Paper SM03

Untangling the Dependencies Web

Rhys McCrosson, PHASTAR, Glasgow, UK
Simon Purkess, PHASTAR, London, UK

ABSTRACT

Clinical submissions often contain hundreds of datasets (split across raw, SDTM, ADaM and TFL) with a complex dependency flow linking datasets both within and between the different levels. Traditionally these links are recorded by programmers in a QC tracker or program header, but this is prone to human error and only provides a fragmented view of the full web of dependencies.

In this paper we explore ways of programmatically scanning dataset programs (including any external macros that are called) and specs to build a complete network of dependencies, which can be interrogated to answer a range of questions, including:

- Do the dependent datasets listed in the spec match the datasets that the production program is reading in?
- Is the QC program using the same datasets as production?
- Given a subset list of TFLs for an additional analysis, what subset of raw/SDTM/ADaM datasets are required?

INTRODUCTION

Understanding the flow of data through a clinical trial is crucial to ensure that data can be fully traced through the different levels of study reporting. To understand this data flow we must identify *dependencies*, i.e. any case where Program A must be run before Program B – generally because Program A creates a dataset which is used by Program B. Failure to identify these could result in Program B being run before Program A in a batch run, leading to programs either failing or running out-of-date data.

EXAMPLE STUDY AND ASSUMPTIONS

For the purposes of this paper, we'll be referring to an example study with four levels:

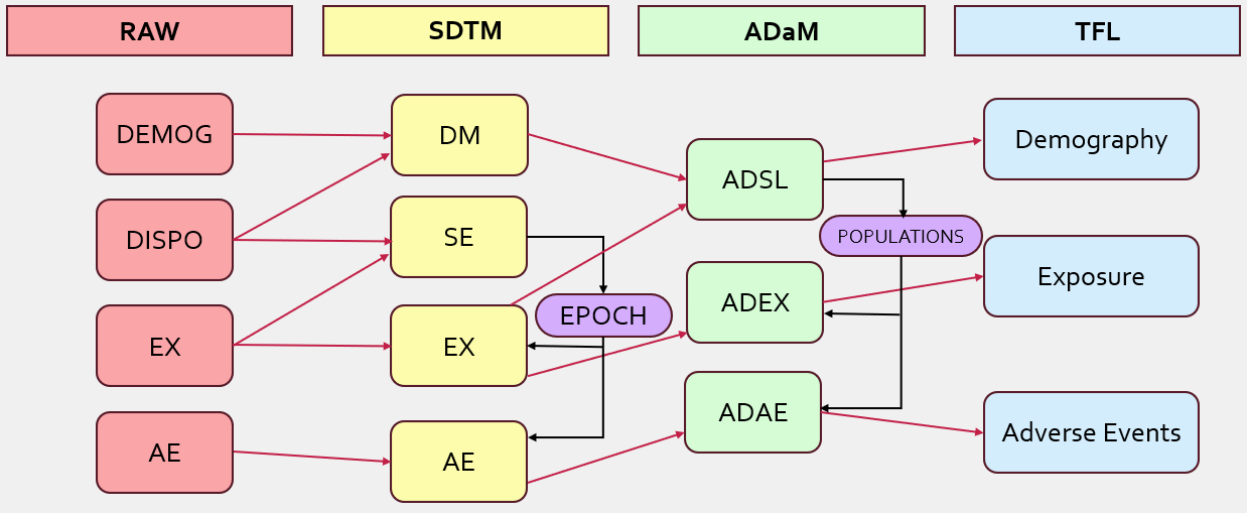
1. RAW: Data as collected by study sites
2. SDTM: Standardised tabulation datasets following the CDISC SDTM model
3. ADAM: Analysis datasets following the CDISC ADaM model
4. OUTPUT: Tables, Figures and Listings (TFL) displaying trial results

Each level (except RAW) contains programs which output datasets by taking various datasets (from the same level and/or previous levels) as inputs. For example, an SDTM program will create an SDTM dataset by manipulating other RAW and/or SDTM datasets.

