

SM02: MAIC-ing Efficient, Flexible and Maintainable code with Modular Programming

Michael Allie, MSD, Zurich, Switzerland

Phuse EU Connect 11 November 2024

Agenda

- 1 What is MAIC?
- 2 Monolithic and Modular Programming
- 3 What Makes a Good Module?
- 4 Workflows
- 5 MAIC Modular System
- 6 Abstraction of the Statistical Methods
- 7 Pros & Cons of Modularisation
- 8 Conclusion

What is MAIC?

What is MAIC?

MAIC [Matching-adjusted indirect comparison] is a type of Indirect Treatment Comparison [ITC] which can be used when you have IPD [Individual Patient Data] for one study and aggregate data [AgD] for a comparator study

MAICs use IPD from trials of one treatment to match baseline summary statistics reported from trials of another treatment. After matching, by using an approach similar to propensity score weighting, treatment outcomes are compared across balanced trial populations^{1, 2}

Many statistical procedures in SAS accept a WEIGHT statement where a weighting variable can be applied to derive weighted results

The **ITC** consists of comparing a point estimate for an endpoint calculated from the **IPD study** with a **published one**

Study Treatment Arm	Before Matching					After Matching ^a				
	N ^b	Participants with Event n (%)	Odds Ratio ^c [95%-CI]	ITC Odds Ratio ^d [95%-CI]	p-Value ^e	N ^f	Participants with Event n ^g (%)	Odds Ratio ^h [95%-CI] ^a	ITC Odds Ratio ^d [95%-CI]	p-Value ^e
Internal Study 001 ⁱ										
Examplezumab 1mg	79	64 (81.0)	2.56 [1.26; 5.20]	2.66 [1.17; 6.03]	0.019	57.0	40.9 (71.9)	1.10 [0.43; 2.78]	1.14 [0.41; 3.15]	0.797
Chemotherapy 1mg	88	55 (62.5)				62.2	43.5 (69.9)			
External Study 002 ^j										
Active Treatment + Chemotherapy	271	211 (77.9)	0.96 [0.64; 1.45]			271	211 (77.9)	0.96 [0.64; 1.45]		
Chemotherapy	270	212 (78.5)				270	212 (78.5)			

IPD



60% severe patients in IPD study

AgD



80% severe patients in comparator study



MAIC weighting



Effectively 80% severe patients in IPD study (after weighing)



80% severe patients in comparator study (no change)

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 psoriasis treatment with adalimumab or etanercept. Pharmacoeconomics. 2010; 28(10): 935-45.

Monolithic and Modular Programming

Monolithic and Modular Programming

Programming code can be organised into two major design systems: Monolithic and Modular

In practice, most codebases are modularised to at least some degree (e.g. separate SDTM/ADaM/TLF programs)

Modularisation is not specific to any programming language; the example from this presentation is SAS™ based but it could be applied to other languages such as R or Python

