The Practice of Informatics JAMIA

Application of Information Technology ■

Managing Complex Change in Clinical Study Metadata

CYNTHIA A. BRANDT, MD, MPH, ROHIT GADAGKAR, MS, CESAR RODRIGUEZ, MD, PRAKASH M. NADKARNI, MD

Abstract In highly functional metadata-driven software, the interrelationships within the metadata become complex, and maintenance becomes challenging. We describe an approach to metadata management that uses a knowledge-base subschema to store centralized information about metadata dependencies and use cases involving specific types of metadata modification. Our system borrows ideas from production-rule systems in that some of this information is a high-level specification that is interpreted and executed dynamically by a middleware engine. Our approach is implemented in TrialDB, a generic clinical study data management system. We review approaches that have been used for metadata management in other contexts and describe the features, capabilities, and limitations of our system.

■ **J Am Med Inform Assoc.** 2004;11:380–391. DOI 10.1197/jamia.M1511.

Database applications for the management of scientific and clinical data should be easily modifiable and adapt to scientific advances. One way to meet such requirements is to use a generic ("entity-attribute-value" or EAV) data model[1,2] and an architecture driven by metadata, which is loosely defined as "data that describe data."[3,4] Basically, one captures information such as domain-specific descriptions, application conditions, parameters, and methods in a repository. At runtime, software uses this information to perform various tasks such as data validation, interface generation, and customization. While harder to create, metadata-driven software ultimately requires fewer modifications as the domain evolves.

In highly functional metadata-driven systems, the interrelationships within the metadata become complex, and metadata maintenance becomes challenging. If, by accident, a metadata element is defined incorrectly, subtle or major malfunctions manifest in the system's operation; for example, errors may creep into the data, as discussed shortly. For such systems, it is therefore desirable to build a software layer that minimizes the possibility of incorrect metadata, in other words, a metadata management system (MMS). We describe here a rule-based approach to metadata management that lets a system designer define complex constraints that prevent inconsistencies from occurring when changes are made to the metadata or automatically cascades a series of corrective or maintenance actions to maintain consistency. Although we illustrate this approach in the clinical-trials domain, our approach is relevant to other metadata-driven applications, such as clinical patient record systems. It is also pertinent to "knowledge-base" applications, where the distinction between data and program code is blurred.

Background

The Importance of Metadata Management

We use the following classification of metadata[4–6]:

- *Descriptive* metadata support human users concerned with the software's *application domain*. Examples of descriptive metadata are the *semantic* descriptions of a table's columns and controlled vocabularies used to encode clinical observations.
- *Process-related* or *technical* ("active") metadata support *software operation*. Examples of technical metadata are table and column definitions in a database schema, configuration files or programs used for user-interface customization, and definitions that support validation of input data.

This paper focuses on management of active metadata. Incorrect specification of active metadata may have adverse affects on system operation and on data. For example, if a user or developer insufficiently specifies validation constraints for a field, such as data type, range, and regular-expression and cross-field validation checks, invalid entered data may not be trapped.

Active metadata are particularly important for Clinical Data Management Systems, which may be used to store data on

Affiliation of the authors: Center for Medical Informatics, Yale School of Medicine, New Haven, CT.

Supported by grants NIH K23 RR16042, M01 RR06022, P20 LM07253, U01 CA78266, and ES 10867.

Correspondence and reprints: Cynthia A. Brandt, MD, MPH, Center for Medical Informatics, Yale School of Medicine, New Haven, CT 06520-8009; e-mail: <cynthia.brandt@yale.edu>.

Received for publication: 12/09/03; accepted for publication: 03/24/04.

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

patients in general [clinical patient record systems or (CPRSs)], or patients who are specifically enrolled in clinical studies [clinical study data management systems or (CSDMSs)]. Many of these types of systems use an EAV model for storing clinical data (HELP,[7] the CPMC CDR,[8] Logician,[9] the 3M CDR,[10] the Cerner CDR,[11] and the Phase Forward[12] and Oracle Clinical[13] CSDMSs). An EAV model is especially applicable to large-scale repositories that must manage data for an entire institution across multiple clinical domains, as opposed to an individual department. (For an overview of EAV system architecture, see Nadkarni et al.[14]). Among other things, active metadata are used to restructure EAV data into formats better suited for analysis. Such metadata are typically stored within the database itself in a set of relational tables. EAV systems trade off a relatively simple clinical data subschema for a much more complex metadata subschema. The metadata tables outnumber the data tables significantly. (In our CSDMS, TrialDB,[14,15] this ratio is about 8:1.)

The issues concerning the importance and difficulty of managing metadata are old ones, though possibly not as well known as they should be. Historically, the first implementation of a metadata-driven EAV system in medical informatics was the TMR (the medical record) system, created by Stead and Hammond in the 1970s.[16,17] Stead et al.[18] in an overview of TMR as a successful and mature first-generation integration project, describe why pervasive deployment did not follow despite this success. "Several key barriers to widespread implementation were apparent . . . the effort and expertise required to populate facility-specific metadata dictionaries were too great, and the precision with which content meaning could be mapped between various systems was inadequate." The first barrier concerns technical metadata: The prerelational TMR recorded technical metadata in a rigidly structured file that was the equivalent of multiple relational tables, and the difficulty of ensuring this file's correct syntactic structure complicated the task of ensuring that its contents had the intended semantics. The latter barrier concerns descriptive metadata. At the time, controlled vocabularies such as National Library of Medicine's Unified Medical Language System[19] were absent.

