subject occurrence
  operations:
  - id: Mth01_CatVar_Count_ByGrp_1_n
```

```
    name: Count of subjects
    label: n
    order: 1
    resultPattern: (N=XXX)
  codeTemplate:
    context: SAS Version 9.4
    code: "%CatVar_Count_ByGrp(anid=&anid,gfids=&gfids);"
    parameters:
    - name: anid
      description: Analysis Id
      valueSource: '#/id'
    - name: gfids
      description: Grouping Factor Identifiers
      valueSource: '#/orderedGroupings/[*]/groupingId'
```

In this example implementation scenario, the sponsor has defined a convention where a hash (#) at the start of the metadata element reference in the *valueSource* attribute indicates a "relative reference" (i.e., a metadata element at the specified location within the metadata defined for the analysis that uses the analysis method. According to the sponsor's referencing convention:

- "#/id" means "the value of the *id* attribute of the analysis that uses the analysis method"
- "#/orderedGroupings/[*]/groupingId" means "a space delimited list of the value of the *groupingId* attribute from any/all items specified in the *orderedGroupings* attribute of the analysis that uses the analysis method".

The following example shows the definition of 2 analyses that reference the analysis method that is defined above. Also shown below are abbreviated definitions of the analysis sets and grouping factor referenced by the example analyses. Neither programming code nor parameters are shown in the definition for each analysis because:

- The template programming code defined for the analysis method can be used by "implied reference" (i.e., it is implied by the reference to the analysis method specified in the *methodId* variable) so it does not need to be re-represented in the definition of the analyses.
- The value of the *valueSource* attribute for each of the template programming code parameters indicates the default value to use for each parameter, so neither of the parameter values needs to be specified in the definitions of the analyses.

```
analysisSets:
- name: Intent-to-Treat Population
  id: AnalysisSet_01_ITT
  …
- name: Safety Population
  id: AnalysisSet_02_SAF
  …
analysisGroupings:
- name: Treatment
  id: AnlsGrouping_01_Trt
  …
analyses:
- name: Summary of Subjects by Treatment (Safety Population)
  id: An01_05_SAF_Summ_ByTrt
  methodId: Mth01_CatVar_Count_ByGrp
  dataset: ADSL
  variable: USUBJID
  analysisSetId: AnalysisSet_02_SAF
  orderedGroupings:
  - order: 1
    groupingId: AnlsGrouping_01_Trt
    resultsByGroup: true
- name: Summary of Subjects (Intent-to-Treat Population)
  id: An01_07_ITT_Summ
  methodId: Mth01_CatVar_Count_ByGrp
  dataset: ADSL
  variable: USUBJID
  analysisSetId: AnalysisSet_01_ITT
```

For the first example analysis:

- the value of the "anid" parameter is "An01_05_SAF_Summ_ByTrt" (the value of the *id* attribute of the analysis that uses the analysis method), and
- the value of the "gfids" parameter is "AnlsGrouping_01_Trt" (the value of the *groupingId* attribute for the only item in the *orderedGroupings* attribute of the analysis that uses the analysis method).

The analysis identified as "An01_05_SAF_Summ_ByTrt" references the "AnalysisSet_02_SAF" ("Safety Population") analysis set and the grouping factor identified as "AnlsGrouping_01_Trt" defines a grouping by "Treatment", so the call to the "CatVar_Count_ByGrp" macro referenced in the template programming code generates counts by treatment of subjects in the safety population.

For the second example analysis:

- the value of the "anid" parameter is "An01_07_ITT_Summ" (the value of the *id* attribute of the analysis that uses the analysis method), and
- the "gfids" parameter has no value because there are no ordered groupings defined for the analysis (the analysis that uses the analysis method does not have an *orderedGroupings* attribute).

The analysis identified as "An01_07_ITT_Summ" references the "AnalysisSet_01_ITT" ("Intent-to-Treat Population") analysis set, so the call to the "CatVar_Count_ByGrp" macro referenced in the template programming code generates an ungrouped count of subjects in the intent-to-treat population.

As indicated in the ARS user guide, the ARS does not define or require any specific syntax for metadata element references represented in the *valueSource* attribute of the TemplateCodeParameter class. Sponsors may define and use any convention or syntax that facilitates their implementation strategy. Any defined convention or syntax should be described in the documentation associated with the implementation and communicated to any recipient if the ARS metadata is shared. Sponsors may also choose to include in analysis definitions a representation of any analysis-specific parameter values obtained from metadata element references defined in the *valueSource* attribute of template programming code parameters.

CONCLUSION

The ARS Model includes programming code classes that allow programming code to be defined for outputs, analyses or analysis methods. These classes, and their associated parameter classes, provide the flexibility necessary to facilitate many different implementation strategies, which:

- range in complexity from the relatively simple specification of program files used to generate outputs through to the automated generation of analysis results, and
- allow integration of ARS metadata both with mature programming environments and novel implementations.

REFERENCES

1. CDISC ARS Model: https://cdisc-org.github.io/analysis-results-standard/
2. CDISC ARS User Guide version 1.0: https://wiki.cdisc.org/display/ARSP/Analysis+Results+Standard+User+Guide+v1.0
3. Busa, Bhavin; LeRoy, Bess; 'Pre-launching CDISC Analysis Results Standards' at PHUSE US Connect (Mar 2023) Presentation Link, Video Link
4. Busa, Bhavin; Marshall, Richard; LeRoy, Bess; 'All You Need to Know about the New CDISC Analysis Result Standards!' at PharmaSUG (May 2023). Paper Link
5. Busa, Bhavin; LeRoy, Bess; Marshall, Richard; 'Hands-on Workshop: Deep-dive Review and Testing of CDISC Analysis Results Standards Logical Model and Schema' at PHUSE US Connect (Mar 2023). Workshop files available on ARS GitHub.
6. CDISC Learning System: https://learnstore.cdisc.org/
7. YAML: https://en.wikipedia.org/wiki/YAML
8. CDISC Analysis Results Metadata (ARM) v1.0 For Define-XML v2: https://www.cdisc.org/standards/foundational/define-xml/analysis-results-metadata-arm-v1-0-define-xml-v2-0
9. CDISC ADaM v2.1 Specification: https://www.cdisc.org/standards/foundational/adam/adam-v2-1

CONTACT INFORMATION

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

Richard Marshall
ARS Principal Data Modeler
Principal Data Standards & Automation Architect,
Clymb Clinical
rmarshall@clymbclinical.com

Bhavin Busa
ARS Product Owner & Co-Lead
Principal & Co-founder,
Clymb Clinical
bhavin@clymbclinical.com