This paper refers to SAS® programs and datasets, with datasets stored in libnames with names matching the levels listed above; e.g. the SDTM Exposure dataset is referenced as "SDTM.EX". However, the concepts explored could be generalised to other languages.

The diagram below shows a simplified dependency map, detailing the relationship between the study datasets. The red/yellow/green/blue rectangles represent datasets at the raw/SDTM/ADaM/Output levels respectively, while the purple shapes represent macros. Arrows show dependency relationships between two datasets, either directly or

through a macro.



TYPES OF DEPENDENCIES

CROSS-LEVEL VS WITHIN-LEVEL

Almost all datasets have dependencies on datasets from the previous level – SDTM datasets depending on raw datasets, ADaM datasets depending on SDTM datasets and TFL datasets depending on ADaM datasets. However, there are cases where dependencies skip a level (e.g. a TFL Listing programmed directly from an SDTM dataset with no ADaM input) or within the same level (e.g. SDTM datasets depending on SDTM.SE to derive the EPOCH variable), so it's important to include every level within your dependency search.

DIRECT REFERENCE VS INDIRECT

In most cases a dependent dataset can be found explicitly within a program (e.g. named within the SET statement of a data step), allowing it to be easily found by searching the program text. Unfortunately, not all dependencies are so easy to find! The use of macros means that dataset references often don't appear in a dataset's main program; for example, a study may employ a %EPOCH macro which references SDTM.SE in its program text, but a search of a program calling the %EPOCH macro wouldn't find the SDTM.SE reference.

IDENTIFYING DEPENDENCIES

Traditionally, dependencies have been identified and recorded manually, e.g. in program headers, dataset specifications or a QC tracker. This manual approach has a few issues when trying to compile the dependency data:

- Mistakes are almost inevitable due to human error
- Manual input means dependency data is stored in an inconsistent format which isn't machine-readable (e.g. one programmer may write "SDTM.EX, ADAM.ADSL" while another programmer writes "EX/ADSL" for the same dependencies)
- If dependency data is stored in individual files, it's hard to see a zoomed-out view of study dependencies

Alternatively, programs can be programmatically scanned for key-word tokens to identify an exhaustive list of valid dependencies:

TEXT-BASED SEARCH - APPROACH

Automatically identifying dependencies through use of keyword tokens comes with many challenges. All possible dependencies must be captured and then validated as follows:

1. A program must be scanned for source keyword tokens such as “RAW.”, “SDTM.” or “ADAM.”. Text that immediately follows a source keyword token should then be considered as a potential dependency.

2. Potential dependencies that are contained within comments or the program header should be excluded.

3. Macro calls must be identified within programs to find additional macro programs to search. Identified macro programs should be scanned for potential nested dependencies, analogous to steps (1) and (2).

4. Potential dependencies identified with the same name as the program being investigated should be excluded.

5. Some dataset references relate to an output dataset rather than an input dataset, so these must be identified and excluded. Remaining potential dependencies should be validated by searching for exclusion keyword tokens immediately preceding the source keyword tokens from step (1). Examples of possible exclusion terms include, but are not limited to: “DATA”, “CREATETABLE”, “OUT=”, “OUTPUT”.

TEXT-BASED SEARCH – CHALLENGES

Capturing all possible dependencies and distinguishing true dependencies from false dependencies involves a broad set of challenges. Issues include:

IDENTIFYING AND EXCLUDING DEPENDENCIES WITHIN COMMENTS:

- Comment start points must be identified along with their respective comment end points:

Example [1]: Iterative check to identify the start and end of comments

```
if countc(line,'*') = 1 and ^( index(line,'(') < index(line,'*') <
index(line,')')) then do;

      comment_start = index(line,'*');

      if index(line,';') > index(line,'*') then comment_end = index(line,';');

      else comment_end = length(strip(line));

    end;

    else do; %* Find comment start/end points *%;

      comment_start = .; comment_end = .;

      if index(line,'/*') > 0 then do;

        comment_start = index(line,'/*');

        comment_end   = index(line,'*/');

      end;

      if index(line,%nrstr('%*')) > 0 then do;

        comment_start = index(line,%nrstr('%*'));

        comment_end   = index(line,%nrstr('*%;'));

      end;

    end;
```

