PhUSE EU Connect 2018

Paper TT03

Crowd-Sourced ADaM-Programming and Standardization: A Design Study

Holger Langkabel, F. Hoffmann-La Roche Ltd., Basel, Switzerland

ABSTRACT

If a programmer is starting to work on the programs for ADaM-datasets, she is usually starting with writing the specifications first and then she is proceeding to transferring these specifications into programs. Sometimes, working aids like templates and standards might be available, but often they don't fit the study needs exactly so that a considerable amount of work is still necessary. How easy would life be if every variable that has ever been specified and programmed would readily be available to any subsequent study. This paper wants to propose a system framework which shall enable this. The design takes a database approach both to specification writing and code generation. An accompanying curation process shall enable collaboration across teams and facilitate a lean standardization process. Though maximum flexibility shall be retained, a strict approach to the relational properties of specification items has to be taken to achieve this.

INTRODUCTION

Honestly speaking, most of the tasks a statistical programmer is performing in everyday life aren't very innovative nor does she have to deal with a great amount of variability: Tables summarizing adverse events don't look very different across studies nor does medical history, laboratory findings, or vital signs results. Even primary efficacy endpoints do not tend to vary much within a therapeutic area – though they might be very different across therapeutic areas. As a consequence, the supporting ADaM-datasets don't vary much either. Actually, they vary even less because they also contain information included for technical and traceability purposes that doesn't change much in structure. Still, most of a programmer's time will be devoted to produce programs that process SDTM to ADaM-datasets and most of the programming statements will perform routine tasks. Working aids like program templates or sophisticated macro systems usually exist but their use, too, requires still a lot of work to adapt them to study specificities which in the case of macro systems often isn't a straightforward thing to start with.

This paper wants to present a system framework for ADaM-dataset generation that differs from usual macro-based systems by taking a database approach to both specification and coding. This shall enable a granular modularization of dataset-generating programs which in turn shall maximize re-usability of both specification and code. Thus programmers shall be enabled to focus on the development of truly new code to support the needs of a specific study while not having to completely start from scratch all over again. As a result, both specifications and programs rather become "datasets" and the specification and programming code that derives a certain variable rather becomes an observation within these datasets.

The remainder of the paper will focus on unfolding the details of the proposed system. The first section will describe the framework in general terms and give a first overview of the different components. The second section will describe the specification database in more detail and the general way in which a study-level specification can be generated from the database using a specification compiler. The third section will describe the structure of the code library, the principles and design of the code that is contained in these libraries and how dataset generating programs can automatically be generated from the code libraries. The fourth section will describe the center-piece of the framework which is a graphical user-interface which grants easy access to both the specification database and the code library. The fifth section will shortly discuss the implications for a data standards governance model and how workflows for standards curation could look like. The sixth section shall shortly discuss selected aspects of the feasibility and implementation of this framework.

Nota bene: For illustrative purposes, this paper contains examples of programming code. These examples will use SAS® as statistical programming language. But please note that the proposed system is intended to be language agnostic and that the code stored within the system could be in any statistical programming language and in any combination of different statistical programming languages. Limitations are only set by the statistical computing environment in which this system would be operated.

THE COMPONENTS

Figure 1 gives an overview of the overall system design. The main purpose is to generate two (types of) outputs: specifications and programs that are based on these specifications. The typical scope of the outputs is the analysis

for a primary CSR of a clinical trial (i.e. specifications and programs necessary to generate ADaM-datasets). These outputs are generated from two databases: one for the specifications (called “specification database” hereafter) and one for pieces of code or “code snippets” (called “code library” hereafter). Both databases have a rather complex design: There exists one single information set within each database on the global level, multiple information sets on the project level (one for each project), and multiple information sets on the study level (one for each study within each project). For most of the items within this database structure there has to be a one-to-one relationship between items in the specification database and the code library. The information that is stored within these databases is used to generate study-level specifications and programs in an automated fashion using two scripts (called “specification compiler” and “code generator” hereafter). The specification compiler solely uses the information in the specification database, while the code generator uses the information compiled in the specification to look up the corresponding code in the code library to generate the analysis programs. For convenience, a GUI will be used to coordinate the workflows within this system design.