Metadata Management Approaches

Metadata management is a specific instance of what is called "business logic" in the database literature. Microsoft's definition of this term (generalized by us to eliminate the narrow focus of "business") is "the combination of validation edits, logon verifications, database lookups, policies, and algorithmic transformations that constitute a system's intended behavior." For our specific focus, system integrity, we use the term *integrity logic* (IL) instead. We now consider the various means of enforcing IL.

Using DBMS Facilities

The most obvious way to implement IL is to use the facilities of the database management system (DBMS) that hosts the data. The simplest approach is the use of **declarative constraints**. Declarative means that the developer specifies ("declares"), using a concise syntax, what the desired result should be, and the system takes care of physical implementation details. Examples of constraints are

- *Unique constraints*: In a table of patient visits, the combination of patient ID, date and time of visit, and location of visit should be unique for a given row in the table.
- *Referential integrity constraints* between a pair of tables. A valid patient ID in the visits table must already exist in the patient demographics table, and a patient cannot be removed from the demographics table as long as visits for that patient exist in the visits table.

It is not always possible to specify IL declaratively. For example, in a hospital pharmacy system, the act of dispensing a quantity of drug should appropriately decrease that drug's total quantity in inventory. Such constraints must be implemented within a database as **procedural code** ("triggers") that is stored within the database itself. Modern DBMSs ensure that trigger code executes transparently and automatically when invoked by the associated event (e.g., adding a record to a dispensing table) and cannot be subverted (e.g., by someone with power-user privileges connecting directly to the database).

Using Non-DBMS Middleware

This ideal situation, however, is not always realized in practice. Stored code is typically written in a vendor-specific variant of SQL (Structured Query Language, the lingua franca for data definition and manipulation in relational DBMSs). Current vendor SQL dialects are limited by the absence of support for complex data structures such as arrays. (SQL-99, the new SQL standard, defines arrays, but not all the mainstream DBMSs are even fully SQL-92-compliant.) As a result, certain complex constraints, such as those based on the order of elements in a collection, are very difficult or impossible to express with database-stored code, even though they would be trivial to express in a modern, general-purpose programming language.

The second-best situation is to use "middleware" or "middle-tier" software—a software layer that intervenes between the end user ("client") application and the database. An approach that has gained popularity for this purpose is the "business-rules" paradigm, advocated by authors such as C. J. Date,[20] Ronald G. Ross,[21] and Barbara Von Halle.[22] Business Rules (BR) software utilizes technology that was originally developed for knowledge-base consultation systems ("expert systems"). The system designer declaratively specifies "rules" that codify intended system behavior. Appropriate use of such software requires knowledge of the concepts underlying expert systems, e.g., predicate calculus, forward chaining, rule conflict. Declarative knowledge, when specifiable, is very high level: an engine ("business logic server") that can interpret or compile these rules enables domain experts to implement and test new rules in hours to days, as opposed to weeks or months if programmed by computing staff from first principles. BR systems thus hold the promise of being able to modify system behavior to meet changing needs in "Internet time." As we discuss later, however, this promise is rarely fulfilled in reality.

Related Work on Metadata Management Applications

Several papers and books have described the benefits of managing metadata for data warehousing and the approaches used.[4,5,23] Vaduva et al.[23] define an explicit metadata management subschema for the purpose of data warehousing, along

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

with various "operators" required to preprocess data prior to importing into the warehouse. They also identify version control as an important component of metadata management, although they do not address it in their work. Their paper reiterates the limitations of current SQL implementations in terms of expressivity for defining constraints and operators.

Schulz et al[24] describe a database containing about 500 rules for quality assurance of the Read Codes controlled vocabulary, an instance of descriptive metadata. While the rules were defined in simple English, they needed to be implemented through a variety of distinct mechanisms, including database constraints, stored procedures, regular expression checking, and UNIX scripts. Cimino et al[25] describe a knowledge-base curation and maintenance effort for the Columbia Medical Entities Dictionary (MED), an institutional controlled vocabulary, that uses a Massachusetts General Hospital Utility Multi-Programming System (MUMPS) environment.

The Current Problem: Pilot Study

TrialDB, a generic open-source CSDMS for multicentric, Internet-based clinical studies, uses study-designer–defined metadata, such as the detailed definitions of individual attributes ("questions") and the case report forms (CRFs) that contain them, to automatically generate Web data-entry forms. These forms include sophisticated features such as advanced validation, automatic recomputation of certain fields through formulas based on other fields, as well as conditional enabling/disabling of fields based on values entered in other fields—so-called skip logic. The forms are generated as active server pages (ASP), which include both static and dynamic elements.