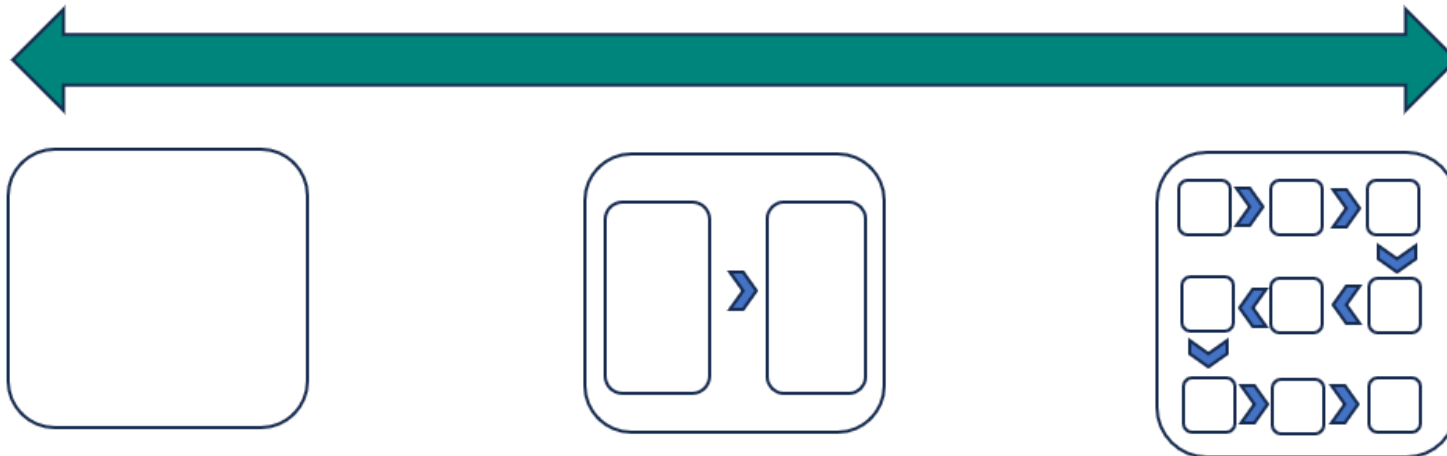
Monolithic Design

- “One to rule them all”
 - All logic contained within a single program



Highly Modular Design

- “I’m doing my part”
 - Code separated into different “modules” which each perform a specialised task



What Makes a Good Module?

What Makes a Good Module?

There are three primary factors to consider when deciding whether or not to create a module

1. Can this be used elsewhere?

- **Tasks that are performed frequently can make good candidates for modules**
 - Reduces the total lines of code due to less copy + pasting
 - Any updates can be performed in a single place and will cascade to all places where module is executed

2. How complex is the task that's being performed?

- **Tasks with high complexity are often better suited to specialised modules**
 - Allows for separate validation which can be completed without executing the entire process (Unit testing)

3. How likely is the method used to perform this task to change?

- **Tasks which could be performed by alternative, equally valid methods should be considered for modularisation**
 - When a different method is needed, the module can be exchanged for another without updating the rest of the codebase

Workflows

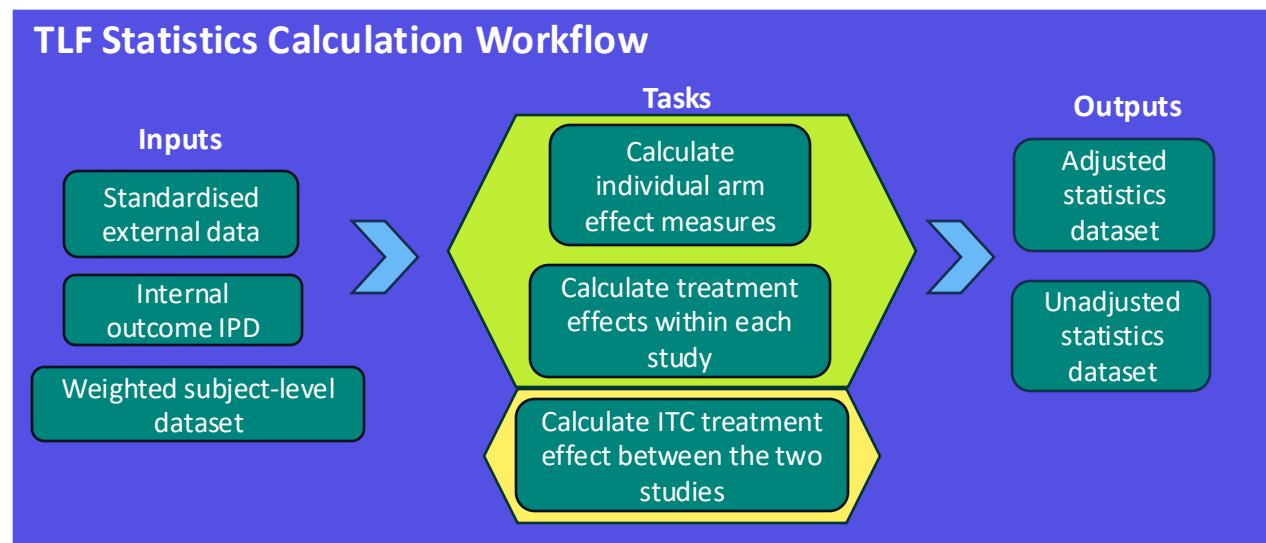
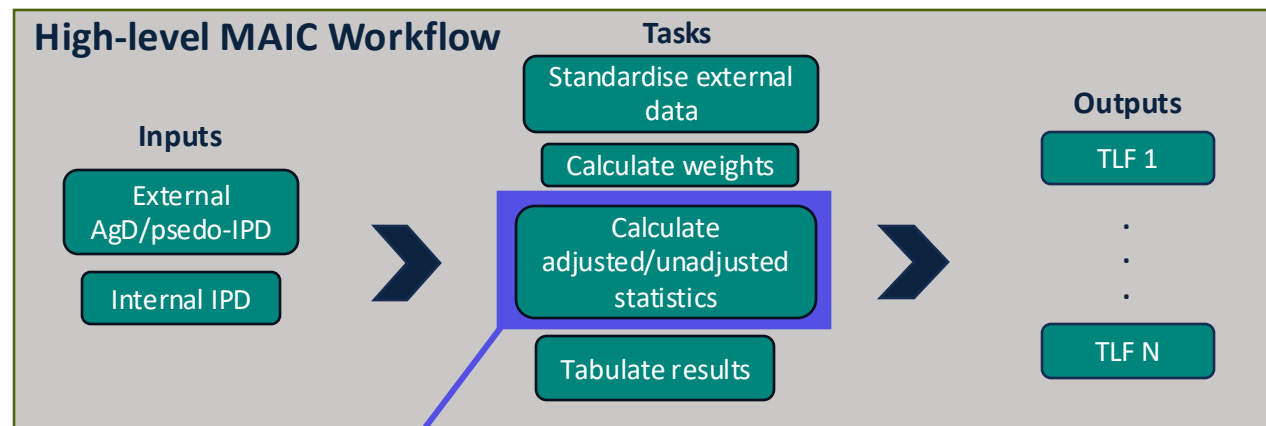
Workflows