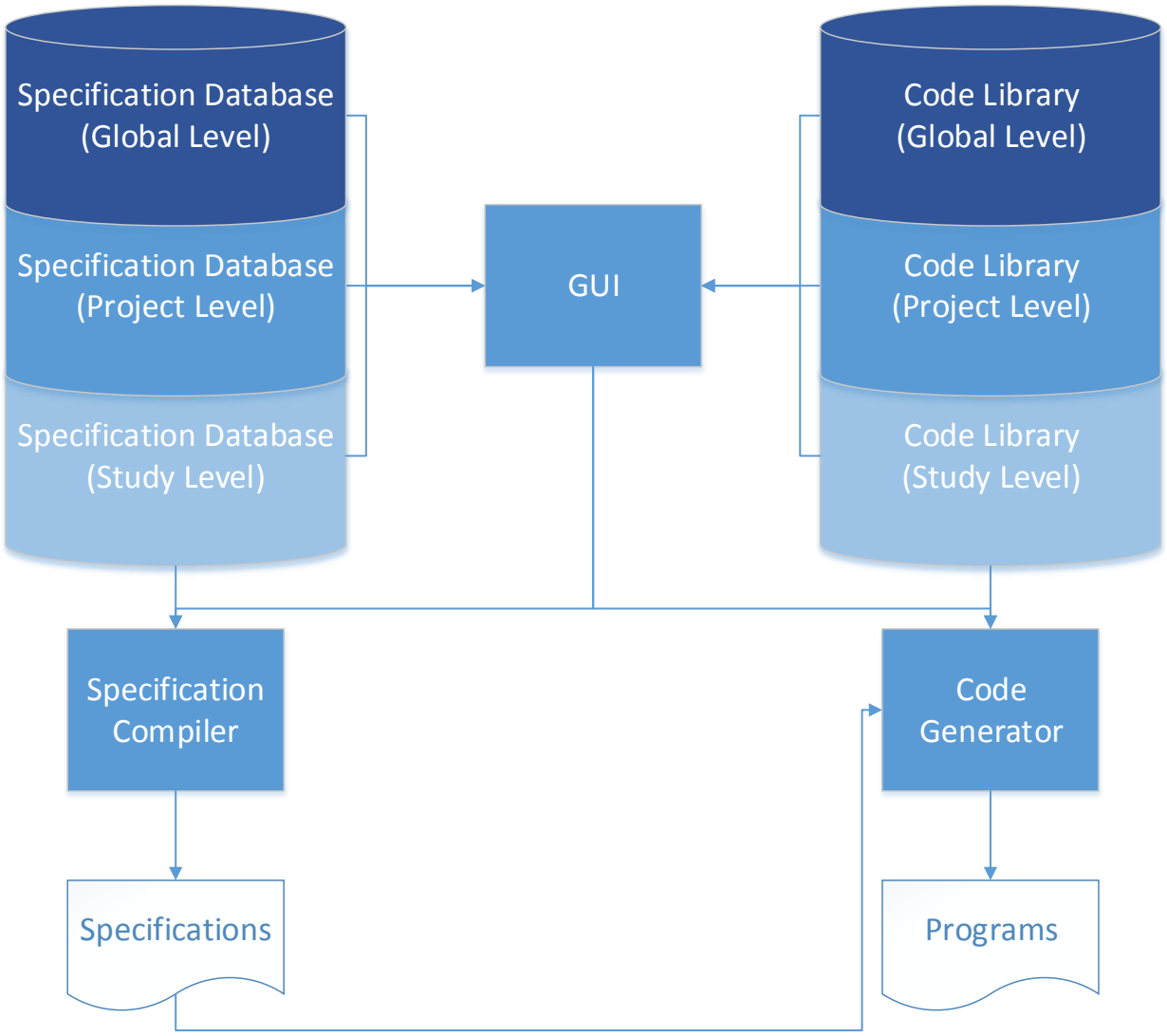

Figure 1 System components

THE SPECIFICATION DATABASE & THE SPECIFICATION COMPILER

THE SPECIFICATION DATABASE

The specification database consists of several instances: one single instance on the company-wide global level, multiple instances on the cross-study project level (one instance for each project), and multiple instances on the study level containing the genuine study-specificities (one instance for each study within each project). More levels - e.g. for therapeutic area – can be added. The instance on the global level contains all variable metadata (variable name, variable label, variable length, etc.) together with the specification for the standard derivation algorithm(s). In contrast, the instances on the project and study level cannot contain variable metadata, but hold only the information about which derivation algorithm from the higher levels has been chosen for a certain variable and – if a new

algorithm shall be defined on a level – they also contain this algorithm definition; variable metadata is always inherited from the global level.

Each of the instances can contain several derivation algorithms for the same variable. This is necessary to (1) allow different derivations for the same variable across different datasets and (2) to allow more than one option for the derivation of a variable on the global level. To allow the unique identification of derivation algorithms, a unique identifier has to be created. It is recommended to define a structured identifier. A possible structure is <variable name>_<level identifier><algorithm identifier>_<version number> where <level identifier> is G for global, P for project and S for study level, <algorithm identifier> is a two digit number (which can be further structured; e.g. 00 could indicate "direct copy from SDTM/ADSL"), and <version number> is an optional two digit running version number which allows for retirement of specific derivation algorithms. Some examples: ADT_G01_01, ADT_G02_03, ABLFL_G04_02, ABLFL_P01_01, ABLFL_S01, EGDTC_G00, SEX_G00.

