Paper SM02

MAIC-ing Efficient, Flexible and Maintainable code with Modular Programming

Michael Allie, MSD Switzerland

Abstract

The concept of modular programming is a simple one. It involves organising the code into self-contained programs, known as modules, that perform a specific sub-task. These modules can be thought of as building blocks that collectively form a larger structure, or as the individual instruments in an orchestra. They can be arranged, reused, or exchanged as required. While each module is simple in isolation, they can accomplish complex tasks when cleverly combined.

A Matching Adjusted Indirect Comparison (MAIC) analysis provides an excellent example of the advantages of a modular approach. The overall process can be broken down into a list of tasks, each of which will vary in predictable ways between different analyses.

This paper uses MAIC to demonstrate modular programming design and implementation. Additionally, advanced SAS™ macro techniques are also highlighted for developing efficient, flexible, and maintainable code.

Introduction

Matching-adjusted indirect comparisons (MAICs) are a type of Indirect Treatment Comparisons that use IPD [Individual Patient Data] from clinical trial of one treatment to match the trial population to the baseline summary statistics reported from another trial of another treatment. After matching, by using an approach similar to propensity score weighting, treatment outcomes are compared across balanced trial populations[1, 2]. MAICs (and other types of indirect treatment comparisons) are broadly used in Health Technology Assessment submissions to inform the comparative benefit of one intervention over another in the absence of a head-to-head trial.

To focus on the theme of modular programming, this paper will not explain the statistical methodology of MAIC, however some statistical papers are included in the references at the end for more information.

Programming code can generally be organised into two design systems: **monolithic** and **modular**.

A monolithic approach involves creating a single program to handle the entire process. In this MAIC example, this would be everything from reading the external input data to creation of the final outputs.

A modular approach consists of multiple individual programs that each perform a specific task within the process.

These concepts are not specific to any programming language. While modular programming is more traditionally associated with software/application development, there are benefits to be found in implementing modular systems within statistical programming.

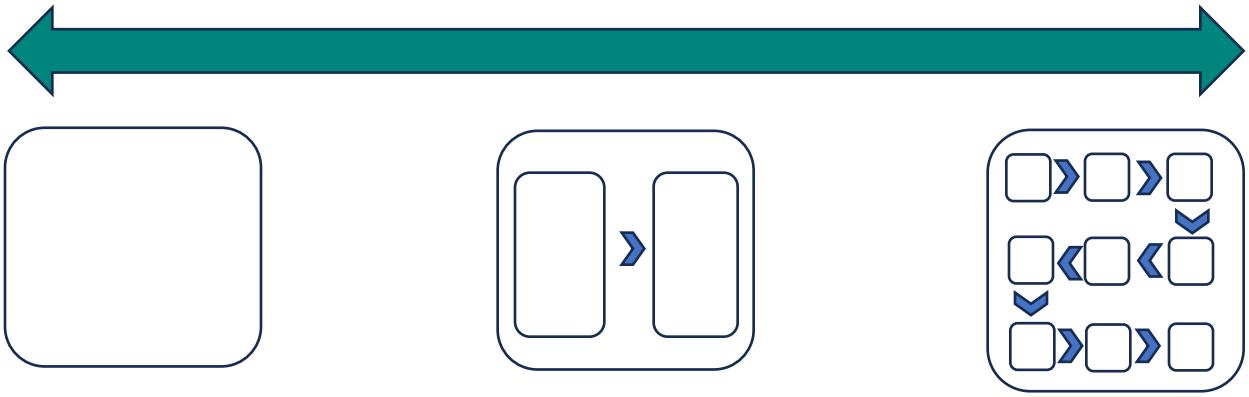
While these are distinct systems, in practice most approaches will fall somewhere on a spectrum between a completely monolithic design and a highly modular design. An adhoc and/or exploratory analysis which is only performed once would be appropriate for a monolithic design while frequently created tables, listings and figures (TLFs) are usually served by dedicated macros.

Complete Monolithic Design:

- Single program to perform all tasks.

Highly Modular Design

- Many programs which each perform a specialised task within the overall process.



