PharmaSUG China 2021 - Paper DS-007

A Customizable Conformance Check Tool Design

Dan Li, Merck;
Haiping Yu, dMed Biopharmaceutical Co., Ltd.

ABSTRACT

Clinical data conformance check is to verify whether certain datasets satisfy requirements of a ruleset. This paper introduces a design idea and tool to unify conformance checks into a validation framework. Another piece of the puzzle of this validation framework design is introduction to database design in terms of metadata version control.

INTRODUCTION

Conformance checks can be divided into two categories. The first category is a general check for data points of a certain ruleset - for example, a conformance check for SDTM datasets. The second type is a check on interconnected complex data - for example, SDTM or ADaM metadata for define-XML generation purposes. the key difference between these two types is that the conditions for the second type of check are dynamically generated from the data and some other conditions. For example, fact A matches the rule and infers fact B; fact B and the rule infers fact C. If fact A is detected, it is necessary to check that both facts B and C exist.

The second type of check is difficult to write with if-else statements. Even for the first type of check, the naive implementation is too slow to meet the practical needs due to its speed.

In this paper, we briefly describe the design idea of naive and review its drawbacks, based on which we propose using rule engine to solve the dilemma of the second type of consistency checking. At the same time, the introduction of the rule engine also solves the problem of the speed of the first type of problem. The introduction of the rule engine extends the application scope of compliance checking. Also considering the ease of use, this paper proposes using a graph database to store global CDISC metadata and user-defined conformance rule metadata. After that, this paper briefly describes the necessity of version control for metadata and the database design ideas for implementing metadata version control.

1 THE ORIGINAL NAIVE DESIGN

The basic idea of the naive design is to apply each check rule to all data points, and then another check rule to all data points until all check rules have been traversed. This process can be illustrated by pseudo-code as below

```
for rule in rules:
    for data_point in data_points:
        check(rule, data_point)
```

where function call check(rule, data_point) means checking whether the current data point passes the check with a conformance check rule.

1.1 WORKFLOW

As shown in Figure 1, in line with the workflow, the core of the code design is also in two parts: the rule (and the validation logic implementation) and the domain instance. This process can be summarized in the workflow shown in this diagram: the user marks the path where the ruleset and the research dataset are stored, and then calls the entry function launch(). The function then wraps each rule in the ruleset into a series of Rule objects, which are then distributed to the corresponding domain objects via the dispatch() function. Each domain object contains its corresponding clinical trial dataset. These domain objects then

call their own validate() function, which returns a report of the results corresponding to a check rule. This includes the check rule id, how many data violate the rule, and a detailed description of the violation.

This design separates the check rules from clinical datasets in the above process from the overall workflow, so users can add custom check rules to customize the checks.

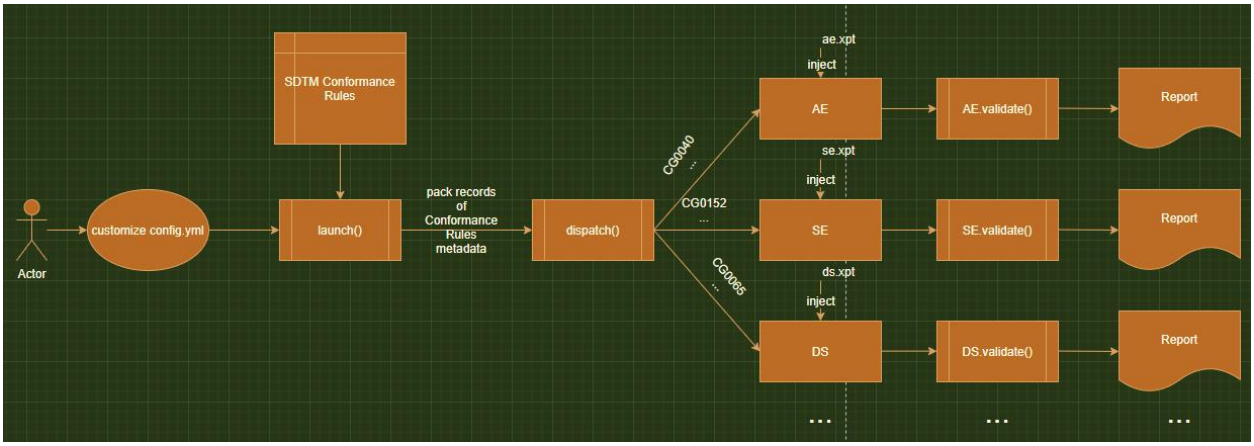

Figure 1 Naive Workflow

1.2 CLASS DESIGN

As shown in Figure 2, consistent with above workflow, the core of the code design is also in two parts: the rule (and the validation logic implementation) and the domain instance.

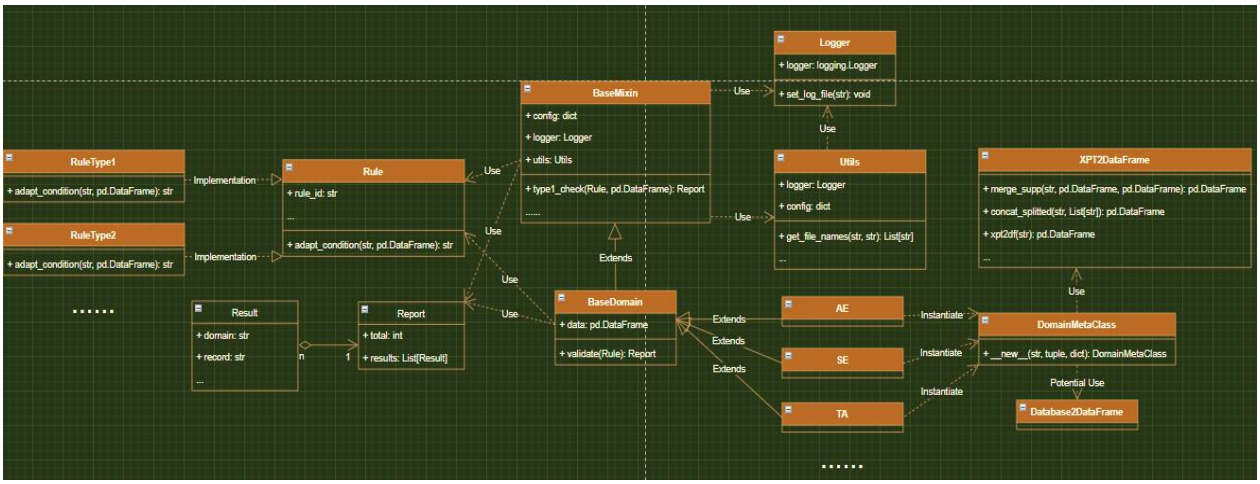

Figure 2 Naive Class Design

1.3 VALIDATE PROCESS

Firstly, I want to introduce the combining of ‘Condition’ and ‘Rule’ columns. SDTM conformance rule CG0041is shown below.