As TrialDB's feature set grew, interdependencies between the various schema components became progressively harder to maintain, and we therefore deemed it desirable to maintain knowledge about these dependencies in a centralized fashion as a database of use cases. We had previously created a mechanism for centralized documentation about schema dependencies in the form of metadata tables that describe the complete database schema: This approach has been previously described in the EAV/CR (EAV with classes and relationships) framework paper of Nadkarni et al.[26] Briefly, all tables in the database and information such as their attributes and constraints (primary and foreign keys) are documented in tables (Meta_Classes, Meta_Attributes, Meta_Joins and others), which also contain entries describing themselves, so as to serve the purpose of self-documentation. The contents of tables are bootstrapped from information in the database catalogs, with documentation added by developers. Other software modules in the system use this information for purposes such as data import and export and ad hoc query. Although such information is centralized, it is not sufficient by itself because it does not contain the data structures required to model more complex dependencies and is therefore unable to prevent inconsistencies in the metadata.

While the IL approach could be applied in principle to the problem, we soon realized that intrinsic database mechanisms would not suffice. For example, consider the use case "changing the order of questions in a form," a common action when a CRF is being developed in collaboration with investigators. Some of the conditions that must be handled are discussed below.

- Based on elementary user-interface guidelines, if a question is disabled based on the value of another question, then the former question must follow the latter. Similarly, questions based on computed formulas must succeed the questions on whose values they depend. Any attempted change in question order that would violate either of these rules must be blocked. The algorithm for checking consistency in both cases is straightforward and based on loading the questions into an array, sorted by putative question order, and then checking for the error condition by iterating through the array. This algorithm is not expressible conveniently in SQL.
- The final action in the implementation of many constraints involves access to the operating system, specifically, regenerating the form on the designer's machine and copying it to a network drive.

In a pilot exploration, we first defined, in declarative English pseudo-code, the steps required to deal with this use case. We then explored whether existing BR software could allow its implementation. We discovered the following practical issues.

- Each vendor's approach—specifically, the declarative language used to specify rules—is *proprietary*. The commercial packages' cost runs into the tens of thousands of dollars, and there is a real risk of vendor lock-in, especially given that there is no widely supported standard for rule interchange, although many have been proposed such as the Knowledge Interchange Format (KIF),[27] Business Rules Markup Language (BRML),[28] and RuleML.[29]
- There are also serious questions about *vendor longevity*: There is a chance that the attrition of commercial AI vendors in the 1980s may repeat in the BR market. (In the Discussion section, we highlight certain impending advances in DBMS engines that may greatly reduce the role of middle-tier business logic servers and, at the very least, will necessitate a complete reworking of BR software architecture.)
- *Expressivity*—the types of operations that are supported by the software—is a major concern.
  - Some commercial systems have arbitrary restrictions in rule grammar. For example, IBM's CommonRules, a Java library, disallows the use of simple arithmetic expressions in a rule condition.
  - BR systems that act as middleware for a database—in particular, software that needs to perform corrective database updates in response to specific events—must be *database aware*. It must not impose artificial hurdles with respect to being able to perform actions that would be trivial if done via SQL statements. Some expert-system shells, e.g., the widely used CLIPS[30] and JESS (Java Expert System Shell)[31] programs, lack database awareness, and it is not trivial to add this capability.
- *Extensibility*, the ability to modify the system for needed domain-specific functions such as CRF versioning, is also a concern. One must be able to reuse code already developed for a production system and invoke existing subroutines without requiring that they be rewritten because the software is unable to utilize them.

We experimented with a trial copy of the Microsoft.NET version of a commercial program, QuickRules,[32] which appeared

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

promising and potentially extensible. QuickRules has an easy-to-use graphical user interface (GUI) for defining rules. We found, however, that passing information from the database to QuickRules required writing a significant amount of code to create numerous classes and considerable effort to test each of them before the rule itself could be tested. This defeated the promise of rapid turnaround, and maintainability was foreseen as a major problem.

Because of these and other issues, we chose to develop our own MMS within TrialDB.

Design Objectives

The TrialDB system MMS needed to be able to perform the following types of actions:

- *Preventive action:* Certain superficially trivial operations must not be allowed under some circumstances, as described in the previous scenarios for question order in a form. For allowable changes, the system must automatically perform cascading actions to correct and maintain consistency of the metadata. Examples of such actions are now described.
- *Corrective actions:* TrialDB stores EAV data in separate data type–specific table, for integers, floating-point numbers, strings, dates, and binary data such as images. Occasionally, the data-capture requirements of a newly defined question may change, and its data type needs to be altered. If clinical data for that question already exist, one must move the data from one data type–specific EAV table to another, with appropriate data-type conversion if possible.
- *Maintenance actions:* A permissible change to question order requires the Web form for that CRF to be regenerated and copied to a folder on a Web server (accessible as a network drive). When a meaningful number of changes have been made to a CRF, a new version is created, the old version is archived, and the changes (revisions) are rolled up for documentation purposes.

The MMS would be evaluated by the following criteria:

- Ease of definition, testing, and modification of constraints: the constraint-development and testing environment should use a GUI.
- Reusability: Common actions that may be used in different circumstances should need to be defined just once and invoked with different parameters as needed.
- The expressivity of the system should be sufficient that it should be significantly simpler to specify the desired actions through the MMS and have them executed automatically than programming the same actions through hand-written code.
- The MMS should be able to utilize the existing development environment fully, and access to the database schema should not be restricted artificially.
- There should be no arbitrary restrictions in what the system cannot do, and one must be able to mix and match approaches. For example, in some circumstances, one may simply wish to call a function that is external to the MMS and use its result in a constraint, as opposed to being forced to (re)write it using a proprietary, and possibly much less versatile or expressive, rule language.