Slices[1] are represented in this model as (sub-)datasets. Variables, whose metadata changes between slices, are not assigned to a dataset but to a slice within a dataset. Otherwise, the variable definitions are the same as for variables outside the slice. The slice itself has metadata associated with it (slice name, slice label, slice definition, etc.).

Figure 2 presents a simplified visualization of the database structure together with the relations of the database tables among each other as it would look like if there was only one instance on the study-level. Besides the tables for variable metadata (tblVAR) and derivation algorithms (tblVARImplementation), it also shows similar tables for datasets (tblVAD and tblVADImplementation), code lists (tblCodeListImplementation and tblCodeListItemImplementation), time windows (tblTW and tblTWItem), and project/study metadata (tblProject and tblStudy). These tables contain the elements that form the specification in the end. They are accompanied by two tables (tblAssignVADImplementation and tblAssignVARImplementation) which link the different elements together and assign – for example – a variable implementation (i.e. the set of variable metadata elements and derivation algorithm elements) to a variable within a dataset.

There are some few requirements that algorithm texts should observe to enable a redundancy-free, lean database:

(1) References to other variables should preferably be one-level references (i.e. not contain the dataset name), so that the same algorithm text can be re-used for the same variable in several datasets.
(2) References to SDTM-variables should be preferred over references to ADaM-variables containing the same information to minimize interdependencies between variable derivations.
(3) Derivations should be specified as a function of other variables and should not refer to other variable derivations unless this cannot be avoided.

THE SPECIFICATION COMPILER

The specification compiler is a script that uses the study-level specification database file as input. This is done by compiling the final specification file based on the following information:

- The study-specific derivation specification elements recorded in the study-level instance,
- The implementation assignments recorded in the study-level specification instance,
- The respective project-level derivation specification elements recorded in the relevant project-level instance,
- The respective global-level derivation specification elements recorded in the global-level instance, and
- The relevant metadata elements recorded in the global-level instance.

The compiled file can be of any format that the script supports (e.g. RTF, PDF, Define-XML).

[1] A "slice" is a subset of a dataset that typically includes a subset of the dataset rows where the variability of metadata is greater between slices than within slices. ADaM-BDS-datasets are usually sliced by parameter.