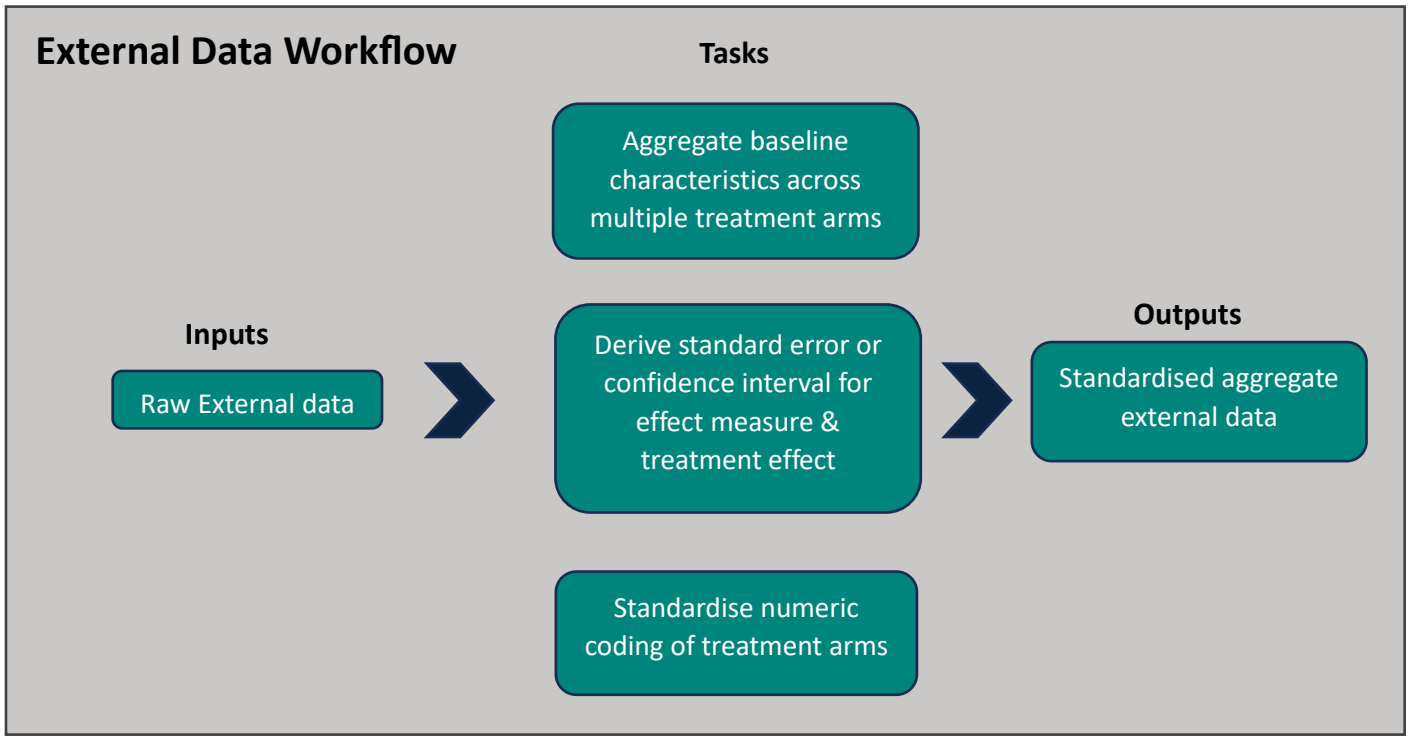
Identifying Potential Modules

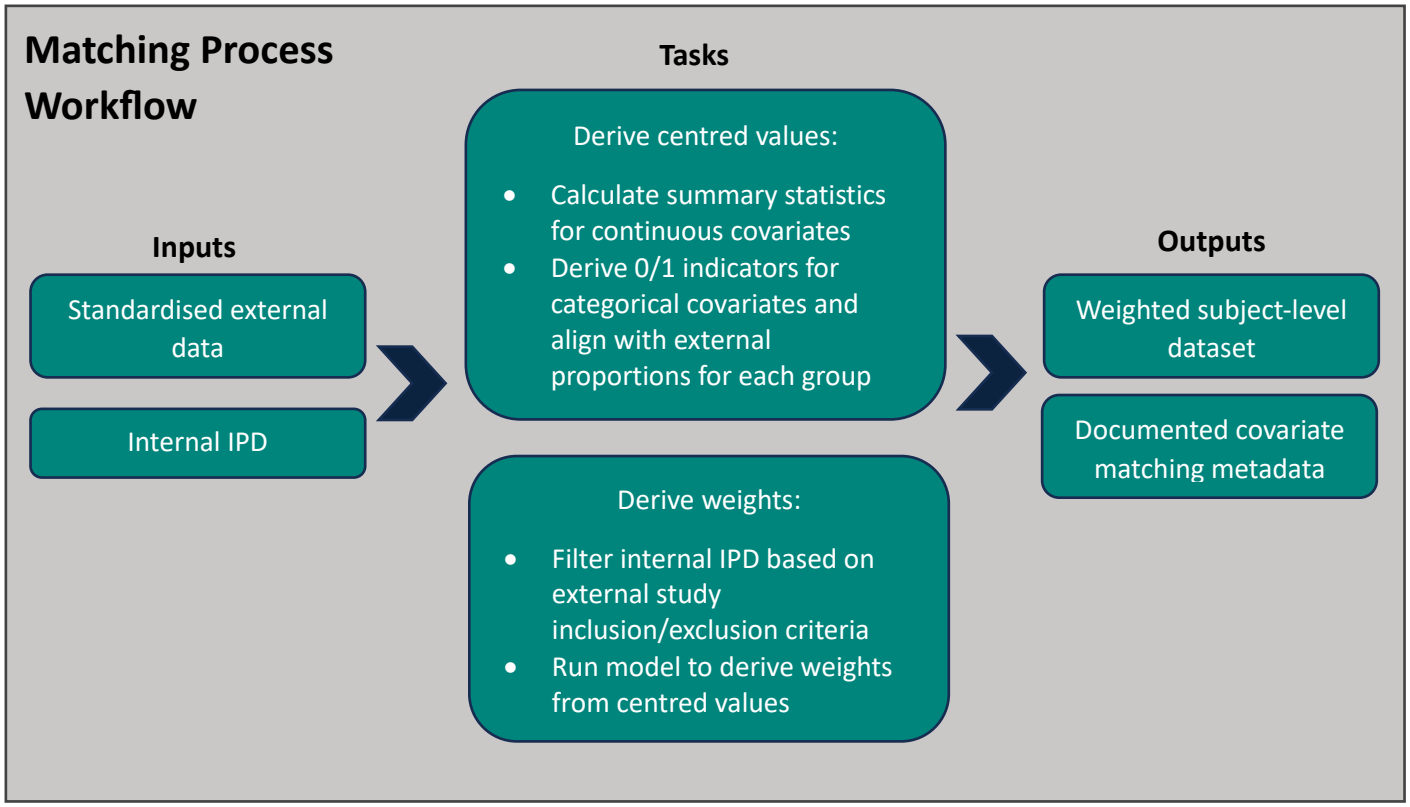
The most difficult part of designing an effective modular system is determining an appropriate level of modularisation. Creating too few modules to support a complex overall process offers no real advantage than a completely monolithic design while an excessive number of modules can obfuscate the logic and create confusion through the "black box" effect, where logic is not immediately visible by a program that calls many different modules. Intuitively naming and designing modules based on the various steps involved in the overall process can help to alleviate this concern. This could be achieved through naming convention (e.g. macros performing analysis start with "a", or tabulation macros beginning with "t") or though verb-based naming (e.g. "count", "pvalue" etc.).

A helpful initial step in deciding the modules that will form the final system is to clearly breakdown the overall process from beginning to end.