System Description

The MMS consists of a *knowledge-base*, a *runtime rule execution engine* that operates on the knowledge base and a *rule editor/testing environment*. Figure 1 is a UML (Unified Modeling Language)[33] diagram showing an overview of the MMS system. The knowledge base is a subschema of the database, which is described in greater detail in Appendix 1 (available as an online data supplement at http://www.jamia.org); the main elements of the schema are described below along with the engine. The editor is used to define use cases, edit and test them in an iterative fashion, and finally "compile" rules. A component that runs within the client's process invokes the rule engine. In the description below, we use the terms *constraint* and *rule* interchangeably because a rule in our system is absolute rather than probabilistic/heuristic and has no exceptions.

Rule Engine Overview

- The rule engine evaluates sets of *rules*, which are grouped into *use cases*. A rule consists of a *condition* (antecedent, premise) and a series of *actions* that execute if the condition evaluates to true. A rule may have an empty condition, in which case, it acts as a subroutine that groups sets of actions together.
- *Facts* are analogous to variables; their values are tested or used in conditions. Facts may be *single valued* (primitive), such as strings, numbers, and Booleans, or *multivalued* (arrays).
- The rule engine uses forward chaining, i.e., firing specific rules when new facts are inferred, or when the value of particular facts changes. It utilizes the Rete algorithm, which was originally devised for the production-rule language OPS5[34] and which has also been implemented in JESS. Rete is an efficient linear-order algorithm to optimize the identification of rules that are affected (activated or invalidated) by an assertion of, or change in, a particular fact, or the facts that in turn depend on a rule. Rete also avoids the re-evaluation of facts that have been previously evaluated. Newly asserted rules are activated for firing in the order of priority. Also, previously activated rules that become invalid because of the newly asserted facts are implicitly retracted from the firing queue.
- The MMS follows the design of modern compiled languages in two respects: (1) MMS uses *strong typing*. That is, the data type of a fact must be specified in advance so that the validity of many operations can be checked statically, and the comparison of, say, a string to a number would be flagged as an error during rule development. (By contrast, JESS and many other rule engines postpone such checking until runtime, which significantly lengthens the debugging cycle.) (2) Facts used in a rule must be *defined ("declared") in advance* as entries in the database. Undeclared facts are flagged as errors during development time.
- The rule condition grammar is described, as a *yacc*[35] specification, in Figure 2. (Yacc is a well-known tool, originally developed for the UNIX platform, which generates parsers based on a high-level Backus-Naur-Form (BNF) description of the parser grammar.) A rule condition allows specification of relational operators (=, <, etc., plus set membership), simple arithmetic operators (including exponentiation), and logical operators (not, and, or). By using