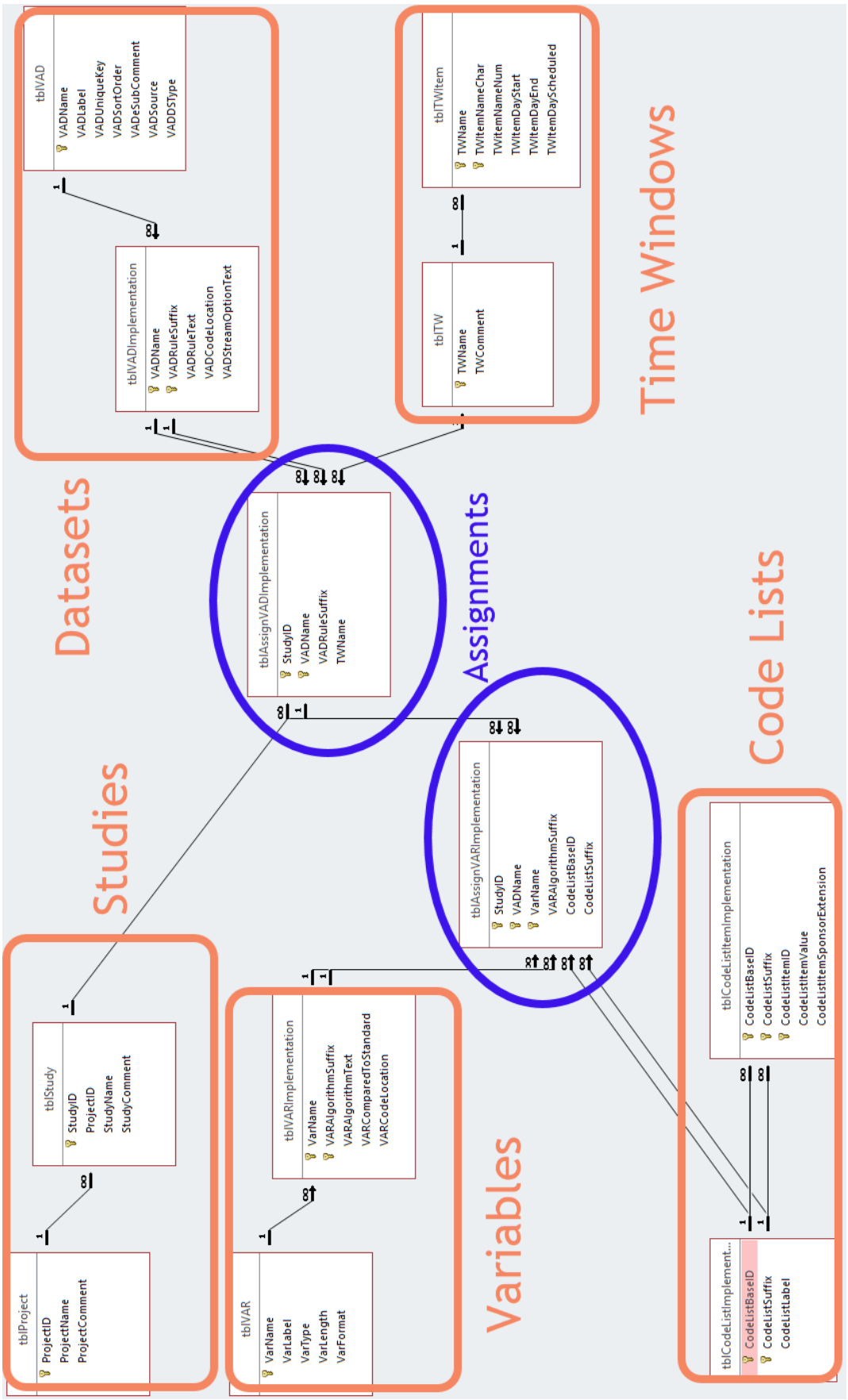

Figure 2 Complete data model of the specification database

THE CODE LIBRARY & THE CODE GENERATOR

THE CODE LIBRARY

In accordance with the structure of the specification database, the code library also consists of multiple layers: A global level, a project level, and a study level. But different to the specification database, the code library is essentially only a defined structure of directories. Each of these directories contains files with code snippets (one file per algorithm definition) associated with the algorithm specifications in the corresponding part of the specification database. To establish a link between specification and corresponding code snippet, a simple naming convention is imposed: The code snippet has to be stored in a file whose name is the unique algorithm identifier (e.g. the code for algorithm ADT_G01_01 is to be stored in a file called adt_g01_01.sas).

There are some few requirements that must be adhered to by each code snippet:

(1) It must take a dataset of a defined format as input.
(2) The output must be the input dataset augmented by one variable and one variable only and as many additional records as required by the accompanying specification. (This requirement may be subject to well-defined exceptions, e.g. DTYPE, date and time imputation flag variables, variables for time-to-event analyses. However, exceptions should be avoided whenever possible.)
(3) It must adhere to naming conventions for replacement strings for the input and output dataset as well as the merge key.

In case of slicing, the slice replaces the dataset in the above requirements. Depending on the design of the statistical computing environment in which this system is operated, there might be an additional requirement on the statistical programming language(s) being used.

Below is an example of such a code snippet file named adt_g01_01.sas:

```
/* Header */

/*CODESTART*/
data _OUTDATA_;
   set _INDATA_;
   adt=datepart(adtm);
run;
/*CODEEND*/
```

THE CODE GENERATOR

The code generator is a script with a rather simple design: It uses the specification file produced by the specification compiler as input and extracts the algorithm assignments together with the variable metadata. Then it looks up the corresponding code snippet (identified by the unique algorithm identifier) and pastes them together to the final dataset generating program.

Below is an example of how the above code snippet would be pasted into the final program generating the dataset ADXXXXXX. Together with the example for the code snippet file above it illustrates how the code snippet is processed:

```
/* Header */

data adxxxxxx01;
   attrib ...
          adt      length = 8 label = "Analysis Date"
          ...
run;

...

* ADT ----------------------------------------;

data adxxxxxx08;
   set adxxxxxx07;
   adt=datepart(adtm);
run;
```

```
...

proc sort data=adxxxxxx42 out=permanent.adxxxxxx(label="XX Analysis Dataset");
   by studyid usubjid adtm;
run;
```

The code elements between the keyword comments `/*CODESTART*/` and `/*CODEEND*/` are extracted from the file and the strings `_INDATA_` and `_OUTDATA_` are replaced with the relevant temporary dataset names. The final program consists of a header, a dataset initiation which creates an empty dataset with all the variable metadata, then a section for each specified variable which is the processed corresponding code snippet, and a final section which implements the dataset metadata.