Rule ID	SDTM IG Version	Rule Version	Class	Domain	Variable	Rule	Condition
CG0041	3.2	1	EVT	AE	AESER	AESER=’Y’	AESCAN = 'Y' or AESCONG = 'Y' or AESDISAB = 'Y' or AESDTH = 'Y' or AESHOSP = 'Y' or AESLIFE = 'Y' or AESOD = 'Y' or AESMIE =

Rule ID	SDTM IG Version	Rule Version	Class	Domain	Variable	Rule	Condition
							'Y'

Table 1. SDTM Conformance Rule CG0041

The logic of this rule is to first select which domain datasets to be selected based on SDTM IG Version, Class and Domain columns. In this example, the dataset to be checked is AE.

For some check rules, the range of datasets to be checked is described by NOT or ALL, and the full range of domains should be provided by SDTM IG Version, as in the following example.

Rule ID	SDTM IG Version	Rule Version	Class	Domain	Variable	Rule	Condition
CG0086	3.2	1	EVT, INT	NOT(AE, DS, DV, EX)	--OCCUR	--OCCUR^=null	--PRESP = 'Y' and --STAT = null and --OCCUR is present in dataset

Table 2. SDTM Conformance Rule CG0086

Let's focus on CG0041. Next, a subset of AE is obtained by applying the condition specified in the 'Condition' column. Each observation in this subset is then checked against column 'Rule'. The observations that do not meet the conditions specified in column 'Rule' value will be printed in the alert message.

A review of this process will show that 'Condition' and 'Rule' columns can be combined as

```
Condition and not(Rule)
```

That is, by taking the inverse of column 'Rule' condition and then taking the intersection with column 'Condition' condition, the observations that violate rule CG0041 can be found.

Secondly, I want to introduce the concept of *atomic conditions*. Combining 'Condition' and 'Rule' column as mentioned above, a composite condition should look like

```
(AESCAN = 'Y' or AESCONG = 'Y' or AESDISAB = 'Y' or AESDTH = 'Y' or AESHOSP
= 'Y' or AESLIFE = 'Y' or AESOD = 'Y' or AESMIE = 'Y') and (not AESER = 'Y')
```

There is only one type of condition that composites this expression, namely

```
<variable_name> <operator> <value>
```

The entire compound expression is made up of this basic component, which is combined by logical connectives (or, and, not, etc.) and parentheses. This indivisible basic component is called the "atomic condition". In addition to the above form, atomic conditions like

```
--SEQ is a unique number per USUBJID per domain, or a unique number per
POOLID per domain, including when the domain is split into multiple files
```

are difficult to express straightforward.

In the naive design, all atomic conditions are counted and summarized, and the logic of complex atomic conditions is implemented in code as the following function:

```
def atomic_type(*args, **kargs) -> pandas.Series
```

where *args and **kargs are arbitrary parameters, this function should have a uniform return type, i.e. a mask used to filter the dataset. In pandas, the data type of this mask is a *pandas.Series* with values of type bool. Then replace the corresponding complex description in the metadata with the function name and corresponding parameters, you can rewrite the Excel metadata into a machine-readable rule metadata.

Lastly, using *ast* (abstract syntax tree) to parse composite conditions to get a tree structure provides desired evaluation order of single atomic conditions, so that when visiting each node of this tree, evaluation of atomic condition on dataset can be performed. The composite condition of SDTM

conformance rule CG0041 will be parsed into a syntax tree as shown in Figure 3, whose visiting order is denoted as 1 – 9.

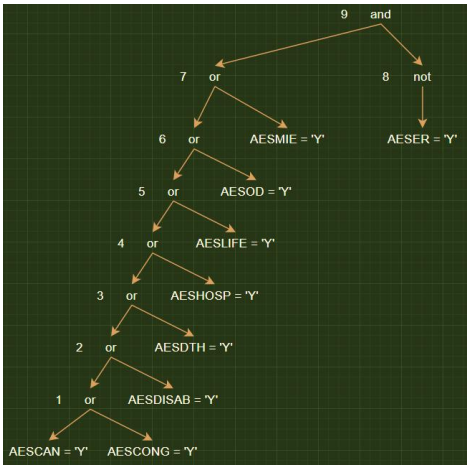

Figure 3 Visiting order of syntax tree

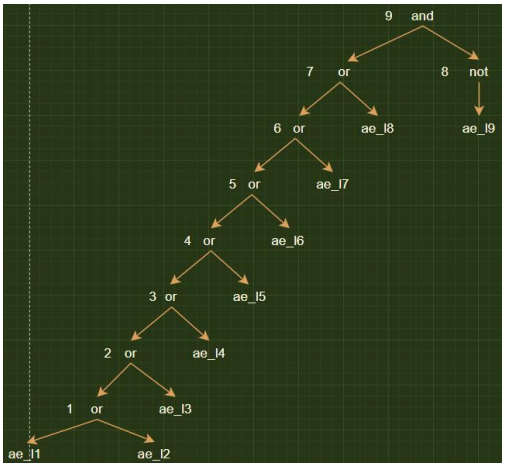

Figure 4 Visiting order with initial evaluation

After each leaf node condition evaluated on AE, a tree shown in Figure 4 is obtained. *ae_l1 – ae_l9* represents datasets evaluated from AE using leaf node conditions. Then the union operation represented by *ae_l1 or ae_l2* is executed. The result dataset *ae_inter1* replaced the *or* operation denoted 1, as show in Figure 5.

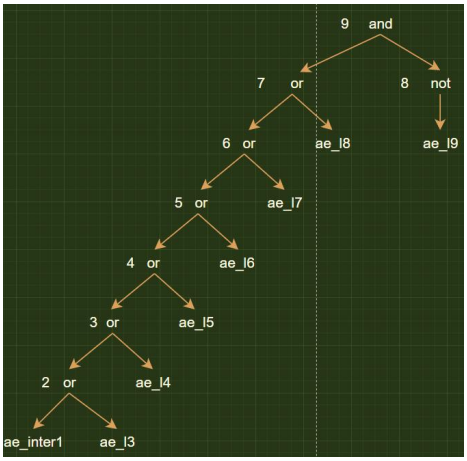

Figure 5 The first operation evaluated

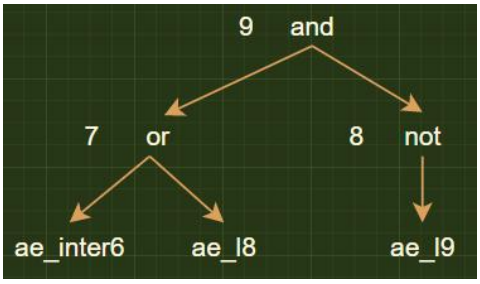

Figure 6 All 'or' operations evaluated

Next, *ae_inter1* and *ae_l3* perform a merge operation labeled 2 to get *ae_inter2*, and so on until *ae_inter6* and *ae_l8* perform a merge operation labeled 7 to get *ae_inter7*, as shown in Figure 6. *ae_l9* performs an inverse operation labeled 8 to get *ae_inter8*.

ae_inter7 and *ae_inter8* finally perform a take intersection operation labeled 9 to obtain the final filtered subset *ae_f*, as show in Figure 7.