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

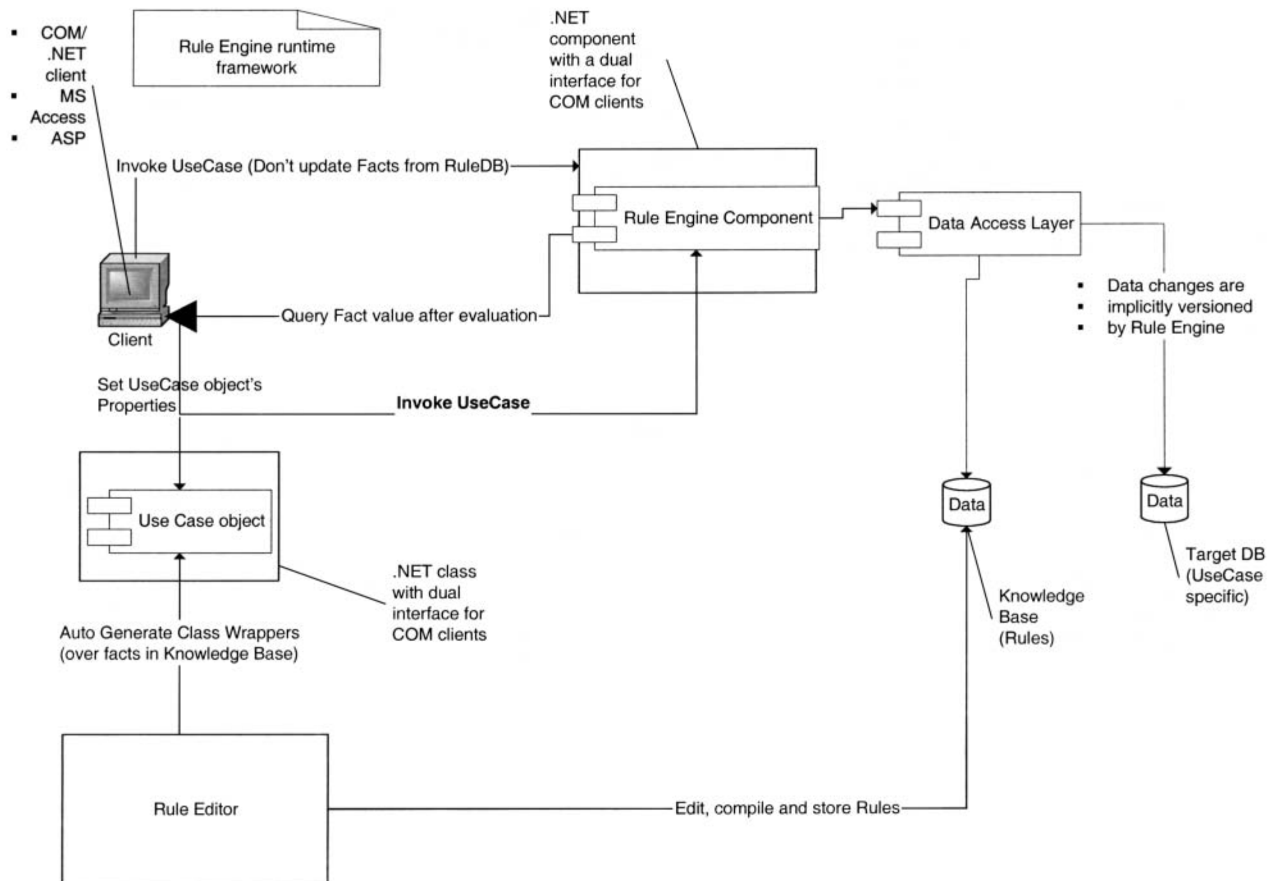

Figure 1. Metadata Management System architecture. This is a UML diagram of the components for the system.

parenthesized expressions, a condition can be arbitrarily complex.

- An important feature of the grammar is that *functions* may be invoked as part of the condition. These functions may be part of the Visual Basic environment, external dynamic-link-library functions called from Visual Basic, or developer-written functions written expressly for use by the rule engine. This allows the system to be extended without having to devise a complete programming language. For this purpose, we utilize a function of the Visual Basic for Applications (VBA) language called *eval*. *Eval*, a feature of languages with artificial-intelligence capability (LISP, Prolog), as well as Perl and some "scripting" languages, blurs the distinction between program code and data by allowing evaluation of a string of data, or metadata, as though it were program code. If the string contains syntactically correct code, it is executed, and a value is returned.

  The advantage of *eval* is that its availability reduces the effort required to build an expression evaluator drastically. Some implementations of *eval* also permit the global system state to be changed: This is not necessarily a good thing because code with potential side effects may be malicious, and there is no simple way to ensure that it is trustworthy.
- An *action* can perform several tasks. Apart from assignment or output (display of messages), it can execute intrinsic, external, or developer-defined subroutines and can also execute SQL statements. Further, an action may invoke other rules in subroutine fashion. An action that operates on a multivalued fact performs an implicit iteration over all fact members.
- The current version of the MMS has a design limitation in that it is *not re-entrant*. That is, because it utilizes global variables and because its actions depend on the global state of a database, only one instance of the rule engine can be running at a time. Attempts by multiple developers/designers to make simultaneous changes are addressed through a serialization/queuing mechanism at the middle tier where the engine resides. For the domain of metadata management, we believe that this restriction is reasonable because metadata changes, apart from being relatively infrequent, often influence the system globally.

  While some metadata changes may be local in impact (e.g., they may be restricted to a particular form), a priori differentiation of changes with local impact from those with potential global impact is not simple.

The Rule Editor

The developer defines and tests rules through the Rule Editor's GUI, which is essentially a forms-based interface to the knowledge-base subschema. Features of the editor environment are described below.

- The designer switches between the top-down and bottom-up styles of development. Top-down development involves

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

Following Yacc usage, terminals are indicated in uppercase.

```
condition_stmt ->     condition;

condition      ->     condition OR condition
                      |      condition AND condition
                      |      '(' condition ')'
                      |      term;

term     ->     '(' term ')'
                |      NOT term
                |      scalar_exp RELATIONAL_OPERATOR scalar_exp
                |      scalar_exp      %prec  '('
                |      scalar_atom IN FACT;
// the last tests for set membership, fact is an array

// RELATIONAL_OPERATOR is one of: =, <>, <, >, <=, >=.

function_call  ->     function_name  '('      expr_list        ')';

function_name  ->     function_namem'.' NAME // method invocation
                      |      NAME; // pure function invocation

expr_list      ->      // Empty
|      scalar_expr
|      expr_list  ','  scalar_expr;

scalar_exp     ->     scalar_exp      '+'     scalar_exp;
                |      scalar_exp      '-'     scalar_exp
                |      scalar_exp      '*'     scalar_exp
                |      scalar_exp      '/'     scalar_exp
                |      scalar_exp      '^'     scalar_exp      //exponentiation
                |      '+'     scalar_exp      //positive number
                |      '-'     scalar_exp      // negative number
                |      '('     scalar_exp      ')'
                |      scalar_atom;

scalar_atom    ->     FACT   // a name defined in the FACTDEFINITION table
                |      literal
                |      function_call
                |      FACT index
                |      FACT '.'   'Count' ;
// the last counts number of items in a set, where fact is an array

index    ->     '[' FACT ']'                  // hash-table reference
         |      '[' NUMBER_LITERAL ']';       // array reference

literal  ->     STRING_LITERAL // string constant in double quotes
                |      BOOLEAN_LITERAL //True or False
                |      NUMBER_LITERAL; // integer or decimal number
```

Figure 2. Yacc specification for a rule condition.