- Possible dependencies that are found between comment start/end points should be filtered out of the list of possible dependencies:

Example [2]: Filtering out tokens which appear within a comment

```
if (0 < comment_start < index(upcase(line),'ADAM.')    < comment_end)
       or (0 < comment_start < index(upcase(line),'SDTM.')    < comment_end)
       or (0 < comment_start < index(upcase(line),'RAW.')     < comment_end)
      then comment = 1;
      else comment = 0;
```

- Comments that span multiple lines pose an especially difficult challenge. Once a comment start point is identified, all text thereafter should be considered a part of a comment, and a comment flag should be used and retained across program lines until the respective comment end point is found.

- SAS supports several different comment formats:
  - the `*asterisk comment;`
  - the `/*block comment*/`
  - the `%*macro comment;`

  These formats each behave slightly differently in situations including nested comments and macro tokens, so they must be considered individually.

IDENTIFYING MACROS:

- Possible macros can be identified through scanning programs for instances of a percentage sign and storing text that immediately follows the percentage sign, using a semicolon or a closed parenthesis as an end point for the macro keyword token.
- Possible macros need to be validated to check if they are true macro calls or if they are contained within a comment or even a quote.
- SAS built-in macros should be excluded from the list of identified macros as they cannot contain dependencies.

OBTAINING DEPENDENCIES THAT ARE NESTED WITHIN MACROS:

- When a list of dependent macros has been identified from a program, they must be iteratively scanned using the same process of searching for a source keyword token (e.g. "**RAW**.", "**SDTM**.", "**ADAM**."). Potential dependencies must then be validated again to determine if they are true dependencies.
- A shortcut to obtaining dependencies for some specific macros can be to scan the code from a macro call directly. A list of macros would need to be identified for this process. If a macro belonging to this list was identified in a program, then the identified macro call would be extracted, and the parameters would be investigated for use of a dependency.

UNRESOLVED MACRO VARIABLE DEPENDENCIES:

Some dependencies may end up being identified as a macro variable, which needs to be resolved to be understood. For example, consider this utility macro %COMBINESDTM which merges an SDTM supplemental dataset onto its parent domain:

Example [3]: %COMBINESDTM macro code

```
%macro COMBINESDTM(domain);

      %* Transpose supplemental dataset *%;
      proc transpose data=SDTM.SUPP&domain. out=supp_trn (drop=_name_ _label_);
        by studyid usubjid;
        id qnam;
        idlabel qlabel;
        var qval;
      run;

      %* Merge supplemental variables onto parent domain *%;
      data &domain._out;
        merge SDTM.&domain. supp_trn;
        by studyid usubjid &domain.seq;
      run;

%mend COMBINESDTM;
```

When we scan this macro for dependencies, it will find `SDTM.SUPP&domain.` and `SDTM.&domain.`, which aren't meaningful without the context of their resolved values. It may be possible to emulate the macro resolution process within the dependency checker in some cases, but the sheer variety of ways of writing macros means that any implementation is unlikely to be foolproof.

It may prove more efficient to include special cases in the dependency checker to define behaviour for common macros. For example, the snippet below identifies any time when a program calls the %COMBINESDTM macro, and notes the value of its parameter as a dependency.

Example [4]: Special case within dependency checker to process %COMBINESDTM calls

```
if index(upcase(line), '%COMBINESDTM') > 0 then do;

      open  = index(line, '(');

      equal = index(line, '=');

      close = index(line, ')');


      if open > 0 and close > 0 then do;

        if equal > 0 and open < equal and equal < close then do;

          start = equal + 1;

          end   = close - equal - 1;

        end;

        else if open < close then do;

          start = open + 1;

          end   = close - open - 1;

        end;

        else do;

          call missing(temp_dep);

          start = .;

          end   = .;

        end;


        if start > 0 and end > 0 then temp_dep = strip(substr(line, start, end));

        else call missing(temp_dep);

      end;

      else call missing(temp_dep);
```

OTHER EXCLUSION CHECKS:

- Text that precedes a source keyword token should be investigated to confirm if it follows SAS syntax for outputting a dataset. Only input datasets should remain as recorded dependencies.