It is important to note that the intended workflow prohibits making any changes to the generated program directly because otherwise edits will get lost if the code generator is run again and – even more importantly – any edits made to the program directly won't be added to the code library for future re-use. Instead any changes that shall be made to the resulting program must be facilitated by making the appropriate changes in the specification and by re-running specification compiler and code generator. Thus, it is also guaranteed that specification and code are always synchronized.

As can be seen from the above example, the way code snippets are stored and pasted together to form the final program results in two advantages:

- **Absence of macro code:** Because derivation options are already selected when creating the specification and code snippets are selected based on these choices, there is no necessity for macro language in the code snippets. As a result, the final program is free of macro code, too. This is in stark contrast to classical macro based ADaM-dataset generation which tries to cover alternative derivation options and scenarios by elaborate macros with parameters to select from available options and which often requires in addition to a macro call further pre-processing of the macro inputs and post-processing of the macro outputs to deliver the desired result.
- **Easily readable code:** As a direct result of the above, the final program is very well structured and easily readable: Each variable as well as variable and dataset metadata all have their own well-defined and enclosed space in the program structure. This makes it easy to locate all the code relevant for a certain variable derivation, which also facilitates an efficient code review.

As mentioned above, the code generator uses one of the static specification files, which are produced by the specification compiler, as input. The reason why the code generator uses one of these files instead of querying the specification database directly is twofold:

(1) The use of an interim output supports the modularization of the system: During implementation, the code generator can be developed without the specification database and specification compiler being fully operational since the static input file could also be created manually. At the same time, the code generator only requires a specified structure of the static specification file and not of the specification database in total which improves the maintenance of the later and also removes the interdependency of the two system components making it possible to re-design or even retire the specification component as necessary.
(2) It improves the workflow within the study team: With having a static file as intermediate, it is possible to work at the specification and at the code simultaneously. The static specification file can serve as a draft which can be used by some team members while others are still updating and finalizing the specification.

A NOTE ON QUALTIY CONTROL

Please note that in this system it is not necessary to perform quality control on the program produced by the code generator. Instead the code snippets can be made subject to quality control: Since the generated program is just a concatenation of the single-variable derivations, a reasonable amount of quality control would already be performed on this program if the code snippets were quality controlled. Performing quality control on the code snippets is a very simple task because these are essentially "parameter-less macros" and presenting a limited number of test cases with different input datasets would be enough to assess whether or not the code snippet is producing the intended output.

THE GRAPHICAL USER INTERFACE

As Figure 1 above illustrates, the graphical user interface (GUI) is the center-piece of the whole system. It is designed to grant easy access to both the specification database and the code library and shall facilitate an easy review and construction of study-level specifications. To do so, it connects the derivation algorithms and code snippets from all levels with each other and presents them on screen in a by-variable fashion.