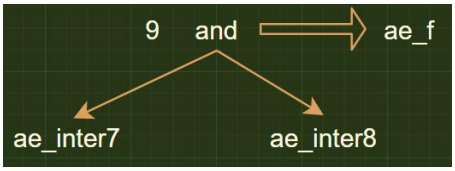

Figure 7 The last evaluation

If the result dataset *ae_f* is not empty, it means that there are records that violate rule CG0041, and information about these records (e.g. USUBJID) needs to be printed to the alert message.

Note that the above procedure does not check whether these variables exist in AE, this procedure only serves as an illustration of how the composite condition is calculated on dataset AE. A real rule check should of course add the existence check of certain variables.

1.4 DISADVANTAGES OF NAIVE DESIGN

1.4.1 TOO SLOW TO USE IN PRACTICE

Suppose a rule set has 400 rules and a data set has 1000 data points. Each rule needs to be checked once on each data point so the total number of checks is 400x1000=400000.

Each check needs to re-evaluate whether the data point satisfies the conditions of the current rule. Assuming that one check takes only 0.001 seconds, it still takes 400 seconds to perform the whole check, which is nearly 7 minutes. And 400 rules multiplied by 1000 data is a rather conservative estimate. Different countries' drug supervisory agencies and different companies may have their own inspection rule sets, and the actual application of inspection rules is much more than 400. And the number of data points that need to pass validation in a real clinical trial is much more than 1000.

Looking at the pseudo-code in section 1, we can see that the time complexity of the naive implementation is m*n, with m denoting the number of rules and n denoting the number of data points. In other words, the check execution time will increase dramatically with the number of rules and the number of data points to be checked, which is clearly infeasible.

1.4.2 UNABLE TO COPE WITH INSPECTION OF COMPONENTS WITH COMPLEX STRUCTURES

Whether applying a certain rule (e.g. CG0041) for datasets conformance check isdeterministic, but in another case, whether checking rules need to be executed may depend on real-time data and other checking rules.

For example, rules described below

```
exist(fact-a) -> assert(fact-b)
exist(fact-b) and exist(fact-d) -> assert(fact-c)
```

can be simply noted as below

```
a -> b -> c
     d /
```

where *a*, *b*, *c* and *d* are all atomic conditions. Checking whether data assert *b* to be true needs to be built on top of data asserting *a* to be true. Same with *b* and *d* infer *c*. On the other side, if data not asserting *d* to be true, then it is certain that data won't assert c to be true; if we already know *a* does not hold on the data, there's no need to check *b* or *c*.

A common example is to check whether the constraints within the SDTM (or ADaM) metadata for the purpose of generating define-XML files are self-consistent. Usually, in-house SDTM or ADaM metadata is an Excel file containing multiple sheets. Such metadata files need to satisfy certain constraints within the sheets, between different sheets, and even between multiple metadata files. Metadata files that satisfying such constraints is a necessary condition of correct define-XML generation.

However, due to the complexity of these constraints, study programmers often overlooks some constraints when filling them out, making the generated define-XML incorrect. It is also oftentimes difficult for internal define-XML generation tools to indicate the root cause of the error, which makes it time-consuming and frustrating.

If these constraints are written as rules and saved in a library, and a metadata check is run after study programmers filling in the metadata, then the violation of the constraints will trigger an alert. This would reduce a lot of time and trouble in finding metadata errors in turn from define-XML errors.

For example, in a common SDTM metadata, the following facts exist, let's name the following fact as

Fact-A:

```
SUPPAE dataset exists in “Dataset Metadata” sheet
```

Fact-B:

```
“WhereClauses” sheet has at least one record whose Dataset Name == “SUPPAE”
```

Fact-C:

```
“WhereClauses” sheet records having Dataset Name == “SUPPAE” and Controlled
Terminology == “AERELA” have Check Value == “AERELPR”
```

Fact-D:

```
The list unique values of QVAL for records with QNAM=“AERELPR” in SAS data
set SUPPAE are ['RELATED', 'MULTIPLE', 'POSSIBLY RELATED']
```

Fact-E:

```
In Controlled_Terminology metadata, records whose CODELIST_SHORT_NAME ==
“AERELA”, their unique Submission Value should contain ['RELATED',
'MULTIPLE', 'POSSIBLY RELATED']
```

The relations that the above 5 facts should satisfy are shown in Figure 8.

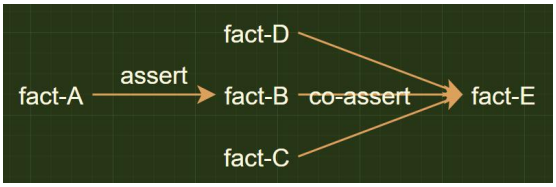

Figure 8 Facts’ relations

If fact-A is found to exist in the metadata, then fact-B should be checked to exist, otherwise the metadata is not compliant; if fact-B, fact-C and fact-D all exist, then fact-E should be checked to exist, otherwise the metadata is not compliant. At the same time, if fact-E does not exist, then it means that fact-A does not exist, and other checks based on fact-A need not be done. The feature of removing the upstream fact from the fact base if the downstream fact in the derivation chain does not exist is called truth maintenance. using only if-else statements to do conformance check of SDTM metadata does not provide truth maintenance feature, and it is easy to imagine that these nested if- else statements will become difficult to understand and maintain over time.

2 IMPROVEMENTS TO THE NAIVE DESIGN

A review of the naive design in terms of the first type check shows that the bottle neck of running slow lies in the inability to reuse the observations obtained by conditionally filtering the dataset. Even if the conditions of two check rules are identical, the naive implementation needs to recalculate the filtering process. Moreover, if observations obtained by the same filtering condition are reused, the granularity should be further fine-tuned - i.e., observations obtained by the same *atomic condition* filtered on the same dataset should be reused. In this way, if a merge condition contains the same atomic conditions for the same data set, then its filtering results can be directly involved in the logical operation of merging or intersecting. This results in a significant saving of computational time.

Charles Forgy proposed the Rete algorithm in his Ph.D. thesis *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*. The Rete algorithm can be a good solution to the time consumption problem caused by repeated computations.