Example [5]: Checking for output dataset references

```
%global excl_clause tlen;

  %let excl_clause = ;

  %let delimiter = %str(.);

  %let search_term = %upcase(&source.&delimiter.);

  %let excl_list = %str(DATA CREATETABLE OUT= OUTPUT);

  compressed_line = compress(upcase(line));

  pos = index(compressed_line, "&search_term.");  %* position of e.g. SDTM. *%;

  %do _i = 1 %to %sysfunc(countw(&excl_list, %str( )));

        %let term = %scan(&excl_list, &_i, %str( ));

        %let tlen = %length(&term);

    %let excl_clause = &excl_clause and not(
         (pos>&tlen) and substr(compressed_line, max(1, pos-&tlen), &tlen) =
         "%upcase(&term)"
         );

  %end;
```

- Possible dependencies that do not follow SAS dataset naming conventions should be excluded. This means that dependencies should only be recorded if they contain numbers, letters, underscores or ampersands (for unresolved macro variables).

CROSS-CHECKING DEPENDENCIES

Once dependencies are identified for a given program, they can be cross-checked against other programs and documents to ensure consistency and strengthen quality control. For example, if dependencies are detected in a production program, the corresponding QC program can also be scanned, and the results compared. Any discrepancies between production and QC dependencies may introduce downstream issues as new data becomes available. Even if two distinct dependency datasets currently yield identical outputs, there is no guarantee that they will not diverge in the future.

Dependencies identified in an SDTM or ADaM could also be used to compare against a relevant specifications document. This would help to ensure that the specifications reflect the reality of how a program is produced. Having up-to-date specifications can then aid other programmers conducting QC as they produce their own equivalent programs.

If a QC tracker has been utilised and happens to store dependencies, then dependencies found in a program could also be compared against the QC tracker dependencies to ensure that the QC tracker is up to date. Any mismatches between the program dependencies and QC tracker dependencies could then be highlighted to study leads for further investigation.

WHY IDENTIFYING DEPENDENCIES MATTERS

The result of our search is a list of each study dataset's dependencies:

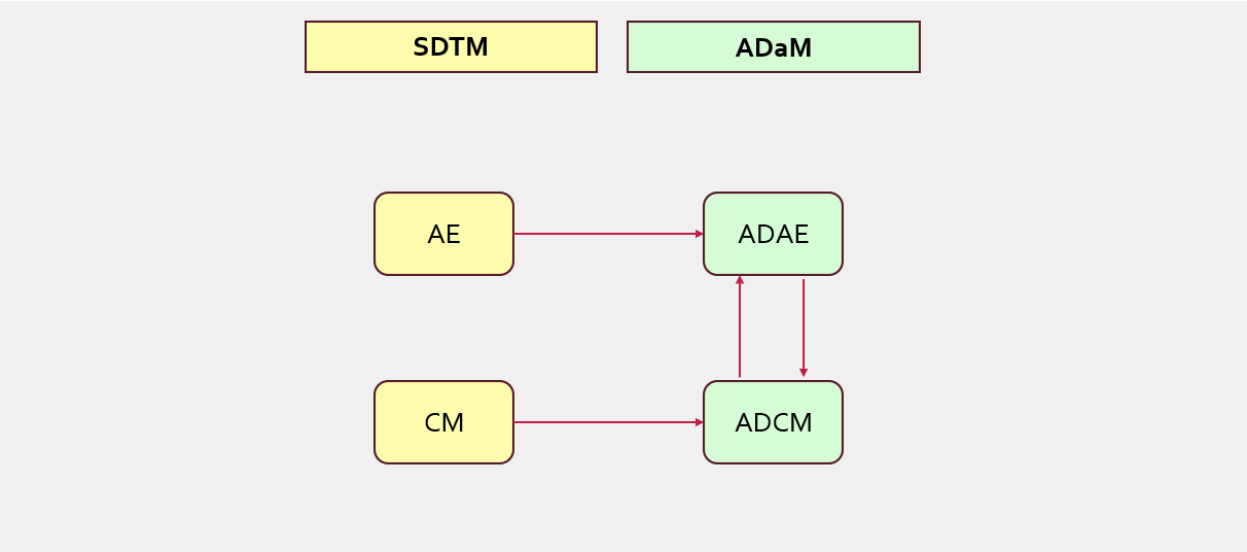
Dataset	Dependencies	
DM	DEMOG	DISPO
SE	DISPO	EX
EX	EX	SE
AE	AE	SE
ADSL	DM	EX
ADEX	ADSL	EX
ADAE	ADSL	AE
Demography	ADSL	
Exposure	ADEX	
Adverse Events	ADAE	