A helpful step in identifying opportunities for modules is to create a clear overview of the end-to-end process

Start by defining a workflow diagram listing inputs, tasks and outputs

Any task, or the creation of any input could be further subdivided into its own workflow

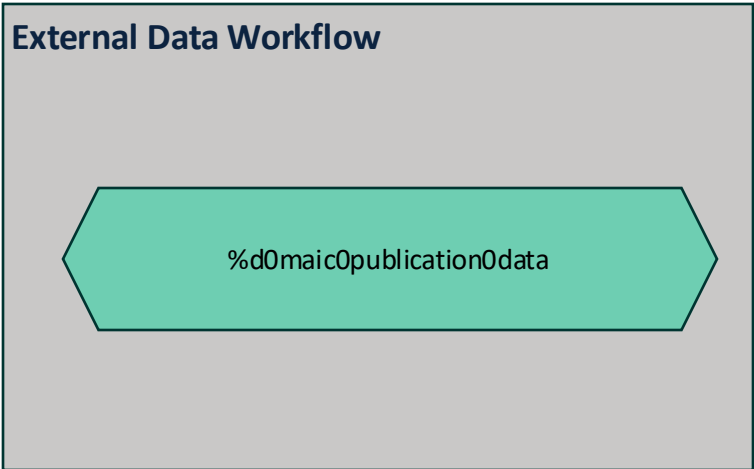
Modules can then be designed to perform one or more low-level tasks depending on repetition, complexity or close relation of tasks



