Paper AD17 – PHUSE EU Connect 2023

Code-Generation: Turning metadata into SAS, R, or Python

Stuart Malcolm, Veramed, London, UK

ABSTRACT

End-to-end automation requires a robust way of converting metadata into executable code.
This whitepaper describes an approach that can be used to generate programs in any language from metadata using open-source technologies.
The generated programs can be incrementally generated which means that manual edits are preserved during the development lifecycle which allows the programs to be re-generated should the metadata change.

INTRODUCTION

This paper describes the user and functional requirements for a code generator designed for use in the implementation of end-to-end automation solutions for GxP compliant clinical trial data projects.

USER REQUIREMENTS

The user requirements can be understood using the two main scenarios for code generation

- "creator" generators, which are typically used once or infrequently during the clinical trial project lifecycle (e.g. scaffolding, project templating, etc.), and
- "pipeline" generators, which are used frequently during project development lifecycle (e.g. SDTM code-generation, TFL Automation, etc.).

Figure 1 below provides a context for the user requirements – the generator consumes metadata and outputs program source code.

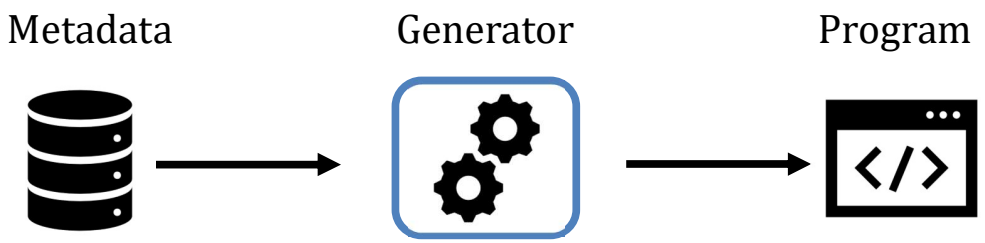

Figure 1 Metadata-driven code generation context diagram

HIGH-LEVEL REQUIREMENTS

1. Ingest machine-readable metadata in any format: JSON, YAML, SQL, etc.
2. Output executable programs in any language: SAS, R, Python, Shell, etc.
3. Generate individual programs (e.g. an individual TFL)
4. Generate multiple programs at the same time (e.g. all SDTM programs, etc.)
5. All input and output files can be versioned (e.g. using Git)
6. Ability to debug code-generation
7. Logging of what was generated, including any issues resulting from 'dirty metadata'

USE CASE ANALYSIS

There are three main actors that interact with the code-generator:

- **Metadata Producers** create the metadata that is used to drive the code-generation. MP's need to understand the standards and metamodels being used in the metadata and are responsible for ensuring the metadata is valid. Note that MP's may need to enrich the metadata as requested by the Automation Programmer.
- **Automation Programmers** create the code generation capability (typically writing code generation templates). AP's need to understand the target programming language and runtime as well as the structure of the source metadata. AP are responsible for ensuring the generator produces valid code. AP's may need to enrich the metadata to support efficient code-generation
- **Trial Programmers** consume the generated programming and run on the target runtime (e.g. A GxP compliant SCE) using clinical trial data. The TP is responsible for ensuring the valid execution of the programming. TP's may need to add project-specific code to the generated code (e.g. pre-processing to address code-quality issues, etc.)