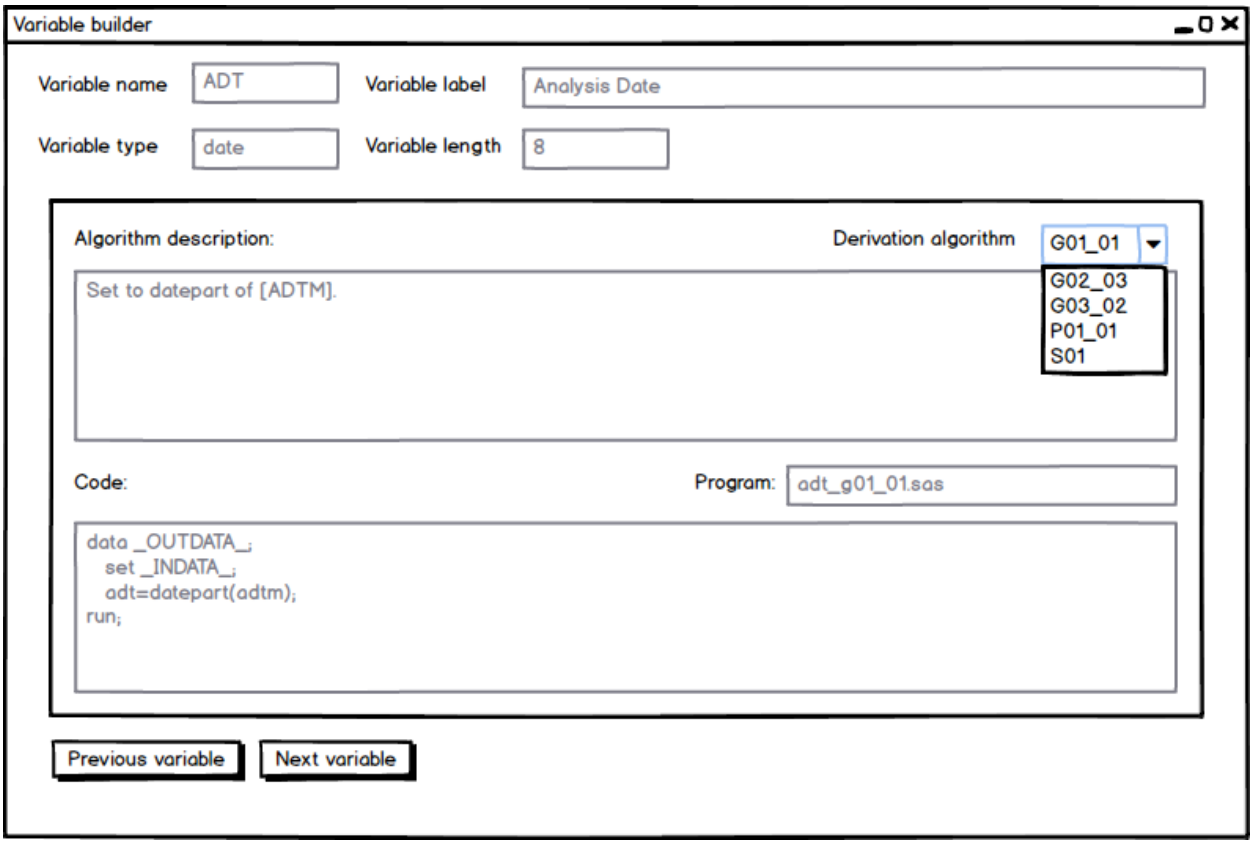

Figure 3 Variable view within the GUI selecting global-level derivation algorithm