We can use this in several ways:

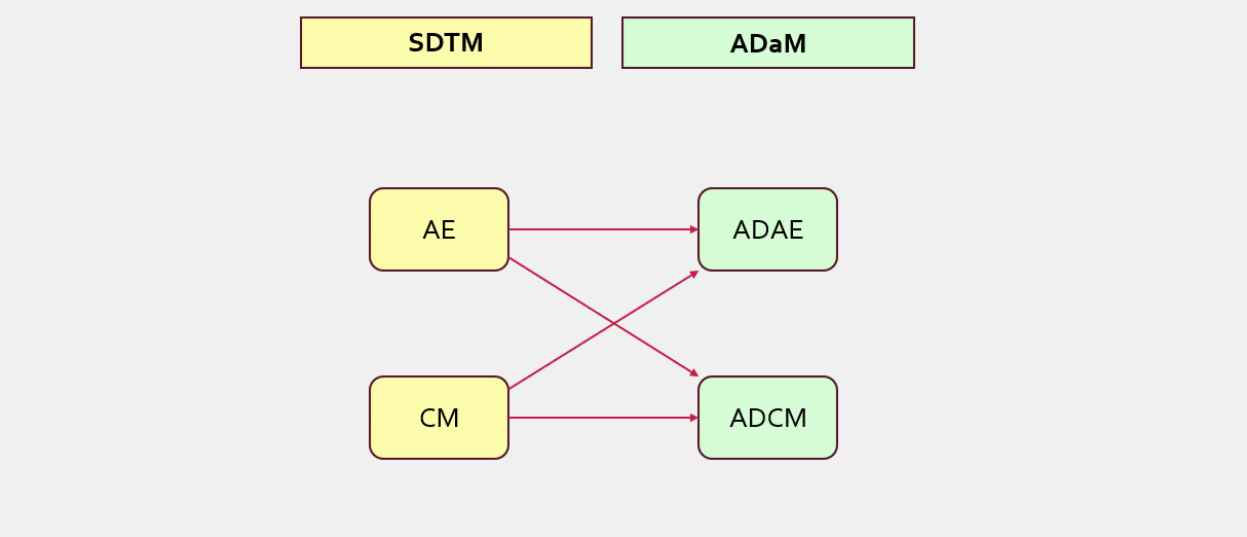
FINDING CIRCULAR DEPENDENCIES

When programs are being developed in tandem, it's possible for circular dependencies (where two datasets are both dependent on each other) to creep in without either programmer noticing. These should be avoided as they make it impossible to cleanly run through the study programs in order.