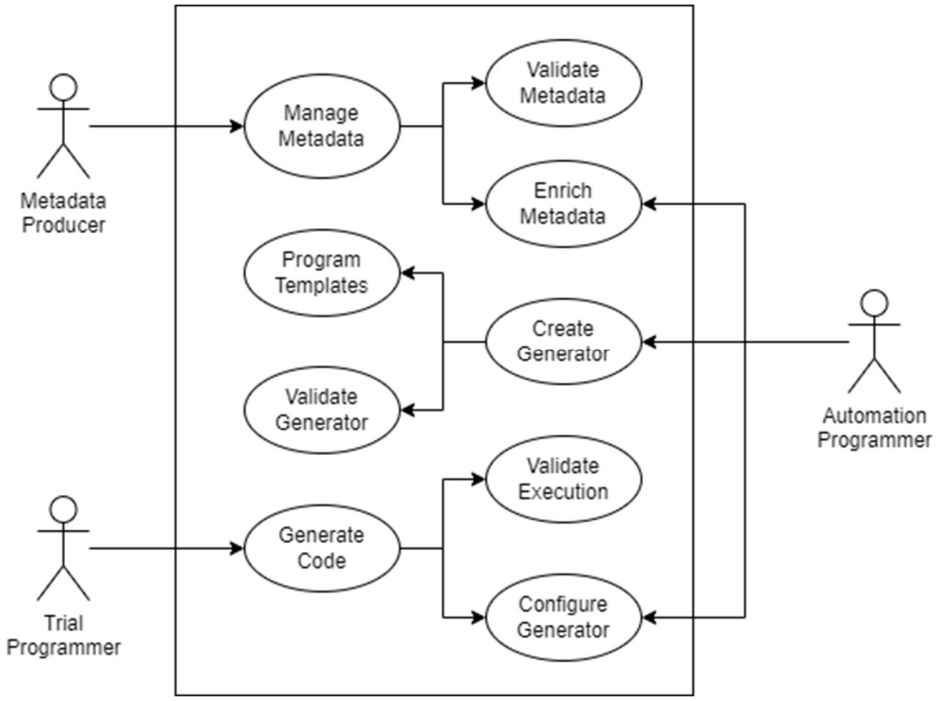

Figure 2 Code-Generation Use-Case diagram

Figure 2 describes the full use-case for code-generation. This document we will focus on the Automation and Clinical Trial Programmer use-cases. To help elaborate the use-case diagram, two key user scenarios are explored – one which describes a "creator" generator, and a second for a "pipeline" generator.

USER SCENARIO 1: CREATOR GENERATOR (SDTM)

In this scenario, the Automation Programmer develops an SDTM project setup automation which the Trial Programmer uses to initialize a new SDTM conversion project. During the initial development period the Trial Programmers adds study-specific programming to implement SDTM.SV (Subject Visit), however during this period a protocol amendment means that additional family medical history needs to be collected. The Trial programmers re-runs the project setup automation to generate additional SDTM.APMH (Associated Persons Medical History) domains.

To walk through this scenario, it has been split into two:

- 1a. Where the Automation Programmer creates an SDTM Project Setup automation and the Trial Programmer uses it to create an SDTM project, then starts to edit the SV program.
- 1b. Where the Trial Programmer updates the configuration to include an additional domain (ADMH) and then re-generates the SDTM project. This results in the SV program retaining the edits from step 1a and a new program generated for the ADMH domain.

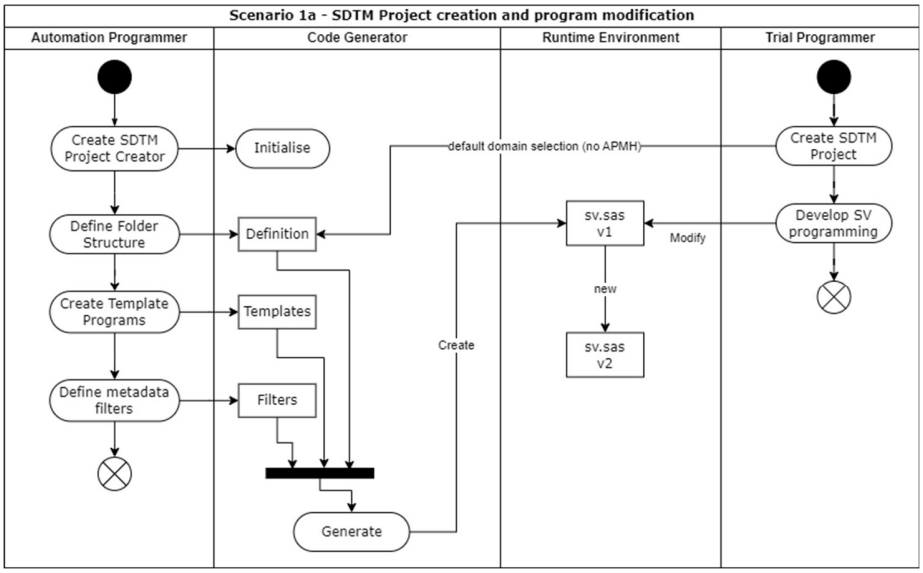

Figure 3 Activity diagram of User Scenario 1a (Create project then manual edit programs)

As can be seen from Figure 3 above, once the Automation Programmer has created the *SDTM Project Setup* generator their involvement is complete, and the Trial Programmer can use the generator to create a new trial-specific SDTM project by configuring the definition (e.g. by selecting which domains should be created). In this example, the Trial Programmer configures the generator to create only the SV domain program (ok – not realistic but makes the point!). Once the SDTM project is created, the Trial Programmer can start work on the generated `sv.sas` program. Which leads on to scenario 1b, where the Trial programmer regenerates the project and continues working.