2.1 RETE ALGORITHM

The problem to which the Rete algorithm applies consists of two elements, *Facts* and *Rules*. *Facts* are the data and the new data generated based on the data and *Rules*. *Rules* have following form:

```
TypeA(condition1)
```

```
TypeB(condition2)
=>
action
```

That is, when the data satisfies the conditions, the actions will be triggered. The conditions above the arrow are called the left side of the rule, and the actions below the arrow are called the right side of the rule.

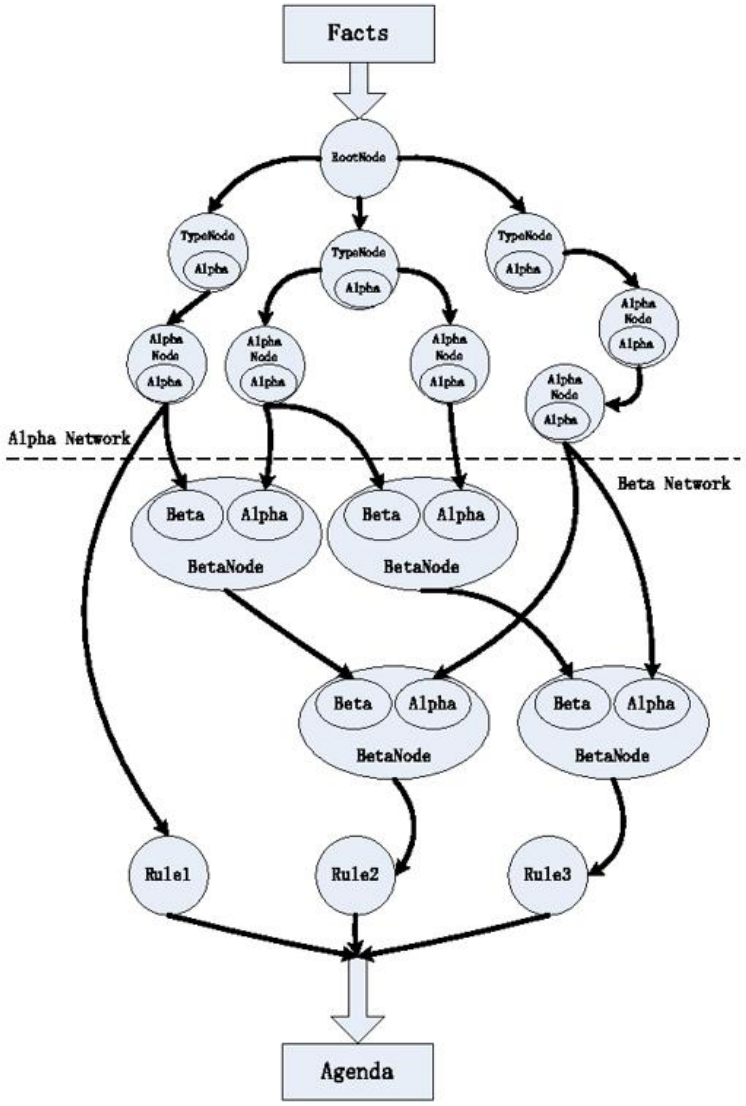

Figure 9 Rete network

The figure above represents a Rete network generated based on a certain ruleset. After the Facts flow into the Rete network and pass through the root node, the different types of facts are transferred to different TypeNodes and saved to the Alpha memory by type matching. The facts in each TypeNode are next passed to the AlphaNodes, which are similar to the atomic conditions and are relational expressions of attributes and values, such as

```
A.predicate == 'lives'
```

When a fact is passed to an AlphaNode it is stored in its memory and passed to its children if it passes the conditions of the current AlphaNode.

The Rete network is divided into alpha and beta networks. alpha network nodes have only one input. beta network nodes include beta nodes and rule nodes. beta nodes have two inputs and are used for join operations. The result of its join operation is passed to its child beta node and used as the left input. Until it reaches the rule node, all the facts that are passed to the rule node are the facts that should trigger the corresponding rule operation.

The superiority of Rete algorithm over naive implementation is that each path from the root node to the leaf node (rule node) defines a complete "left side" of the rule. The data satisfying these conditions are stored in different nodes along the path. When you need to reuse any condition of this path, you just need to fetch the data in the memory of the corresponding node, no need to recalculate it, thus time is saved.

In general, using the Rete algorithm increases the speed by several orders of magnitude compared to the naive implementation. The disadvantage is that it requires more memory than the naive implementation. However, modern rule engines implemented based on the Rete algorithm usually use an improved Rete algorithm that reduces memory use.

2.2 EXAMPLE OF RULE CG0041

Still take CG0041 in SDTM Conformance Rule as an example to see how Rete's algorithm is applied.

In fact, this rule expresses two conditions and one action. condition1 is the range of the dataset to be checked is AE, condition2 is

```
(exist(AE, AESCAN) and AESCAN = 'Y' or
exist(AE, AESCONG) and AESCONG = 'Y' or
exist(AE, AESDISAB) and AESDISAB = 'Y' or
exist(AE, AESDTH) and AESDTH = 'Y' or
exist(AE, AESHOSP) and AESHOSP = 'Y' or
exist(AE, AESLIFE) and AESLIFE = 'Y' or
exist(AE, AESOD) and AESOD = 'Y' or
exist(AE, AESMIE) and AESMIE = 'Y') and
not (exist(AE, AESER) and AESER = 'Y')
```

As shown in the figure below, assume that a record of AE (represented by the red blob at the top) enters the root node of the Rete network. Each node below the root node contains a storage area inside, indicated by another color. The record that passes the current node's condition is passed to its child nodes. This record passes the type node AE and is passed to the exist(AE, AESCAN) and exist(AE, AESCONG) nodes. In the case of the exist(AE, AESCAN) node, if the record meets this condition, it will continue to be passed to the AESCAN == 'Y' node, and so on.

If there are relational operators, such as or, and, not, etc., the beta node appears, takes over the output of the two alpha nodes to perform the required relational operations, and outputs the result of a single relational operation to the input of the next relational operation. Finally, the record of this AE successfully flows into the rule CG0041 node, and as the filtered data that needs to be alerted, the action message() is triggered to print the corresponding alert message.