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

defining a use case, and the rules that comprise the use case in skeletal form. Bottom-up development involves defining the facts that are used in a rule condition and the rule actions. Ideally, facts are defined before the condition and actions are written: Fact-names are available for selection in a list box during rule composition to minimize typing errors.

- Various static checks are performed during rule definition, for example, checking for undefined facts, type checking of expressions involving relational comparisons, or ensuring that array-index expressions are only permitted for array-type facts.
- A rule may be given a *priority* to indicate its order of evaluation within the use case. While some authors, such as Ross, indicate that a rule-based system should not rely on rule prioritization and infer the correct order of execution, the reality of our application domain is that rule conditions must often be tested in a particular sequential order, for example, one rule may assign a value to a fact, and other rules test this fact for values. There should be no reason to prevent a designer who knows the domain from telling the engine how to proceed.
- A rule can be tested by temporarily assigning particular values to the facts that it uses. Since fact definitions are stored in the database, these assignments persist between testing iterations until they are changed. During testing, the entire expression in the condition is *eval*-ed. After the rule has been tested and determined to work as intended, it is "compiled" to generate a stack-machine equivalent of the condition expression and to create structures utilized by the Rete algorithm. Compilation is partly an efficiency/scalability consideration; however, it also allows creation of a dynamic-link library that references the rule engine and its compiled use cases, by creating "wrappers" around the latter. The developer of client-side metadata-editing code uses this library, as described later.
- Testing is done on a test version of the TrialDB database, which is a complete clone of the production system that is regenerated every night. This is because certain actions in a rule may alter the contents of one or more database tables. While we use the DBMS's transaction mechanisms to make sure that all actions in a use case are executed as a single atomic unit, the correctness of the operations can often be verified only by inspecting the contents of the tables concerned after use case execution has completed. If tested directly on a production system, incorrectly specified rules can have unfortunate consequences that may not be rectifiable without much manual metadata re-entry or a full restoration from a database backup. We have an option where actions that cause database change may be simulated rather than actually executed, by echoing messages to the user. In the final stages of rule testing, however, there is no substitute for effecting and testing database changes.

Version Control

Versioning support is an important issue for metadata. Since metadata determine system behavior and the meaning of data, it is important that metadata evolution be tracked, audited, and documented. For example, based on interim analysis of data, the questions in a particular CRF may be modified, e.g., rephrased or reordered, or new questions added to better meet the study's research goals. The U.S. Food and Drug Administration's 21 CFR part 11 regulations require preservation of all versions of case report forms used in a clinical study.[36]

In the TrialDB MMS, the concept of "versioning" is currently applicable only to CRFs. While all changes to items are stored in metadata audit tables, new versions of forms are not automatically machine generated: The study designer and/or the principal investigator determine what set of changes constitutes a new version. For example, a new version may be required when a single metadata item on the CRF changes or a number of items on the CRF change. Most commonly, a cosmetic change, such as a rearrangement of questions that have not been reworded and addition of a heading, does not require a new version, although some investigators' protocols may require it.

We create a new CRF version through a transaction-based mechanism to prevent concurrent changes to the CRF by other designers. The rule engine then stores a partly autogenerated summary of the changes for that version based on the series of atomic changes that have been made. The Web forms and hard-copy equivalents (MS Word files) representing the older version of the CRF are archived in a source-code control system. Following the approach of modern version-control software, the atomic changes to CRF elements are stored as "reverse deltas" (the differences between the current version and the previous version, expressed in terms of the current version). The subschema for MMS versioning is described in Appendix 2.

Connecting the Middleware to the TrialDB Microsoft Access Client

The rule engine is currently implemented as middleware, which means that it is the responsibility of client-code developers to invoke it appropriately at various points in the GUI code for metadata definition. This GUI is currently hosted in a Microsoft Access application. The developer links the rule engine and its compiled use cases to this application. Through Microsoft's IntelliSense technology, the symbolic names of use cases and the facts that each use-case utilizes become available during code development: one can now perform input-output operations on the facts of a use case. This makes coding easier and minimizes logical coding errors.

Example of a Rule

If the user changes the order of questions in a group of questions (e.g., moves the third question down to the eighth position in a group of 20 questions), the "Change Question Order" use case will be called from the Access Client Visual Basic code. Figure 3 is a screen shot of the rule editor showing the rules for this use case. Here is a summary of the main steps in this use case (it has 12 rules that invoke a total of 25 actions). Note that most of these steps will be performed individually through fairly complex procedural code.

- Assign values to the facts used in the rules such as the question ID and current serial number (3) of the question being moved, along with the serial number it is being moved to (8).