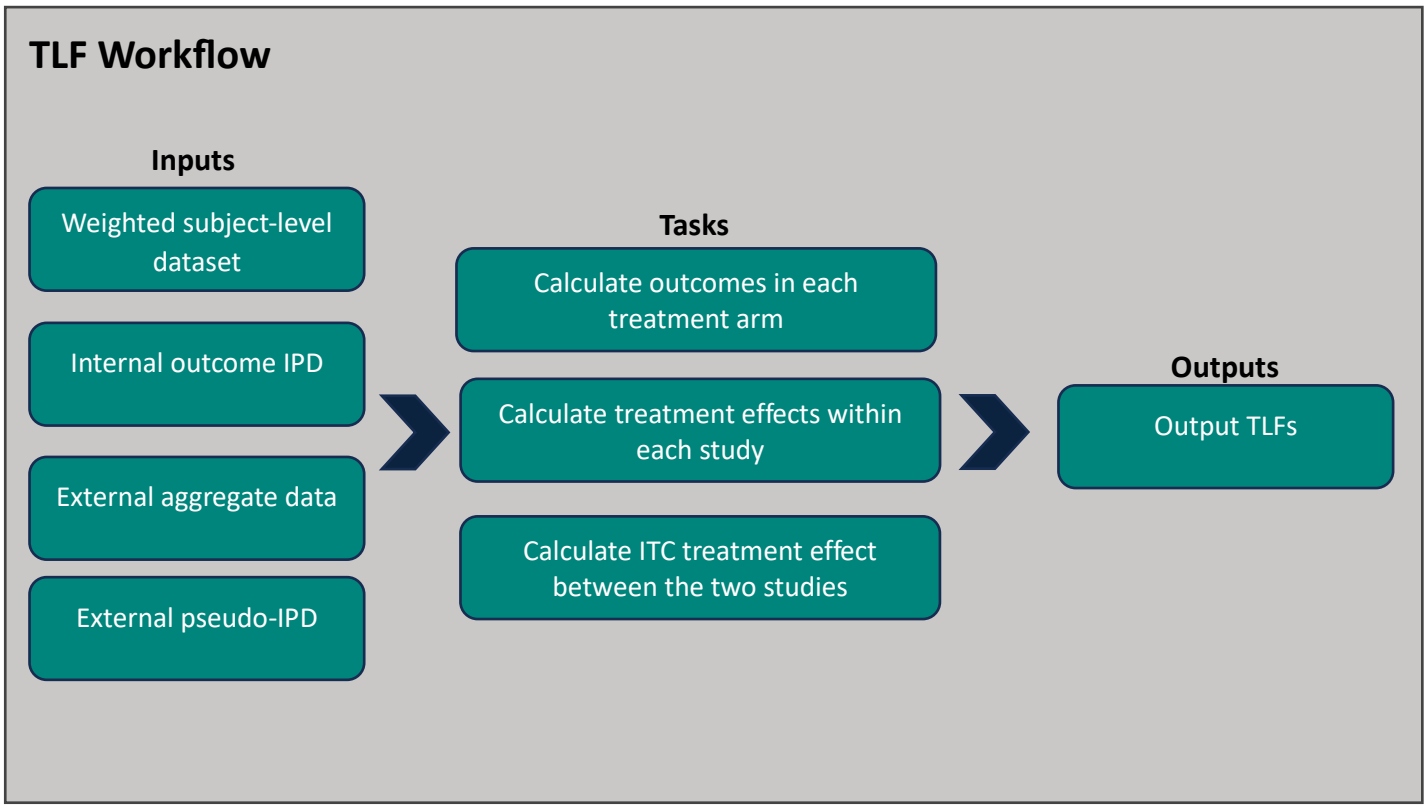
The overall process can be broken down into discrete workflows containing inputs, tasks, and outputs. A good rule when dividing a complex process into workflows is to consider intermediary items that should be validated. Items which are created via complex derivations or contain specific requirements to enable subsequent tasks should have defined workflows to create them. These are natural breakpoints to confirm everything is working as intended and they could be used to easily divide work between multiple people if desired.

For our MAIC example, the process can be divided into three workflows for processing external data, performing the matching process and creation of TLFs as shown in the figures below.





Below is a general TLF workflow for MAIC analysis; more specific workflows for each kind of MAIC analysis (e.g. anchored vs. unanchored, binary/continuous vs. time to event etc.) could be created in similar manner.



From the workflow level view, modules can then be designed based on the individual tasks. Some general points to consider are:

1. How complex is the task?

   Tasks with high complexity are always worth considering for a dedicated module so that they can be closely validated, independent of all other inputs and outputs.

2. Is this task repeated elsewhere?

   One of the largest benefits to modular design is the ease of reusing code in multiple places. Not only does it reduce the overall lines of code compared to copy/pasting verbatim code, but it also aids in maintainability. All future updates to a module can be implemented in a single place and will always propagate through the system wherever the module is used.