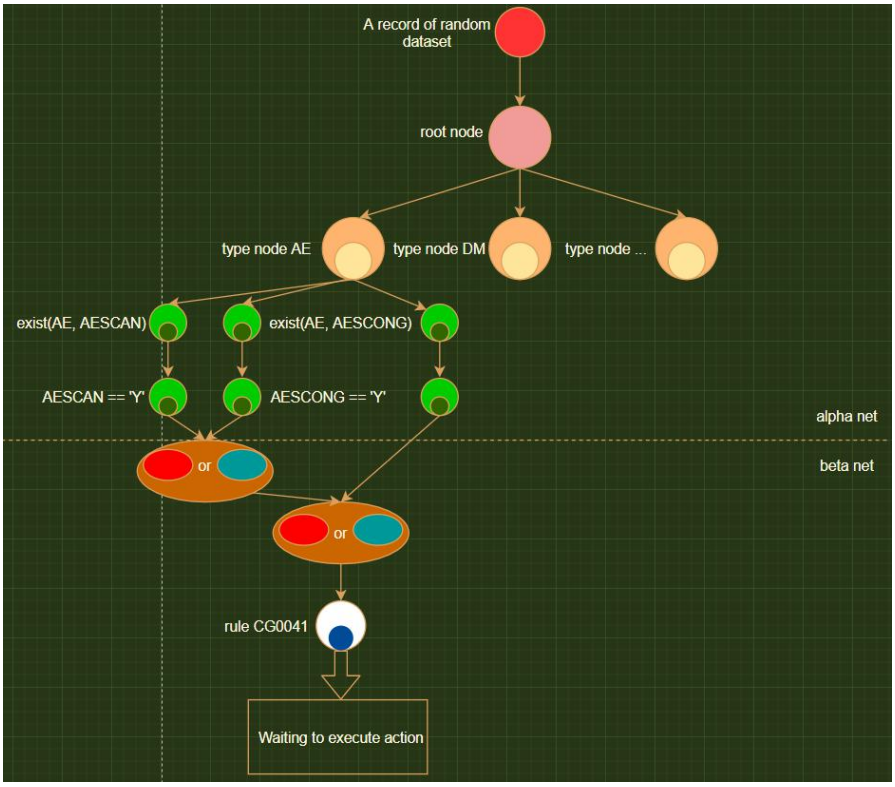

Figure 10 Rete network for rule CG0041

2.3 RULE ENGINE

A rule engine is a tool based on Rete's algorithm. The most tried and tested rule engine is CLIPS, developed by NASA's Johnson Space Center, a rule engine written in C but with a Lisp-looking rule-writing syntax. Several space projects have been successfully implemented with the help of CLIPS. clipspy and pyclips are two python wrappers for CLIPS that allow you to write CLIPS rules and interact with data in a python environment.

experta and durable_rules are also better known python rule engines.

It is reasonable to quickly implement a small proof of concept prototype with, for example, experta, and then use clipspy to implement more formal rule writing, since the reliability of CLIPS has been tested rigorously.

2.4 DESIGN AFTER CONSIDERING RULE ENGINE

As shown in the figure below, similar to the naive design, the user first needs to specify where to get the rules set through the configuration file; if it is a local file, the storage path needs to be specified; if the rules data is obtained through the API, the URL ofthe API and the user name and password needed to connect to the database need to be specified.

Different rules engines use different conventions for parsing the rule text, so the SDTM conformance rules here need to conform to the syntax rules of the rules engine used.

The launch() function turns the ruleset into a Rete network, and the data from the dataset flows into this network to produce a list with an order. Each element of the list contains two parts, one is all the observations that match a rule, and the other is the action triggered by these observations, i.e. the message() function, which prints out the corresponding alert message for data that violates the current rule.

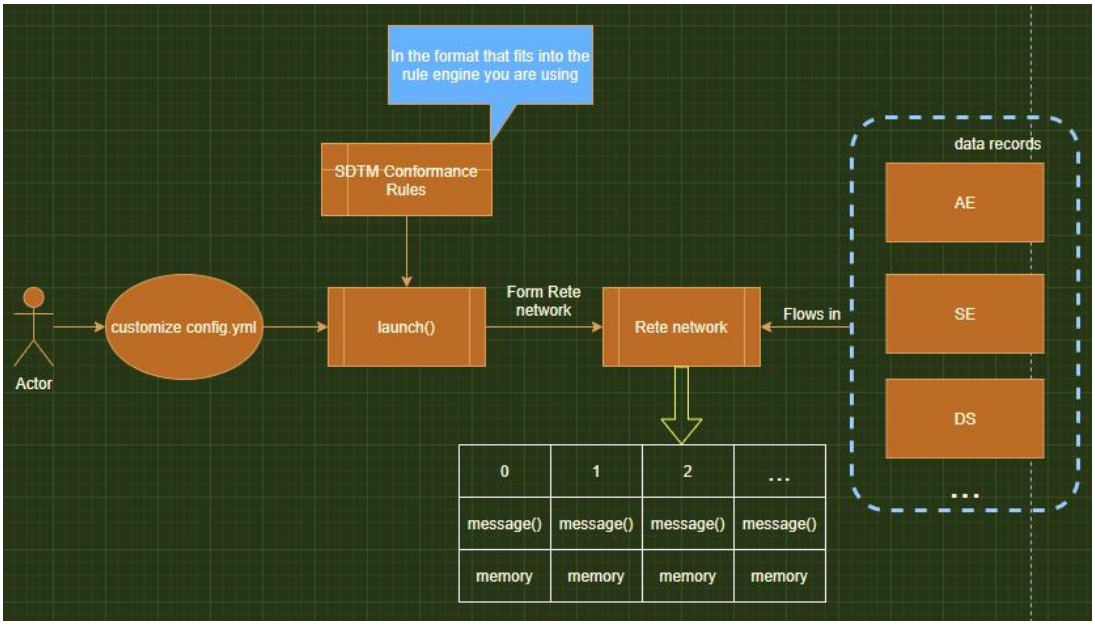

Figure 11 Workflow considering rule engine

And the code design is greatly simplified by the use of rule engine, as shown in the following figure.