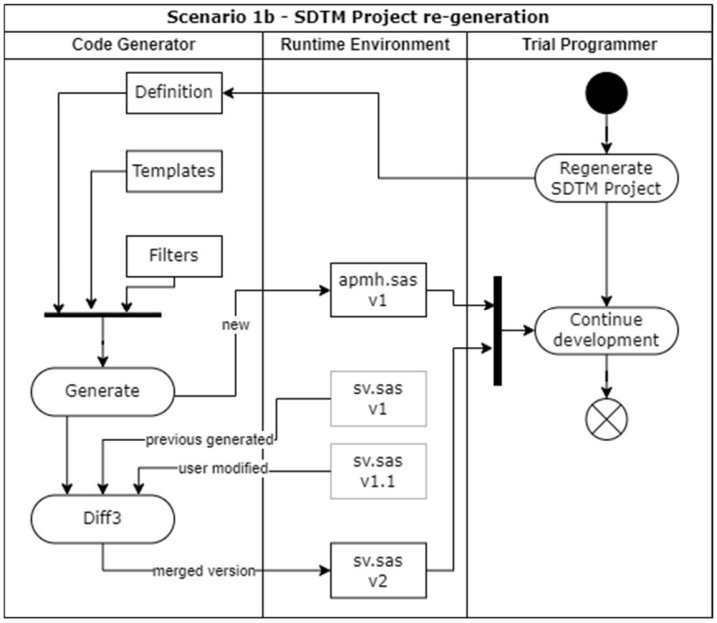

Figure 4 Activity diagram of user scenario 1b (Project is regenerated)

USER SCENARIO 2: PIPELINE GENERATOR (TFL)

In this scenario, the Automation Programmer develops a TFL automation which the Trial Programmer uses to generate all the standard safety reporting. As the efficacy shells are reviewed and approved, the Trial programmer enriches the metadata with the new shells and regenerates the TFL programming for QC.

The activity diagram for this scenario is essentially the same as Scenario 1b above – where the Trial Programmer starts with a project which is populated with safety TFL programs, and then adds efficacy outputs to the definition incrementally, and regenerates the project each time while also running development and QC activities in parallel.

FUNCTIONAL REQUIREMENTS

Figure 5 below identifies key functional components of a code generator. This diagram represents groups of functionality and relationships between them – it is not a design requirement.

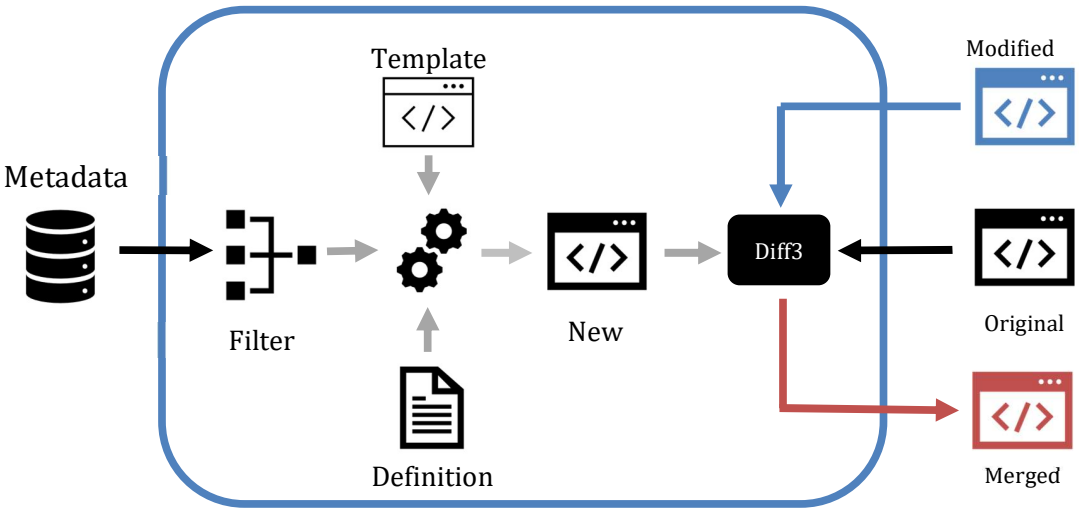

Figure 5 Functional components of code-generator

DEFINITION

The definition is a machine-readable description of the operations that the code-generator performs to translate the metadata into code. Typically this will be implemented as one or more configuration file(s) – e.g. json, yaml, etc. or a DSL (Domain Specific Language) e.g. Cue, or embedded in a general purpose programming language.

