Paper PP03

Programming By Rules

Magnus Mengelbier, Limelogic, Malmö, Sweden

ABSTRACT

Rules are one of the cornerstones for analysing clinical trials, either defined through regulatory guidance, analysis plans or specifications. As standards for both the capture, analysis and reporting of data prevail in a business process; the flexibility is further constrained or more precisely defined. We can use this to our advantage by creating an evolving book of rules that can be implemented to verify that the capture data follows standards and study expectations as well as deriving the more standard variables for analysis and reports. We consider how a rule book can influence standard programming and the two SAS macros that are the cornerstone for rule based programming.

INTRODUCTION

Two popular myths are that all analysis is unique or all analysis is standard analysis and consistent. Both are valid but we are fast approaching consistency as CDISC and other industry standards have become mainstream and a de-facto standard.

Most organisations have a set of standards of varying detail, official and enforced through templates or unofficial propagated through copying the last or a similar analysis. As data standards mature, evolve and are refined, additional elements are incorporated into the standards as part of the natural process. More organisations are progressing to develop a book of standard derivations that drive consistency across all analysis.

It is not uncommon for data set standards, and associated standard derivations, to be implemented through large all-inclusive and ever complex standard macros. Another approach is to create many smaller functional and less flexible macros to execute a single defined derivation efficiently. Or simply eliminate macros entirely and focus on template code that efficiently derives one or a set of related rules.

Although each approach has its own merits and drawbacks, the functional approach, many small macros, is more efficient in meta-data driven environments as one macro would implement one derivation rule. Because the simplicity of the functional rule-based macros, creating test cases for validation is a straightforward exercise.

PROGRAMMING BY RULES

There exist countless different programming approaches that are efficient, with Good Programming Practice for Clinical Trials fostered by the community as one. One guiding principle is that the code should be to a degree self-documenting, clear, logical, etc. We have undoubtedly experienced one massive DATA step where all sorts of derivations occur and a conceivable simple update will unravel everything.

Programming by rules implements a modular approach and set pattern that combines the derivation of related variables, much like the structure of specifications. We can implement the approach as all-inclusive macros, such as all the rules related to Adverse Events, smaller macros associated with a single rule or as “regular” SAS code.

A good example is Age versus Body Mass Index (BMI), where AGE and BMI are derived in two separate rules, simply because they are not related.

```
%sdtm_age_1_0( data = work.source_dm, out = work.age );
%sdtm_bmi_1_1( dm = work.source_dm, vitals = work.source_vs, out = work.bmi );
```

Regardless of our coding approach, we need to consider that a derivation can be a simple affair, consisting of one line, or span multiple DATA steps and SAS procedures.

TWO IMPORTANT MACROS

The rule-based programming will result in many small data sets, conceivably one for each variable in extreme cases, that will need to be collated into the final data set. We can simplify the building of the final data set with two utility macros, `%rules_register()` and `%rules_make()`.

The basic idea is that after each rule, we register the intermediate data set, the derived variables, and any necessary merge keys and at the end use that information to create the final output data set.

RULES_REGISTER

The `%rules_register()` macro is a utility that creates and manages a list of intermediate data sets and associated variables so that the final output data set can easily be created.

```
%rules_register( registry = work._rules_registry_,
                 rule = ,
                 data = ,
                 vars = ,
                 keys = ,
                 dupkey = E);
```

The *registry* parameter points to a registry data set where each registration is stored and later used to build the final output data set. If the registry data set does not exist, it will be created the first time `%rules_register()` is called.

The *rule* is an optional reference to the derivation rule as defined in a book of rules or the specification that relates to the intermediate data set and variables. The rule parameter is not used other than for documentation and traceability when reviewing the registry data set, but we will explore possible uses further on in this paper.

The *data* parameter specifies the intermediate data set that contains the derived variables specified in the *vars* parameter.

The *keys* parameter defines the merge keys to be used when adding the variables to the final output data set. If the keys parameter includes the value ***_core_,*** the variables are considered the core variables and are added first.

The *dupkey* parameter represents a feature that will check if the keys identify a single record. Depending on the parameter value, the macro will add a SAS error (E), warning (W), note (N) to the log if duplicates exist. If *dupkey* is empty, the check will be ignored and not performed.

RULES_MAKE

The `%rules_make()` macro creates an output data set based on the intermediate data sets and variables as defined in the registry, starting with the core variables. The order of the variables is defined through the *meta* data set, if provided, or in the order they have been added to the registry.

```
%rules_make( registry = work._rules_registry_, out = , meta = , strict = Yes );
```

In addition to the registry parameter discussed above, the macro has the expected *out* parameter to specify the final output data set.

The macro will add a SAS warning or error, as specified by the *strict* parameter, to the log if a variable specified in the *meta* data set is missing from the registry, e.g. not derived.

META-DATA DRIVEN

The above macros are not dependent on meta-data beyond an option, but it is easy to extend the programming by rules approach to be in part or entirely meta-data driven. We can accomplish this by extending the data set specification with rule references and introducing a convention for referencing and selecting code.

The changes to the specification, most often in Excel, are subtle. We add three columns, a rule reference, rule options and an inclusion flag (Figure 1). The flag can be a simple Yes or No, or more extensive to build on the CDISC conventions of Required, Permissible, and the always unsophisticated Not Applicable.