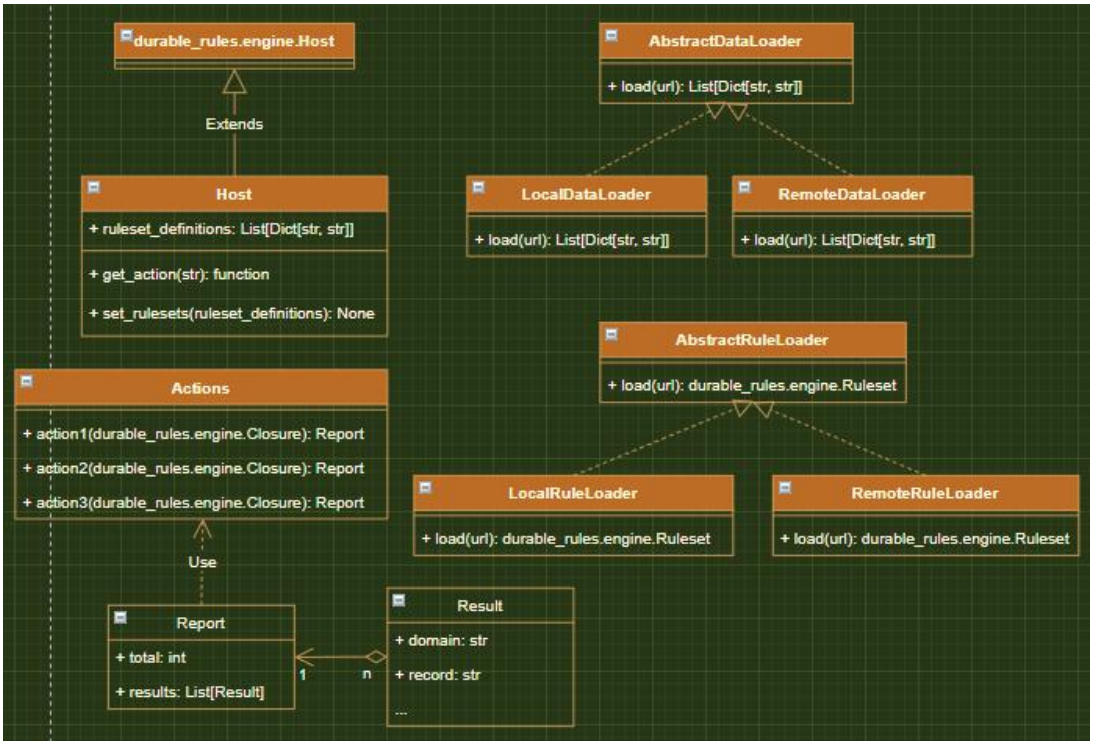

Figure 12 Classes considering rule engine

The rule example shown below uses a python rule engine called *durable_rules*, which allows rules to be defined using JSON. An example of a JSON rule definition is as follows:

```
{
  "animal": {
      "r_0": {
                "run": "frog",
                "all": [
                      {
                        "first": {
                                    "$and": [
                                      {"predicate": "eats"},
                                      {"object": "flies"}]
                                  }
                      },
                      {
                        "m_1": {
                                    "$and": [
                                      {"predicate": "lives"},
                                      {"object": "water"},
                                      {"subject": {"first": "subject"}}]
                                }
                       }]
              }
             }
}
```

It is equivalent to the following rule defined using python code:

```
with ruleset('animal'):
    @when_all(c.first << (m.predicate == 'eats') & (m.object == 'flies'),
              (m.predicate == 'lives') & (m.object == 'water') & (m.subject
== c.first.subject))
    def frog(c):
        c.assert_fact({'subject': c.first.subject, 'predicate': 'is',
'object': 'frog'})
```

'animal' in the above JSON rule is the name of the rule set, 'r_0' means this is the first rule, 'all' needs to meet all the conditions of this rule corresponding to the action--frog function will be triggered, and accordingly there is 'any', which means that any of the conditions in the rule meet an action will be triggered.' first' is used as a referent, and the '$' symbol is the operator for logical operations. These syntaxes only apply to the rule engine durable_rules, different rule engines have different syntaxes for writing rules.

This rule describes a rule in a rule set called animal. Two conditions must be met to trigger this rule: Condition 1 is that the value of the predicate attribute of the data is equal to 'eats' and the value of its object attribute is equal to 'flies'. Condition 2 is that the value of the predicate attribute of the data is equal to 'lives' and the value of the object attribute is equal to 'water' and the value of the subject attribute is equal to the value of the subject attribute in condition 1.

When the data satisfies conditions 1 and 2, the execution of the frog function is triggered and the data {'subject': c.first.subject, 'predicate': 'is', 'object': 'frog'} is inserted into memory as the new data inferred from the data.

In a more concise form, this rule could be written in the following format:

```
condition1: (m.predicate == 'eats') & (m.object == 'flies')
condition2: (m.predicate == 'lives') & (m.object == 'water') & (m.subject
== c.first.subject))
==>
action: frog()
```

If facts as below are put into above rule

```
[
  {"subject": "Kermit", "predicate": "eats", "object": "flies"},
  {"subject": "Kermit", "predicate": "lives", "object": "water"}
]
```

A new fact will be inferred:

```
{"subject": "Kermit", "predicate": "is", "object": "frog"}
```

New facts can be inferred from facts, and this inferred fact can be used as the basis for other rule inferences. This feature is ideal for solving the second type of checking problem mentioned in this paper.

The integration of the rule engine into the naive design not only solves the slow speed problem of the naive design for the first type of checking, but also extends the scope of compliance checking, making the second kind of checking easier to solve as well.

From the above example, we can draw out several key points of conformance rule metadata design and code design after considering the rule engine. First, all conditions in the rule set need to be examined comprehensively and categorized into atomic types.

For these atomic types of conditions, you may need to implement the corresponding function, such as the following:

```
--DECOD and --PTCD have a one-to-one relationship
```

User needs to define a function to return the mask of records that satisfies one-to-one relationship for variables –DECOD and –PTCD:

```
def is_one_to_one(dataset, var1, var2) -> pandas.Series
```

After a thorough examination of the conformance rule set, the atomic conditions that require customized function are recorded and implemented, such as *is_one_to_one*. Above one-to-one atomic condition can then be converted into:

```
{
  "func": "is_one_to_one",
  "params": {
    "var1": "--DECOD",
    "var2": "--PTCD"
  }
}
```

The rewritten rules can then be read into the rule engine and transformed into a Rete network. Depending on the specific implementation of the selected rule engine, the code for parsing JSON rules may need to be override, e.g. *durable_rules* does not support functions with parameters, the above *is_one_to_one* function has 2 parameters, which requires modifications to the original implementation of *durable_rules*. The specific design of the rule set should take into account the specific implementation of the chosen rule engine.

Based on the above description, the steps that the user needs to customize the check rule are

- Define a new atomic condition
- Customize the functions needed to evaluate this atomic condition
- Describe the rule according to the chosen rule engine, following the syntax of that rule engine

The rule engine can then execute the new check rule.

3 GLOBAL METADATA AND RULE SET SAVING, REQUESTING, AND VERSION CONTROL

3.1 SINGLE SOURCE OF TRUTH (SSOT) PRINCIPLE

SSOT principle means that in an information system, data elements are only saved and changed in the same place, and all data-based operations and judgments should refer to this single source of data.

For example, 2 SDTM_Conformance_rules.xlsx files, although they were both downloaded from CDISC website, were saved on two different computers and modified into different versions. Performing a conformance check using these two different versions will result in errors that are difficult to trace.

The application of SSOT principle of this design is to request SDTM IG metadata to a neo4j database via CDISC Library API and save the modified conformance rule metatadata to the same neo4j database as well.