MAIC Modular System

MAIC Modular System: Data Modules

External Data Workflow



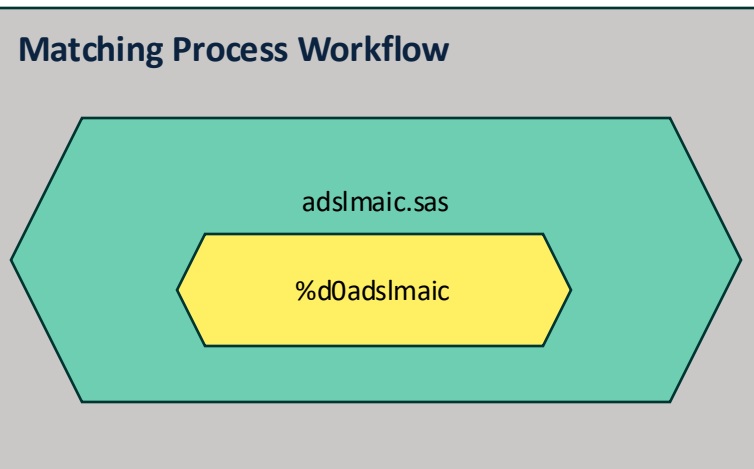
The diagram shows a light gray rectangular box containing a teal-colored hexagon. Inside the teal hexagon is the text "%d0maic0publication0data".

%d0maic0publication0data

All tasks related to importing and preparing the external data are handled by a single SAS macro:

1. Importing the data from a spreadsheet
2. Aggregating together baseline statistics from different treatment arms
3. Calculating supplemental statistics (standard error or confidence interval) needed for performing the ITC on the endpoint data
4. Assigning standardised numeric coding to treatment arms so downstream macros can automatically identify comparator and anchor treatments

Matching Process Workflow



The diagram shows a light gray rectangular box containing a teal-colored hexagon. Inside the teal hexagon is the text "adslmaic.sas". Within the teal hexagon is a yellow hexagon containing the text "%d0adslmaic".

adslmaic.sas

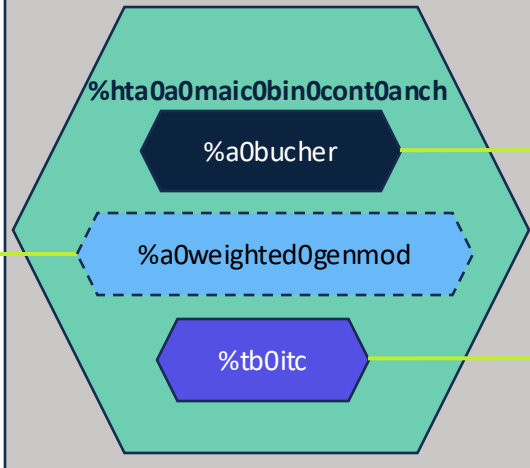
%d0adslmaic

The matching process (where the weights are calculated) is performed in an open code program, with a helper macro which performs some of the more logistical tasks:

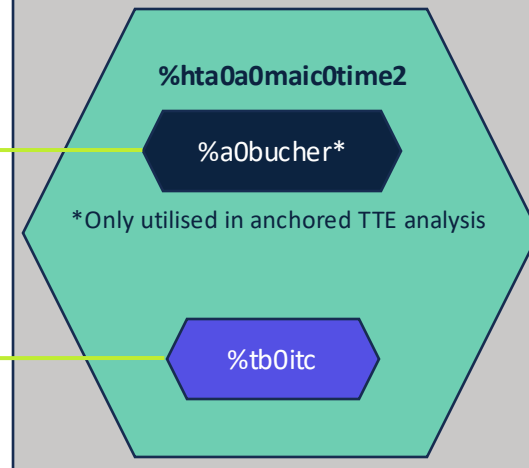
1. Creation of binary indicators for categorical values
2. Derivation of centred values
3. Documenting matching metadata in dataset for use by downstream macros

MAIC Modular System: TLF Modules

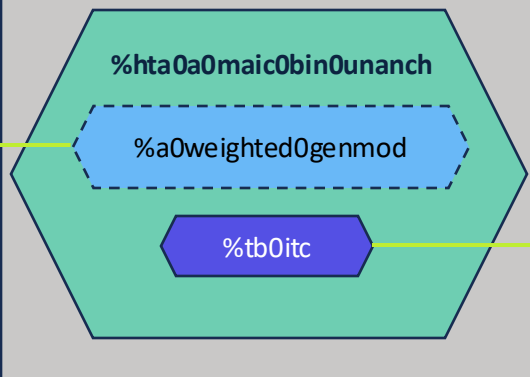
Anchored Binary/Continuous Endpoint Table Workflow