Rule	Rule Options	Include	Deriv
dm_vs_1_2		Required	
dm_vs_1_2		Required	
dm_vs_1_2		Required	
dm_bsa_3_1		Required	

Figure 1. Updates to the specification

One important consideration is either to ensure that the rule documentation corresponds to the derivation description in the specification or that the description is removed and reference the rule documentation.

There are different perspectives on how to ensure that rule references in the specification are valid and correct. A simple approach to avoid misspelled references and copy-paste errors is to perform simple validation, which is the role of the `%rules_meta_validate()` macro.

```
%rules_meta_validate( specification = , out = , report = pdf );
```

The macro in its current form is quite primitive and will only verify that a macro or piece of SAS code that is referenced exists, but not that it is the correct SAS code. Further development is being considered that would extend the macro with features to perform checks for rule options, more extensive parameter checks and any other environment attributes.

RULE MACRO APPROACH

We hinted at a standard macro that implements a pre-defined rule in the previous section. A rule macro is a very simple, short and concise regular SAS macro that follows a simple convention. The macro is designed to only implement a single rule with as few options as possible. If there is a similar derivation rule, this is implemented as a separate rule macro. This can be considered a naïve approach, but experience has shown that moving the focus to specifying concise rules in the rule book and specification will greatly improve efficiency and future maintainability. It also focuses effort on converging and achieving a standard in analysis.

The rule macro follows a set naming convention and structure with three reserved macro parameters *data*, *out* and *instep*. Additional parameters are assumed to be rule specific.

```
%my_rule_macro( data = , param1 =, param2 =, … , out = , instep = );
```

The *data* and *out* parameters define the input and output data sets, respectively. If multiple input data sets are required, then the additional input data sets are considered rule specific and would be specified as rule specific parameters. In the latter, the *data* parameter would refer to the primary or main input data set.

The *instep* parameter is reserved for future use to allow a rule macro to be combined with other rules macros within the same DATA step and procedure to improve the execution efficiency. Experimental implementations have shown promising results but are not stable for mainstream use.

The remaining macro parameters, e.g. *param1*, *param2*, etc., are used to define variable references and other rule options that are specific to each rule macro.

A sensible approach is to implement a simple naming convention for the rule macros. We could just call the macros `%rule1()`, `%rule2()`, etc. but that can make finding and managing the collection of rule macros a daunting task as the number can easily enter the thousands. If you consider that each macro is short, concise, testable and you are required to maintain the code anyway in regular SAS programs, then the benefits are tangible,

Popular naming conventions include both the context, e.g. SDTM, ADAM, etc., and the data domain. For example the macro `%rules_sdtm_dm_age_1()`, or simply `%sdtm_dm_age_1()`, would derive AGE and AGEU in the SDMT DM domain. The 1 in this example is the variant, e.g. first derivation rule for AGE, as one other may exist for paediatric studies. If a version identifier should also be included, something like `%sdtm_dm_age_1_1()` could suffice. The basic idea is that the naming convention is flexible to your process but consistent and self-referencing.

Validation is also significantly simplified as each macro contains a single rule with no or very restricted options. As a consequence, the required test cases for each macro are surprisingly few and very well defined, which results in very rapid development and code that easier to maintain and test.

META-DATA GENERATING SAS PROGRAMS

The concept of having meta-data generate programs is not far off given that we have a pattern for rule programming, convention for standard rule macros and the documentation to drive the process.

```
%rules_make_program( meta =, output =  );
```

The *meta* parameter is of course the reference to the meta data set. Output location provides a reference to the folder where the generated programs will be created. Simple, but then there must be caveats.

There will be situations where the set of rule macros will not include all the rules required for an analysis. It may a decision that only 80% of the derivations should be covered by rule macros or simply a rule macro is not available at the time of analysis. One approach is for the `%rules_make_program()` to automatically include a "custom" program just prior to creating the output data set.

The `%rules_custom()` macro is a simple solution that will look for the "custom" SAS program file in the current folder and `%include` its content, if it exists. This will allow a system to auto-generate and maintain a SAS program based on meta-data alone, while at the same time allow for manually maintained code.

A natural extension would be to wrap this entire process, from a book of rules to meta-data generating programs, with a suitable interface and the promise of point-and-click reporting is a reality. The added benefit is that the approach retains a high degree of flexibility with regards to both analysis and the subsequent programming.

CONCLUSION

We all advocate, practice and adhere to programming by the rules set out in study protocols, Statistical Analysis Plans and associated specifications. In practice, we are practicing meta-data driven programming, or programming by rules, as we progress from source data sets to final report deliverables, but the path from one-off programs to SAS programs generated entirely from meta-data will required shifts in programming conventions, small new utilities and the predictable updates to input specifications.

The programming approach discussed highlights the results of taking a programming process from one-off programs to entirely meta-data driven with the flexibility you would expect. Add a simple meta-data interface and you have the ability to create study report deliverables with the click of a button while retaining a flexible analysis and programming process.

CONTACT INFORMATION

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

Magnus Mengelbier
Limelogic AB
Malmö, Sweden

E-mail: papers@limelogic.se
Web: www.limelogic.se