3.2 CDISC LIBRARY API AND YOUR NEO4J DATABASE

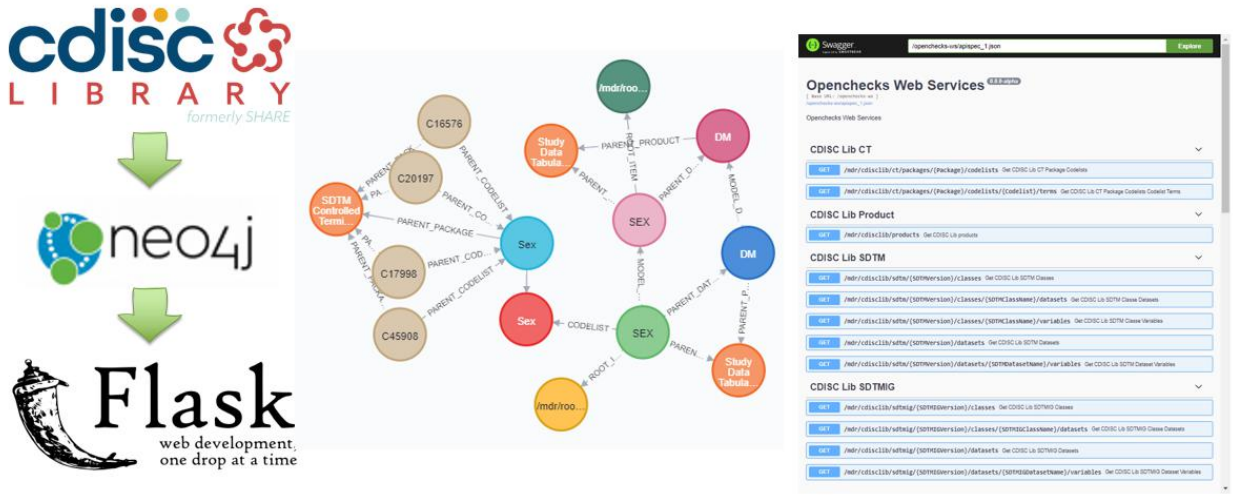

Figure 13 API technologies

CDISC Library is a neo4j database and API service provided by CDISC. CDISC saves all its metatadata into a neo4j database and makes it available through the API to users who would otherwise need to download an Excel file to obtain the metadata.

Use of the CDISC Library API requires account authorization. For CDISC members, non-CDISC members and open-source software developers have different authorization methods, please refer to CDISC Library API for details.

Anyone who wants to apply this conformance check tool design could use a remote neo4j database to store global metadata requested through CDISC Library APIs. And he/she could put conformance rules metadata into this database and request by APIs created according to customized needs.

3.3 METADATA VERSION CONTROL

3.3.1 WHY VERSION CONTROL

The design described in this paper is dedicated to a scenario where users can customize the check rules, so the conformance rule set is actually divided into two parts: the basic check rules provided by CDISC, and the custom check rules added by users.

The use of the first part of the rule set is mandatory, while the use of the second part may vary depending on the situation. For example, the use of custom check rules may be different for different studies and different milestones of the same study. In order to provide sufficient flexibility and reduce data redundancy, it is necessary to design the structure of the neo4j database where the rules are stored so that the user has flexible access to the required granularity (all conformance rules corresponding to the study/milestones).

The following figure shows a scenario where for the same study, different milestones need to have their own version of metadata (CDISC conformance rules and user's own customized conformance rules).

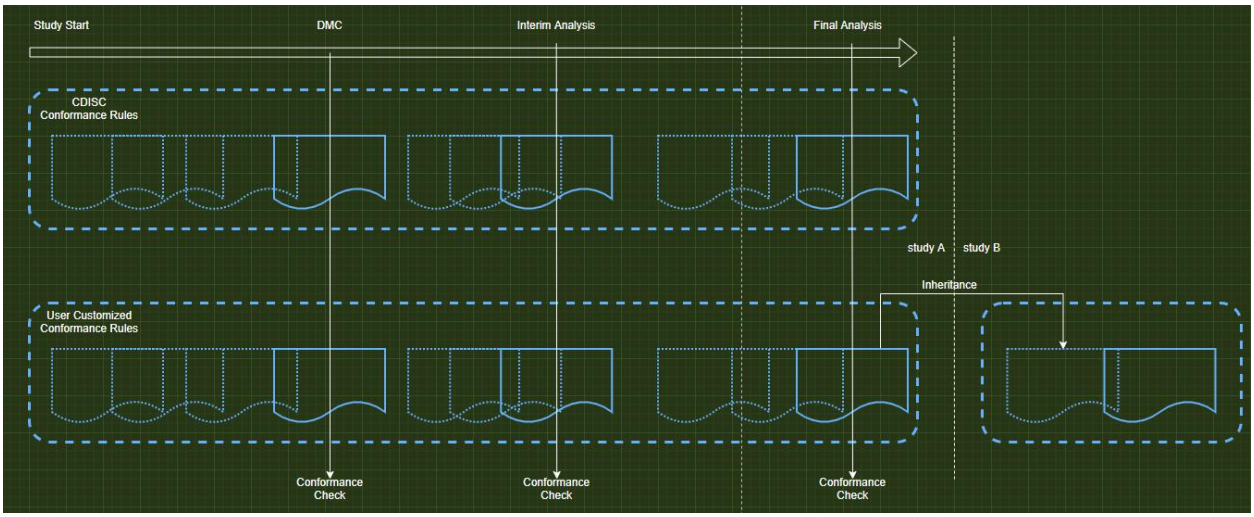

Figure 14 Study timeline and milestones

The left and right sides of the vertical line represent two studies, and the left study has three milestones on the timeline: DMC, Interim Analysis, and Final Analysis. Both metadata change continuously over time (indicated by the dashed document symbols), and at milestones these metadata are formalized (indicated by the solid document symbols) and are used as input for the conformance check.

The version of the metadata at each key time point can be inherited as the starting point for another timeline, as shown by the blue dashed box to the right of the vertical line in the figure.

3.3.2 GRAPH DATA VERSION CONTROL DESIGN

The top left part of the figure below shows the structure of the user customized conformance rule in the neo4j database. The top node is a node with no attributes, indicating that all nodes below it belong to user customized conformance rules. The nodes below this top node contain a version attribute, indicating the version of the customized conformance rule metadata. They are connected to the parent node by a HAS_VERSION relationship.

Each user customized conformance rules version node is connected to a specific rule node through the HAS_RULE relationship. The specific rule node contains the values of all columns of the conformance rule record. They are distinguished by a globally unique identifier *uuid*. For example, user customized conformance rules version 1 contains [uuid1, uuid2, uuid3, uuid4] -- 4 rules nodes, while user customized conformance rules version 2 contains [uuid3, uuid9] -- two rules nodes. The structure of CDISC Conformance Rules is shown in the lower left side of the figure, which is exactly the same as the structure of user customized conformance rules.