For example, at ADaM level we might have ADAE containing a "Subject took Concomitant Medication to Manage this AE" variable, and ADCM containing a "Concomitant Medication Taken due to AE" variable. If the ADAE program refers to ADCM and ADCM refers to ADAE, this creates a circular dependency:



The fix for this is straightforward – replace the ADaM source datasets with their SDTM equivalents:

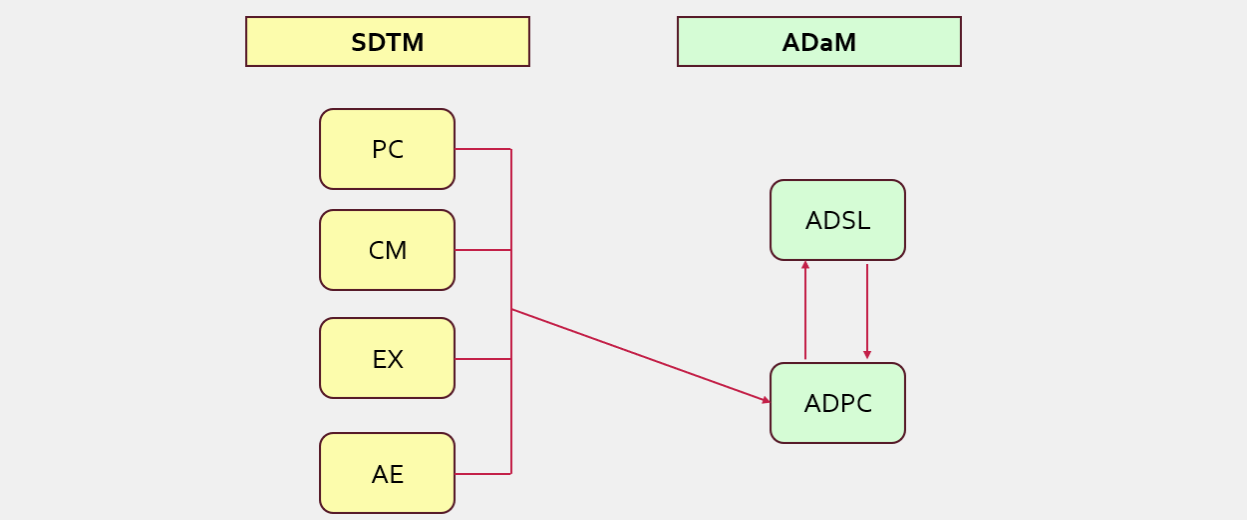


SOLVING COMPLEX CIRCULAR DEPENDENCIES

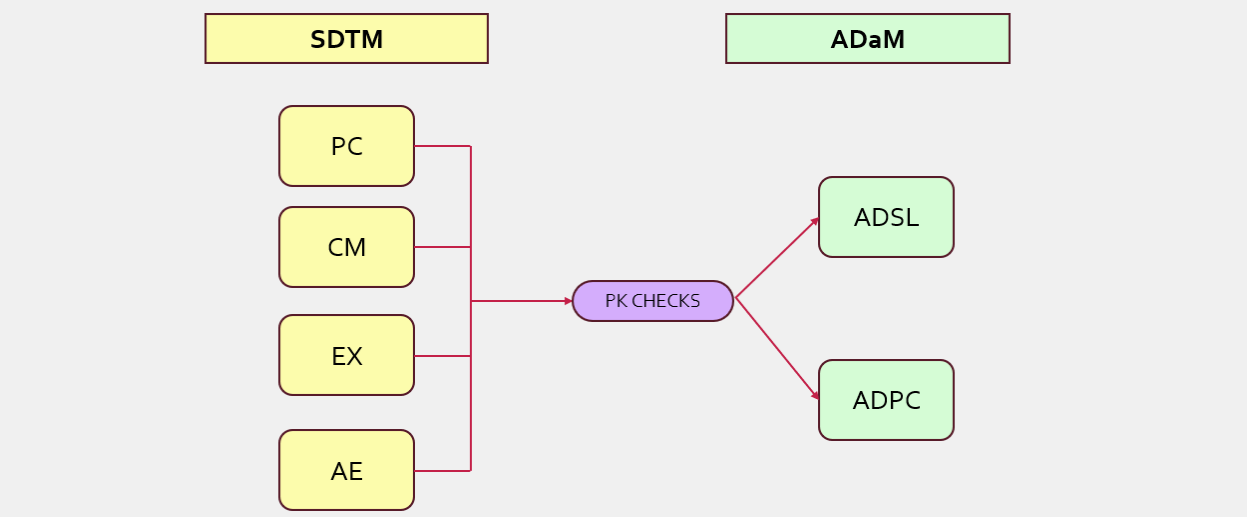
Sometimes it's not so simple to avoid circular dependencies. One previous study contained a particularly complex PK analysis:

- Each PK sample (in ADPC) was categorised as 'valid' or 'invalid' based on a variety of factors including AEs, exposure and concomitant medication
- Subjects were only included in the PK analysis set (in ADSL) if at least 50% of their samples were 'valid'

This presented a problem: ADPC was dependent on ADSL for various subject-level variables (e.g. treatment arm, study dates and populations), but ADSL was dependent on ADPC to determine the 50% cutoff for the PK analysis set.



One solution would be to copy the sample-level validity checks into both ADSL and ADPC, but this creates a risk of the two versions of the checks diverging if future updates are made. Instead, we can save the checks into a macro which gets called by both ADSL and ADPC:



IDENTIFYING UNEXPECTED DEPENDENCIES

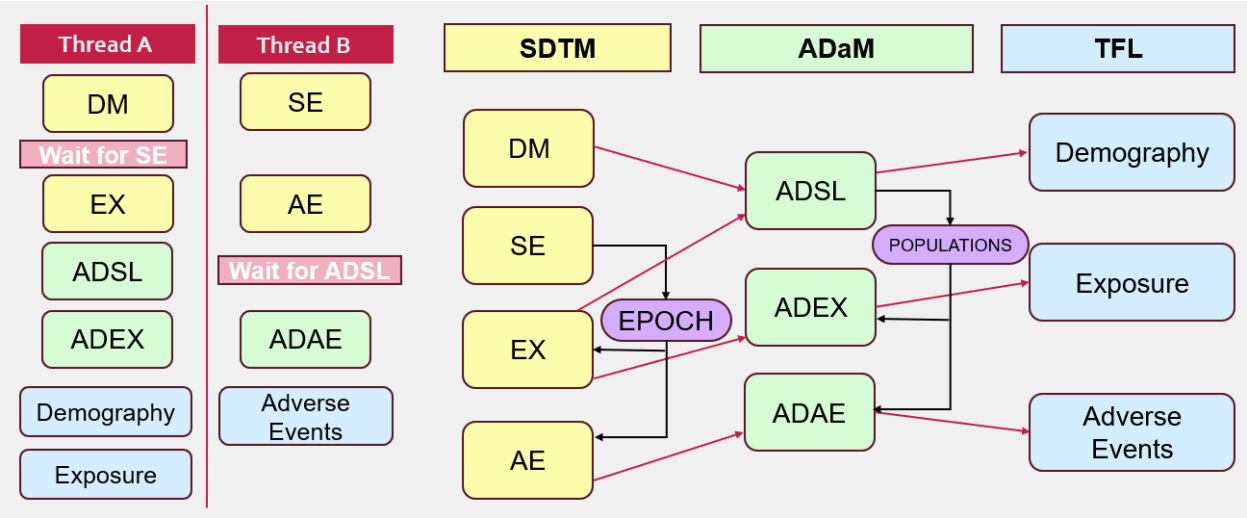
A study's programming will generally be developed by a large team, with no single person having full knowledge of what's in every program. The study leads' assumptions of dataset dependencies may differ from the reality of what's been programmed, so a programmatic check is a useful tool to identify any unexpected dependencies.

REVIEWER'S GUIDES

Dataset dependencies should be documented in the Reviewer's Guides as part of study submission. Programmatically generated dependency lists can ensure that these are easily and accurately populated.

CREATING BATCH RUN FILES

When new raw study data is received, it's advisable to run all study programs on the new data. A batch run file can be created which automatically runs each program in a specified order – a dependency map can be used to generate a batch file which runs programs in the correct order, and even run certain programs in parallel if they are independent.



An illustration of how the example study can be run in parallel across two threads, while ensuring that no programs are run out of order

AUDITING RUN TIMESTAMPS

A final study run should have all study programs batch-run with saved logs to show that all programs were run in the correct order. An audit script (armed with the dependencies list) could check through the timestamps of all programs to ensure they have been run in an appropriate order.