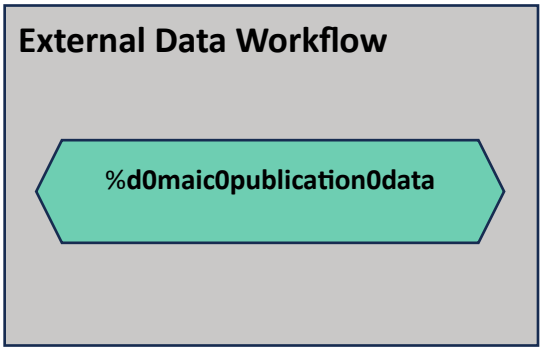
3. Is there a high probability of structured variation for this task?

   "Structured variation" in this context is a change in value, but not structure of an object. Especially where there is no singular correct derivation and it is variable based on the requirements of analysis.
   Anchored MAIC analyses provide an example of this concept; there are many different techniques to calculate a treatment effect, but the indirect treatment comparison (ITC) estimate of two treatment effects can be derived in the same way regardless of the individual study treatment effects. If calculation of treatment effects is contained within a module, it can then be substituted for a different module depending on what treatment effect and/or statistical method is needed.
   Having smaller and more specialised modules which are executed mutually exclusively can lower the overall complexity of the codebase due to less reliance on logical branching.

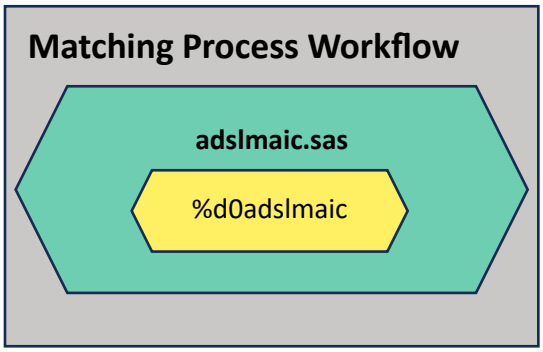
MAIC Analysis Modular System

Below is a breakdown of the different modules created to conduct MAIC analyses. They are separated into two types: modules related to data creation/standardisation and modules which generate the final TLFs.

Data modules



Due to relative simplicity in the individual tasks of the external data workflow, they are all handled by a single SAS macro.