Key functional requirements of the definition include:

1. The target folder structure to be generated
2. The contents of the target folder structure
3. Contents of target folders can consist of:
    a. Individual generated programs
    b. Groups of generated programs
    c. Static files (e.g. data files, etc.)
4. Definition of filters that are available for generation in each output folder
5. Templates to use for generating each output program
6. Code-generation control parameters
    a. Name of generator
    b. Generator version
    c. Templates to use for generation
    d. Metadata to be included in every program generation
    e. Whether Diff3 is applied or not
    f. File globs to watch and trigger regeneration when changed
    g. Destination location where generated code is output
7. Generator-specific configuration parameters (e.g. for an SDTM generator this may be a list of domains to generate, etc.)

TEMPLATES

Templates are files used to generate output programs. They are text files that contain markup that is used to control how metadata is inserted into the output file.

1. Templates files can contain a complete template or make use of partial template snippets
2. Partial templates can be reused across all templates
3. Templates are stored in a directory structure that reflects the output directory structure
4. Templates use a consistent markup style – for example `{{ variable }}` or `{% command %}` etc.
5. Template markup supports:
    a. metadata inserted directly into the output file
    b. metadata filtered and formatted before being inserted into the output (e.g. uppercase, etc.)
    c. indentation is preserved in the output fie and can be controlled within markup
    d. Looping through metadata
    e. Conditional statements to control output
    f. Passing variables to partial templates

FILTER

Filters are patterns that define how to generate code. Filters are analogous to web routes where the path into the output folder structure is equivalent to the web url.

1. Filters are defined by
    a. A path pattern into the output folder structure
    b. The controller/operation to use to generate the code
    c. The Template(s) to use for this filter
    d. The unique name of the filter (so that it can be reused)
2. Filter path patterns are paths into the output folder and can contain context markup variables, for example
    a. `/this/is/a/path`
    b. `/this/is/a/path/to/program.py`
    c. `/path/to/{prod|qc}/runall.sas`
    d. `/{program}.r`
3. Code generation operations are controllers (as per the MVC design pattern)
    a. Operations take as input a model (i.e. Filter) and View (i.e. Template)
    b. Operations are defined in the Generator functionality
    c. Operations output code to the destination location (see Definition)

DIFF3

Diff3 merging allows the Trial Programmer to edit generated code and then regenerate the code while preserving manual edits. The intention of this functionality it to allow the Automation Programmer to build-in “edit here” sections in the generated code to allow the Trial Programmer to e.g. write pre/post-processing code, or other study-specific manually generated code.

1. To facilitate Diff3 merging, the Automation Programmer should ensure that Trial Programmers don’t have to write code at the ‘edge’ of templates – there should be stable code on both sides. So, for example, add comment blocks where users should add code:

```
/** Pre-Processing code start **/

* Add Your User Code Here;

/** Pre-Processing code end **/
```

2. The Diff3 functionality takes as input three files.
    a. The newly generated code file
    b. The previously generated code file (without user modifications)
    c. The previously generated code file (with user modifications)
3. The Diff3 functionality creates as output a single file that contains the merged input files.
4. If the input files cannot be merged, then a conflict occurs
5. Merge conflicts are output with using delimiters to indicate the conflict
6. The text used to delimit merge conflicts are configurable by the user (see Definition)

GENERATOR

The generator is the controller at the heart of the code-generator. It marshals the metadata and feeds it to the appropriate templates to generate the output code.

1. Multiple generators are exported and available to be used by filters.
2. Typical generators are:
    a. Individual program generator (e.g. for `/path/to/{program}.r`)
    b. Group of programs generator (e.g. for `/path/to/progs`)
3. A generator takes as input a filter.
    a. Resolves all the input metadata.
    b. Adds global definitions to the metadata.
    c. Defines aggregate metadata context for the filter (i.e. merges all the metadata context – defined along with the templates - for the filter path)
4. A generator applies the resolved metadata to the template(s) defined in the filter
5. The resulting output program is written to the output location (see Definition) with the filter path applied
6. The generator maintains a log of all inputs, outputs and results

CONTACT INFORMATION

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

Stuart Malcolm
Veramed Limited
Email: stuart.malcolm@veramed.co.uk
Linkedin: https://www.linkedin.com/in/stuart-malcolm/

Brand and product names are trademarks of their respective companies.