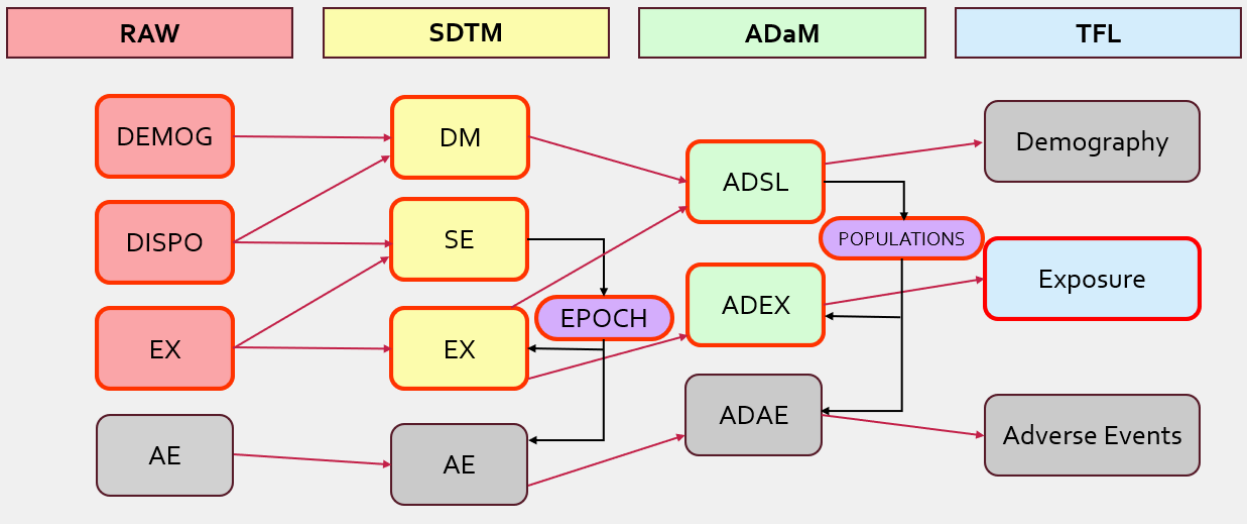
FINDING THE SOURCE OF ISSUES

When rerunning programs on new data, it's not uncommon to find issues cropping up that weren't there before. Identifying the source of the issue as fast as possible is crucial to avoid delays, but the interconnectivity of datasets means it's not always obvious what's causing the problem. Understanding all the datasets that a program is directly and indirectly dependent upon can help to trace an issue back to its root cause. This can aid study leads in managing team resourcing if issues are programming-related; otherwise, it can aid in discussions with clients regarding data issues.

CASE STUDY – SUBSET RERUN

To illustrate one application of dependency detection which inspired this paper, here is a scenario we encountered during a recent study. Following primary reporting of all study TFLs, a Safety Follow-up was required, meaning we needed to reproduce a subset of TFLs based on a later cut of the raw data.

To minimise costs, it was decided that only the necessary subset of raw data was going to be cleaned for us, so we needed to identify a list of raw datasets that were required for the subset analysis. To achieve this, we performed an iterative search – determining which ADaM datasets fed into the TFL subset, then which SDTM datasets fed into that ADaM subset, and finally which raw datasets fed into that SDTM subset.



An illustration of which datasets are required to rerun just the Exposure TFLs from the example study – required datasets/macros are outlined in red

CONCLUSION

Automated dependency checking is a hugely useful tool for properly understanding the structure of a study's datasets, ensuring that any piece of data collected can be traced through all levels of study reporting. We hope that the concepts explored in this paper will serve as inspiration on your own journey through the dependencies web.

CONTACT INFORMATION

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

Rhys McCrosson / Simon Purkess
PHASTAR
2D Bollo Lane
London W4 5LE
Email: rhys.mccrosson@phastar.com / simon.purkess@phastar.com
Web: www.phastar.com