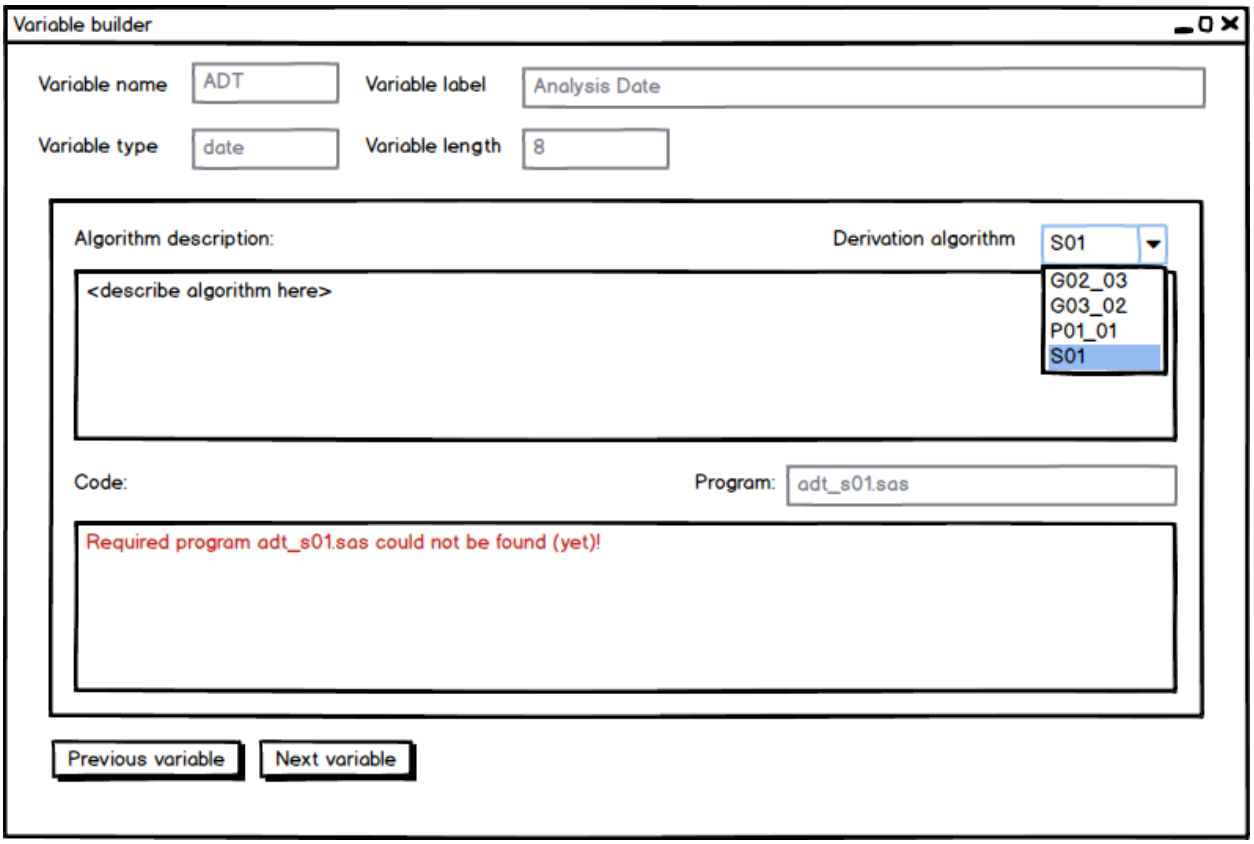

Figure 4 Variable view within the GUI selecting study-level derivation algorithm

Figure 3 shows a mock-up of the variable view within the GUI. This view displays the variable metadata at the top and below the derivation algorithm from the specification database together with the accompanying code snippet from the code library. Because the mock-up is showing an example for a derivation defined on the global level, none of the fields can be edited except for the algorithm assignment itself.

This figure also illustrates the significance of the unique identifier for the derivation algorithm. Only this identifier enables the GUI (and the code generator) to link the specification details with the corresponding code snippet and to display both together. At the same time, it can be seen how the interrelations between the different variable metadata items in the specification database illustrated in Figure 2 are used to collect and display all metadata relating to a certain variable implementation.

Figure 4 shows the same view but with the study level specification being selected. In this case, the field for the algorithm description can be edited and any edits will be saved into the specification database on the study level. At the same time the GUI is displaying a warning message because the file with the code snippet does not exist (yet) and hence no code can be displayed either. But even if the file would exist, the field showing the code snippet would stay disabled because the code snippet should not be created using the GUI. This allows for specification and code being created by different programmers simultaneously and this also allows for code development making use of the functionalities and other tools available in the statistical computing environment.

STANDARDIZATION, STANDARDS CURATION AND (CROSS-COMPANY) COLLABORATION

Because the general structure of both the specification database and the code library is the same on all the levels, this system supports streamlined and automated data standards governance. Study teams will simply pull available global and project standards from the repositories and add their own study specific derivations at the study level. Once they have completed their task they can simply file a “pull request” to the standards curation team which will perform a quality control on the specification texts and the code snippets and – if passed – promote both to the desired level.

Unlike usual macro-based programming approaches, this system can easily support a great variety of competing standards and standards governance could simply be reduced to letting the user community determine which the most useful derivation is rather than trying to define it upfront. If a mandatory file storage convention for the final static specification file of a study is put in place, it is possible to search these files in an automated manner and create usage statistics for individual derivations. In some sense, specification and code can become data itself within the suggested system framework, which also allows for analyses with other purposes (e.g. quality assurance/quality control).

At the same time, non-standard study-specific derivations can easily be copied to other studies. Due to the highly modularized programming approach, a study team can copy over the study-level specification database or import the final specification file and copy the code library. Then the study team can keep, delete or change variable implementations as necessary with the code being updated automatically.

Additional to company-wide standardization, this system could easily be extended to an industry-wide collaboration platform by making the specification dataset structure and the code snippet requirements public. Specification compiler, code generator, and GUI could also be shared publicly in an open-source project.