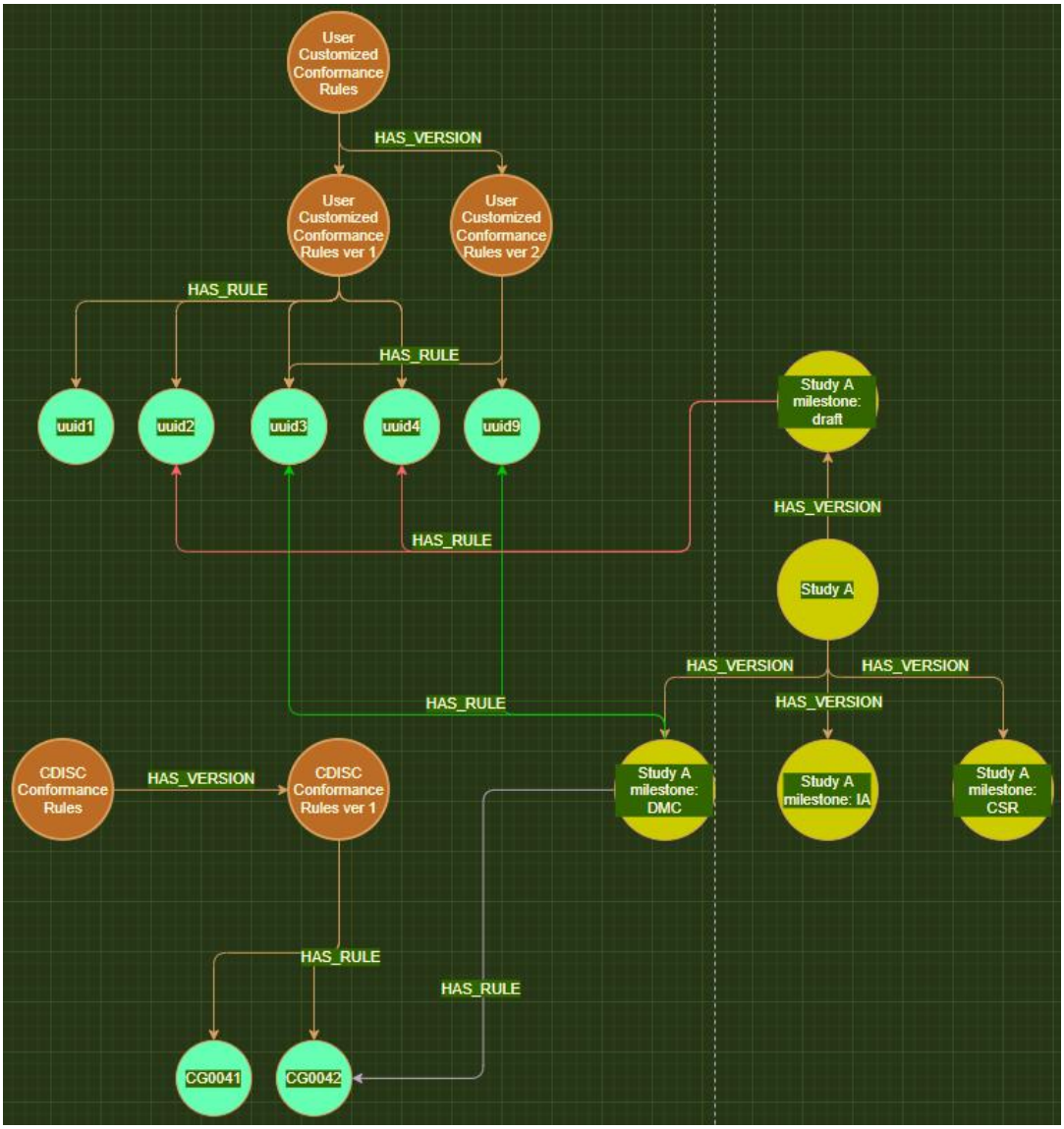

Figure 15 Graph data version control design

On the right side of the figure is the structure of study, containing a top-level node of Study A, which contains the name attribute of study. The study milestone node has a milestone name attribute in addition to the study name. Study milestone nodes are read-only except for the nodes with milestone attribute equals to draft (all edits to customized conformance rules during the course of a study can only be made through the draft node. It represents the continuously changing dotted document icon in the Figure 14). The user can finalize the draft node as a milestone node at a milestone via the API - just create a study A node, set its milestone attribute to the desired milestone name, connect it to the study A node with a HAS_VERSION relationship, and then connect the draft node then copy all the HAS_RULE relations of the "outflow" node to this newly created node.

For another study that needs to inherit the metadata version of a particular milestone from study A, it is also necessary to copy the HAS_RULE relations "outgoing" from the particular milestone of study A to the draft node, and then the node can be used as the starting point for further editing.

The above design of the neo4j database data structure can be exposed to the user through a set of APIs (e.g., using Flask or FastAPI), so that the user's additions, deletions, and changes will maintain this

structure, and the functionality described in the previous section will be continuously available through this structure.

4 MANDATORY REQUIREMENTS FOR CDISC CONFORMANCE RULE SET

Users can have their own defined conformance rules, but this should be used as an additional check rule set. the CDISC conformance rule set needs to be guaranteed each time it is executed. You can check the neo4j database logs to see which rules are requested for a conformance check.

5 CONCLUSIONS

This paper describes a software and database design idea for conformance check. This design allows users to meet the minimum requirements for customizing conformance check even if they do not have access to a commercial conformance check tool and metadata repository.

REFERENCES

Abhinav Jain2012., Ephicacy Consulting Group Inc., “Why organizations need MDR system to manage clinical metadata?” Accessed July 27, 2021. https://www.pharmasug.org/proceedings/2018/SS/PharmaSUG-2018-SS17.pdf#:~:text=MDR%20system%20is%20centralized%20repository%20system%20and%20metadata,a%20cloud-based%20application%20and%20has%20interactive%20GUI%20interface.

Mohamat Ulin Nuha, TU Delft, “Data Versioning for Graph Databases”, Accessed July 20, 2021. https://repository.tudelft.nl/islandora/object/uuid:3cdbd161-3e6b-463a-957d-00ec0942917a/datastream/OBJ/download

Wikipedia, “Single Source of Truth”, Accessed July 23, 2021. https://en.wikipedia.org/wiki/Single_source_of_truth

Wikipedia, “Rete Algorithm”, Accessed June 25, 2021. https://en.wikipedia.org/wiki/Rete_algorithm

CONTACT INFORMATION

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

Dan Li
Merck
dan.li@merckgroup.com

Haiping Yu
dMed Biopharmaceutical Co., Ltd.
haiping.yu@dmedglobal.com