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

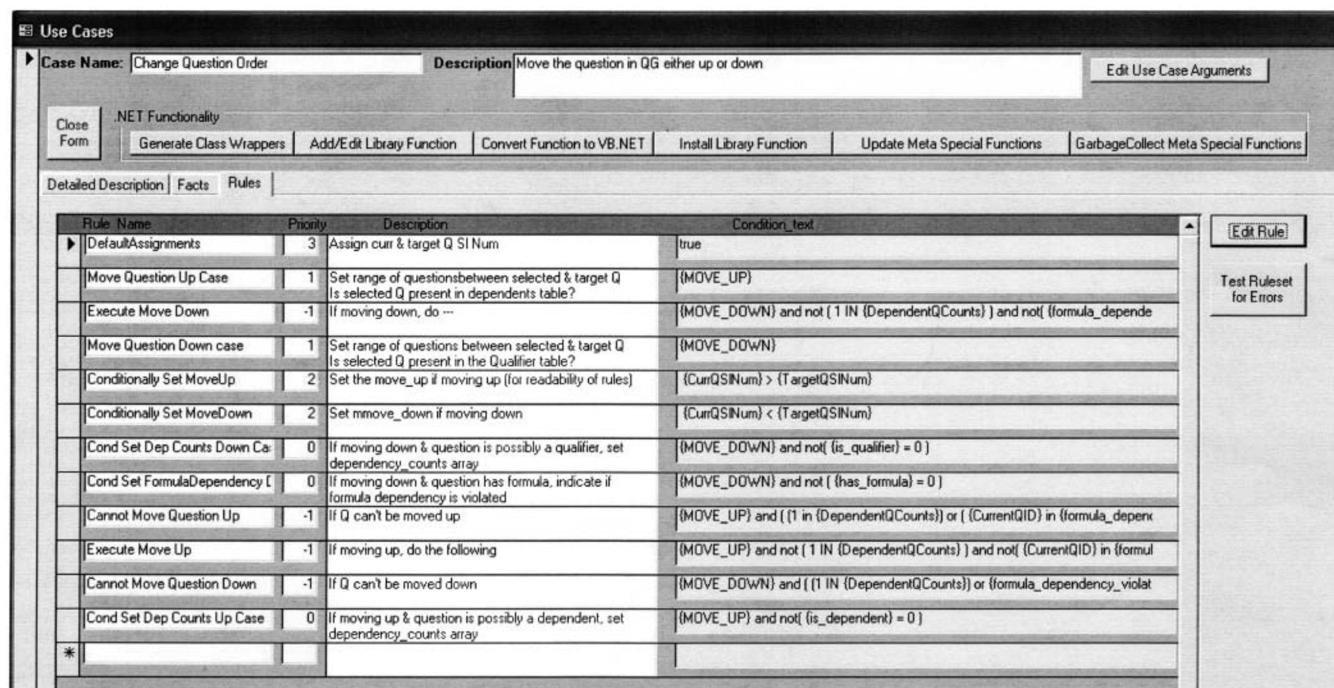

Figure 3. Screen shot of the rule editor showing the 12 rules and their priority, description, and condition text for the Change Question Order use case.

- Check to see whether the question is being moved up or down in serial number order. The order of movement is important because: (1) if a question is moved up, we must ensure that it is not dependent on questions that were previously ahead of it but now succeed it and (2) if a question is moved down, we must ensure that other questions that depend on it do not precede it.
- Get a collection of all the questions that are affected by the move, i.e., questions between the old and new positions.
- If the target question is to be moved down, check to see whether it controls CRF skip logic for questions in the "affected questions" collection or is used in the computed formulas of questions in this collection.
- If it is to be moved up, check whether another question in the "affected questions" collection controls its editability via skip logic or whether its value is the result of a computed formula based on a question in this collection.
- If either of these conditions is true, validate dependencies to determine whether change in position is allowed.
- If the change violates the set conditions (e.g., a skipped field should not precede the question whose value determines whether to skip that field), stop the process and notify the user by composing appropriate detailed diagnostic messages.
- If the question can be moved, then move it by performing a SQL update.
- Save the changes in versioning tables.
- Regenerate all dependent Web data entry forms that use this group of questions. Move these forms to the Web server.

If the use case is prevented from executing successfully, an administrative user must examine the error diagnostics and take appropriate action. (For example, move other questions first before trying to move the third question.)

Status Report

The MMS has been moved into production. We now describe our initial subjective evaluation based on our evaluation criteria.

- *Ease of definition, testing, and modification:* We realize that a task is as easy or as difficult as the problem it seeks to address. As the "Change Question Order" use case illustrates, there are no short cuts to thinking clearly and hard about the problem and describing the steps in plain English/pseudo-code before trying to implement them in a rule engine. The advantage of documenting this pseudo-code within the use case itself makes the system more understandable for everyone.
- *Reusability:* Reusability is implemented through the equivalent of a subroutine capability (the Action_Exec_Rules Table). Identifying common tasks that can be reused across rules has facilitated this goal. Examples of such tasks are "move data for an attribute from one table to another" and "regenerate form and move to network drive." These tasks must be implemented, but once this is done, they become reusable building blocks.
- *Expressivity:* Ease of defining specific use cases is relatively flexible. One can cover the domain without requiring the developer to learn new languages or recreate existing methods and operations. There are no arbitrary restrictions in what a rule cannot do, a point amplified in the Discussion.
- *Use of existing environment, access to database, and lack of arbitrary restrictions:* The VBA environment provides reasonable access to the database (in part through wrapper functions that we have written in the context of other projects) and makes creation of a GUI straightforward with a very modest level of coding. Availability of *eval* has

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

greatly facilitated extensibility: This function provides access to fairly complicated procedural code whose details can be hidden.