CONSIDERATIONS ON IMPLEMENTATION AND MAINTENANCE

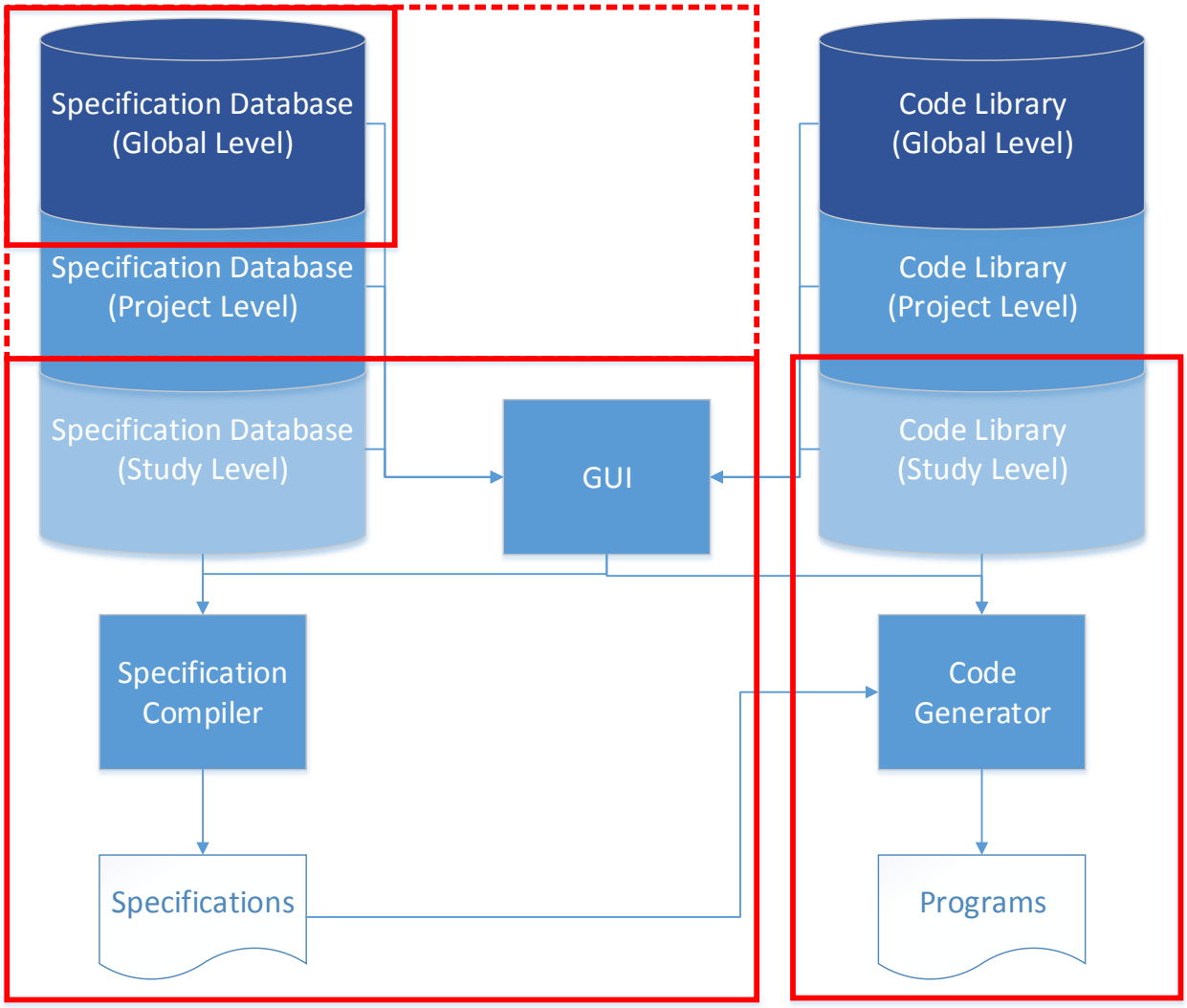

Figure 5 Current status of proof-of-concept activities

Though this paper presents a design study, several of the parts have already undergone a proof of concept (framed by solid lines in Figure 5 if completed, framed with dashed lines if currently ongoing). The lessons from these activities have informed this paper.

An intriguing property of the proposed system is its modularity which allows for staggered, scalable implementation: The specification database concept is independent from the programming strategy and can therefore be implemented independently from both commitment to and existence of the code library concept. At the same time, the code generator uses a static specification file as input and works therefore independent of the way the specification was created. The GUI is just a convenience in the first place and its development can be postponed as resources allow and preferences require. But also specification compiler and code generator are independent from specification database and code library respectively. Instead the latter two can simply be created as classical template collections which can be used together with copy-paste techniques to create the respective outputs (though this is obviously a very laborious task).

Another interesting property of the proposed system is its leanness: Please note that the system itself only consists of two scripts (specification compiler and code generator) and the GUI. These are the only parts that require programming resources. The rest of the system comes down to information management: The specification database is nothing more than data tables in a relational database and the code library is essentially only a file storage system. The contents of the specification database and code library wouldn't be provided by system developers, but instead be crowd-sourced from the users who populate the study-level instances during their usual

programming activities. The project-level and global-level instances would then continually grow through the curation processes.

CONCLUSION

Though many details have still to be worked out along the way of implementation, this paper hopefully provides a good idea of the general design of the proposed system and shows the great potential of efficiency gains that this system is intended to facilitate.

Future steps for the research into this design will include extending the features that have undergone proof-of-concept (cf. Figure 5) to include the capabilities supporting project and global level activities and to link existing tools for the specification database and code library. Hopefully, the results can be shared publicly in the future.

ACKNOWLEDGMENTS

Some of the ideas in this paper draw from the presentations ‘AD05: Efficiency and Delivery Driven by Metadata’ by Magnus Mengelbier at PhUSE Annual Conference 2016 and ‘SI06: ADaM mapping - key considerations for a metadata driven realization’ by Elena Glathe at PhUSE Annual Conference 2017.

Special thanks go to Urs Naef who developed the basic concept of the specification database as well as all the details that come with the implementation. He also kindly provided Figure 2.

CONTACT INFORMATION

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

Holger Langkabel
F. Hoffmann-La Roche
663/3.107
4070 Basel, Switzerland
Email: holger.langkabel@roche.com