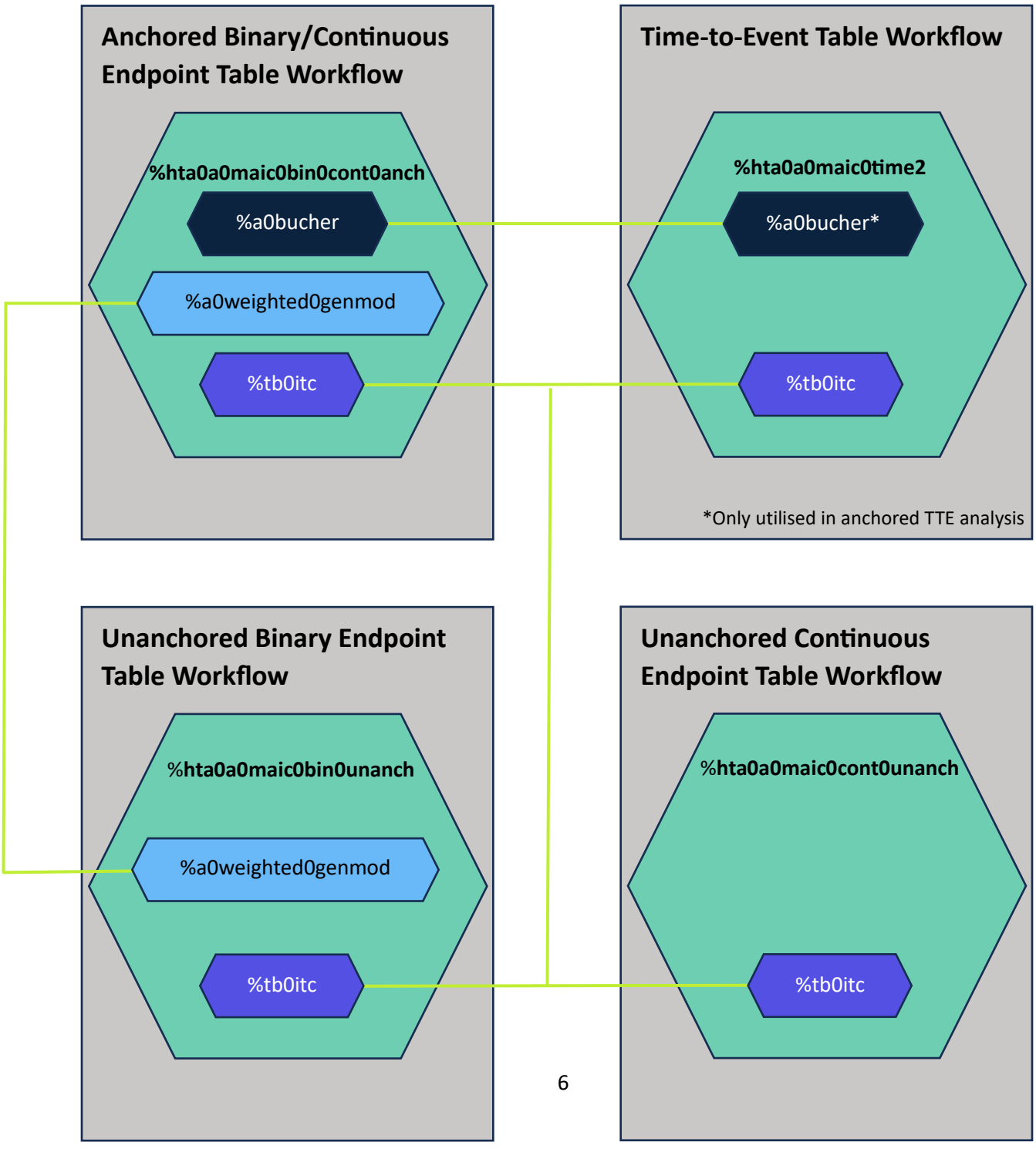


Creation of the weighted subject-level dataset (adslmaic) is handled by an open code SAS program to allow flexibility in the statistical model for the derivation of the weights. It utilises a helper macro %d0adslmaic which handles the more mechanical tasks relating to the metadata: calculating centred values and capturing the relationship of categorical values to external matching proportions.

Table modules

There are four primary TLF templates supported by the MAIC modules. The figure below shows the different modules that handle each workflow. There are three types of macros which make up table modules:

1. Wrapper macros (hta0xxx) are user facing macros which generate TLFs from analysis data.
2. Analysis macros (a0xxx) are submacros which implement statistical methods for a given TLF.
3. Tabulation macros (tb0xxx) are submacros which structure the analysis results for display and create the final .rtf file.



One of the key objectives within this design was to reuse logic as much as possible without copying verbatim code. The a0bucher module computes the ITC using Bucher method[3] for all anchored analyses independent of the specific endpoint because the comparison of the treatment effect point estimates does not depend on the method used to calculate the estimates. Similarly, a single tabulation macro tb0itc controls the display for all the MAIC outputs. There are some minor differences between the layout of the outputs, but they are similar enough that a single module is sufficient.

One of the unique aspects of MAICs is the variety of supported endpoints; it is a method which involves generating a point estimate from IPD and indirectly comparing to a published endpoint. This means that there are many statistical methods which could be included, so another objective was to provide an extensible platform for performing MAICs using a variety of endpoints. The diagram above references the module a0weighted0genmod as the primary analysis driver, but this is actually a dynamic argument in the wrapper macros %hta0a0maic0bin0cont0anch and %hta0a0maic0bin0unanch.

There are two parameters that facilitate this:

```
%hta0a0maic0bin0cont0anch(

  ...

,analysis_macro_name=new0method0xxx
,analysis_macro_extra_params=%str(,param1=value1,param2=value2)
);
```

- analysis_macro_name: Name of analysis macro (%adhoc0method0xxx in the example above) to compute individual arm effect measures and intra-study treatment effects
- analysis_macro_extra_params: A string in the form (,param1=value1,param2=value2) for parameters unique to &analysis_macro_name. This allows the use of additional parameters for maximum flexibility.