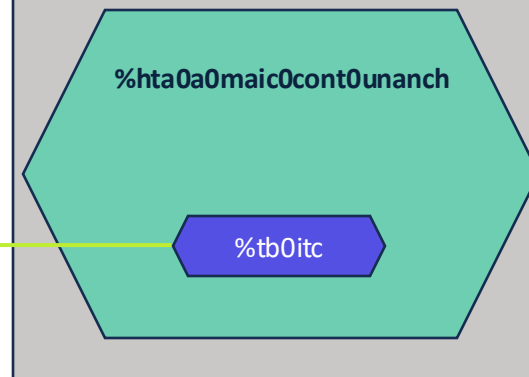
Time-to-Event Table Workflow



Unanchored Binary Endpoint Table Workflow



Unanchored Continuous Endpoint Table Workflow



Each major MAIC TLF template has its own workflow.

The macros involved are categorised into three types:

1. Wrapper macros (hta0xxx): These are user facing macros that create TLFs from analysis datasets. They may call other types of macros as submodules
2. Analysis macros (a0xxx): These macros are called within wrapper macros to perform statistical calculations required for the TLF
3. Tabulation macros (tb0xxx) These macros structure the statistical results, prepare them for display and create the final TLF

Tasks common across multiple workflows helped to determine module design

Abstraction of the Statistical Methods

Abstraction of the Statistical Methods: Fixed String Macro Invocation

- Invoking the analysis macro in the traditional manner, with a fixed string (e.g. `%a0weighted0genmod`) requires branching the logic within the wrapper macro for each supported analysis macro
- This means that integrating any new analysis macros will require an update to the wrapper macro



```
%a0weighted0genmod(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_var=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
);
```

`%if &method = X
%then %do; ...`

Method 1

Method 2

Method 3

...

Method N

Abstraction of the Statistical Methods: Dynamic Macro Invocation

- Analysis macros may instead be invoked dynamically (e.g. `%&analysis_macro_name.`), as a parameter of the wrapper macro
- This allows additional analysis macros to be integrated into the wrapper macro without requiring any updates to the wrapper macro itself. The logic within the wrapper macro maintains a flat structure independent of the number of supported analysis macros
- Due to MAIC requiring effect measure/treatment effect calculations to be performed twice (for adjusted/unadjusted), this introduces some additional complexity in implementing this strategy

```
%&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_var=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
);
```

An additional macro variable is created by the wrapper macro which marks each invocation of the analysis macro. This can be used within the analysis macro to control logic specific to adjusted/unadjusted analysis

analysis_macro_name=X

Method X

Additional parameters may be specified when invoking the wrapper macro through a parameter in the form `%str(,param1=value1,param2=value2)` which can then be unquoted for use by the analysis macro. This is similar to the `"..."` functionality in R functions

Pros and Cons of Modularisation

Pros and Cons of Modularisation

Pros

- Independent maintenance of modules
- Updates can be made in a single place and will be consistently applied wherever the module is used
- Allows multiple team members to work on modules in parallel
- Complex tasks can be individually validated without requiring the entire process to be executed
- General flow of modular code can be simpler to follow due to replacement of large sections of code with intuitively named modules

Cons

- Directly viewing the code for the entire system can be slower when it is split up into many different modules.
- Each module requires additional documentation
- Debugging systems with high levels of module nesting can be more difficult.
- Initial onboarding of new staff to a highly modularised system can be time consuming

Conclusion

Conclusion

- Higher levels of modularisation are not strictly better, however there are evident advantages to be gained through this approach when sensible modules are created based on your programming objectives
- Having a strong understanding of the overall process will provide a lot of help when designing the modules that make up a system
- Modular design can be effectively applied within statistical analysis, and it does not belong to any specific programming language. Examples can be found in many other languages such as the Golem framework for R Shiny application development

Acknowledgements

I would like to extend my thanks to the Relative Effectiveness Initiative team. The work presented today 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



Thank you

Email: michael.allie@msd.com

Copyright © 2024 Merck & Co., Inc., Rahway, NJ, USA and its affiliates. All rights reserved.