Then inside the wrapper macros, the analysis macro is called 2-3 times depending on the analysis. Always at least twice for adjusted/unadjusted results and potentially an additional time for the external data when performing an anchored analysis and using pseudo-IPD.

There is then just a single challenge remaining to solve; since the analysis macro and its parameters are only specified once when the wrapper macro is invoked, these values cannot differ between the multiple times the analysis macro is executed by the wrapper, which is not optimal for our purposes.

To allow for different logic between the calls of the analysis macro (e.g. something unique to adjusted/unadjusted/external analysis), the wrapper macros create a macro variable analysis_call_id, which takes the value 1 for the unadjusted analysis call, 2 for adjusted and 3 for external. This value can then be used within a generic analysis macro to branch the logic if required.

```
%*-------------------------------------------------------------------------------*;
%*- Step 07: Before matching (unadjusted) statistics                            -*;
%*-------------------------------------------------------------------------------*;

%* Create macro variable visible to analysis submacros which identifies before/after matching and external analysis;
%let analysis_call_id=1;
%&analysis_macro_name.(indata=_observation
                      ,outdata=int_bef
                      ,te_label=%str(&effect_type. (Before Matching))
                      ,subject_var=&subject_var.
                      ,weight_var=
                      ,therapy_data_var=&therapy_data_var.
                      ,reference_therapy_data_varn=2
                      ,stratum_var=&stratum_var.
                      ,outcome_var=&outcome_var.
                      %if %length(&var_event_val.) %then %do;
                        ,var_event_val=&var_event_val.
                      %end;
                      %if %sysfunc(indexw(BEFORE BOTH,%upcase(&use_robust_variance.))) %then %do;
                        ,robust_variance=Y
                      %end;
                      %else %if %length(&use_robust_variance.) %then %do;
                        ,robust_variance=N
                      %end;
                      ,by_var=&by_var.
                      ,effect_type=&effect_type.
                      ,alpha=&alpha.
                      ,debug=Y
                      %unquote(&analysis_macro_extra_params.)
                      );
```

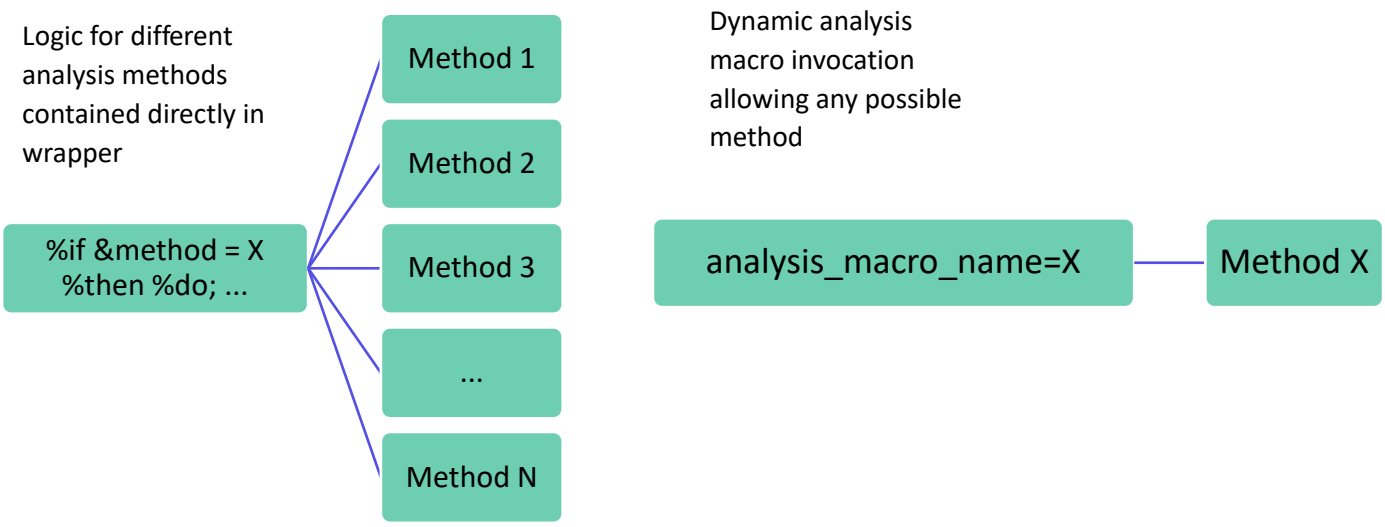
The %unquote() function is used here so commas are recognised by the macro processor when calling the analysis macro. %str() is used to mask the commas when calling the wrapper macro

With this system in place, any future analysis macro may be used as long as it meets the following requirements:

1. The macro must contain at least the core parameters specified above.
2. The output dataset structure is specific to the requirements of the subsequent logic in the wrapper macro.

These requirements provide minimal restrictions to the statistical methodology or endpoint, allowing relatively quick and simple implementation of varied analyses to support future project needs. New modules could be developed and validated independently from the wrapper macros. This reduces the amount of up versioning and regression testing needed and also keeps the logical structure of the wrapper macros relatively flat. The alternative would be many branching conditions checking the method specified at macro invocation which can reduce the overall readability.

This dynamic design allows new analysis methods to be integrated into the existing MAIC process without modifying any of the existing code. All that is required is that the new analysis macro is written and specified when executing the wrapper macros.



Conclusion

Although the concept is traditionally associated with software engineering or application development, a modular programming approach offers many advantages within statistical programming analyses such as MAIC. There may be some additional upfront effort involved in the design, but the final system is significantly easier to maintain and enhance further. By abstracting the statistical method used to calculate the effect measures and treatment effects into a replaceable module, new methods can be independently developed as separate modules and seamlessly integrated.

Separating the logic into individual modules does have some downsides. Inheriting and/or initially learning such a system requires reading all the individual documentation. In a monolithic design, all the code can be inspected in a single place which can make it easier to gain an understanding of all the low-level logic. Intuitively named modules based on the actions they perform (e.g. “tabulate”, “plot” etc.) help to preserve an understanding of the overall process, but any deep dive into the logic will require looking into each module. Avoiding excessive modularisation is key to minimising these potential negatives.

Depending on the size of the system, it can be beneficial to programmatically document the interdependencies between modules. Techniques such as network graphs can be applied to large systems to provide insight into the interconnectivity of the various modules[4].

References

[1] Signorovitch J.E. et al. Matching-adjusted indirect comparisons: a new tool for timely comparative effectiveness research. Value Health. 2012; 15 (6): 940-7

[2] Signorovitch J.E. et al. Comparative effectiveness without head-to-head trials: a method for matching-adjusted indirect comparisons applied to psorias treatment with adalimumab or etanercept. Pharmacoeconomics. 2010; 28(10): 935-45.

[3] Bucher HC, Guyatt GH, Griffith LE, Walter SD. The results of direct and indirect treatment comparisons in meta-analysis of randomized controlled trials. Journal of clinical epidemiology. 1997 Jun 1; 50(6): 683-91.

[4] Christof Binder, Hoffman-La Roche. Looking at SAS code as data; PhUSE EU Connect 2017 (https://phuse.s3.eu-central-1.amazonaws.com/Archive/2017/Connect/EU/Edinburgh/PAP_DV08.pdf)

Further reading

David M. Phillippo, A. E. Ades,1 Sofia Dias, Stephen Palmer, Keith R. Abrams, Nicky J. Welton. NICE DSU TECHNICAL SUPPORT DOCUMENT 18: METHODS FOR POPULATION-ADJUSTED INDIRECT COMPARISONS IN SUBMISSIONS TO NICE. 2016

Acknowledgements

I would like to extend my thanks to the Relative Effectiveness Initiative team. The work presented in this paper is the result of the collective effort and expertise of all collaborators:

Caterina Ferioli, Lidia Mukina, Gosia Wojtecka-Glod, Justin Chumbley, Ram Gaduputi, Rachid Massaad, Carl Herremans, Emma Crawford, Biniam Hailu, Dave Gelb, David Maher-McWilliams, Shahrul Mt-Isa, Mikkel Oestergaard

Contact Information

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

Michael Allie
MSD
The Circle 66
8058 Zürich
Switzerland
Email: michael.allie@msd.com