The lessons that we have learned are summarized below.

- The initial steps are the hardest. Defining the building blocks and implementing them take time; as more experience accumulates, more blocks need to be added. Once implemented, however, productivity increases greatly because one is freed from dealing with low-level details. The building blocks are mostly procedural code, and their correct implementation depends on an in-depth knowledge of the metadata schema as well as the clinical studies domain. This is where the majority of effort has been expended, and we doubt that commercial "business-rule" systems, which focus on declarative rules, would result in significant productivity gains for the present problem.
- The development paradigm—maximizing the number of errors that are caught at rule definition time rather than at testing time—requires the users to develop a particular discipline, e.g., defining facts before using them.

We have documented and implemented ten of our more important and difficult dependencies as use cases. There is on average 4.5 rules per use case (range, 2–12). About an equal number of dependencies still exist as procedural code that is currently scattered in various places in the TrialDB client code, and the act of centralizing them will improve the system's documentation and result in a code clean up, improving its maintainability.

While this article has addressed CRF generation primarily, we also have use cases that go beyond CRFs. For example,

- We address the issue of consistency of choice sets (the list of valid, discrete responses to specific questions): inappropriate modifications to a choice set may "orphan" existing stored data irrespective of the CRF issue.
- Randomization of patients into particular arms of a study is driven by the values of particular decision variables (e.g., the patient age, gender, stage of disease). Removing a question from a study requires that it also be removed from randomization criteria.
- Permissions of users within and across studies within TrialDB are determined by user-privilege metadata, which must not be inconsistent. (For example, if a user has permission to add data to a specific study, he or she must have view privileges as well.)

Performance

Performance of the system is reasonable. The limiting factor in rule evaluation is the database access required by the rule condition or the execution of individual rule actions: The time for expression parsing itself is negligible. For example, the "Change Question Order" use case runs in less than a second even for large forms. The use case "change a question's data type," however, could take significantly longer if there was a significant amount of legacy data in existing tables that needed to be converted.

Discussion

The management and control of metadata in metadata-driven systems are vital. The metadata hold the system's knowledge, and their consistency determines system integrity. Our approach to this problem is to use database mechanisms where possible but document them centrally. For all other cases, use a knowledge base to serve the dual purposes of centralized documentation and consistency maintenance through executable rules.

Rule-based approaches have been used successfully since the 1970s in medical informatics to manage knowledge for clinical "expert systems" such as Mycin[37]; however, the idea that all knowledge can be defined through rules is now known to be an oversimplification.[38] While a rule engine drives our MMS, the rules are only scaffolding: the hard work is really performed by procedural code that is invoked as part of a rule condition or during execution of rule actions. While *declarative knowledge*, such as a rule, is appropriate for certain tasks (e.g., base SQL is itself a declarative language), one of the limiting aspects of many commercial rule-based systems is their de-emphasis of *procedural knowledge* or restrictions on the types of procedural actions that are supported. Imposing a declarative paradigm for a complex task that is naturally specified procedurally is not conducive to designer/developer productivity. The decision as to whether to buy a BR system or build one must therefore be made carefully, based on several issues, notably the development environment, extensibility, database and operating-system access, and ability to reuse existing code libraries.

Commercial BR systems as well as our own MMS are based on a middleware approach. For middleware to operate, the client-application software that connects to it must be appropriately re-engineered. This may not be a trivial task, and further, a person with direct administrator-level access to the database could bypass and subvert the middleware completely by, for example, entering SQL statements through a command line. Modern DBMSs increasingly support greater extensibility at the database tier. For example, Oracle 9i allows database-stored procedures to be written in Java instead of Oracle's procedural extensions to SQL. While powerful, Oracle's approach has some limitations: Java does not support *eval* because of the requirement for code safety; Oracle-stored procedures cannot access the operating system or existing compiled libraries written in other languages because they do not support the Java Native Interface (JNI); and there is no support for integrating debugging of stored code.

A version of MS SQL Server (code name Yukon[39]) that is currently in beta is intended to allow writing of stored-procedure code in any .NET-supported language, with full access to the .NET class libraries and integrated debugging in the Visual Studio .NET environment. The arrival of software such as Yukon (and possible improved versions of Oracle) will require that BR vendors re-engineer their software completely to generate back-end code rather than rely on the present middleware approach.

The Issue of Generalizability

It may be possible to apply our approach to other problems of metadata management in biomedicine. To transplant it to a different domain, only the metaschema (described in Appendix 1) and the rule engine that operates on it would be necessary. One replaces the remainder of the database schema with tables that are appropriate to the problem at hand: For that matter, even an EAV data model is not necessary.

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

Generalization is possible because the metaschema treats the rest of the schema, or application-specific code, as *data* rather than hard-coding references to it. The data are typically components of rules. The data take the form of

- SQL statements that are stored as text and executed on demand.
- Machine-generated object IDs that reference the names and descriptions of individual tables or columns in the rest of the database, which are stored in the metaschema.
- Text that invokes external programs/utilities by being passed to the operating system.
- Text that invokes application-specific subroutines, which can be executed using the *eval* mechanism, described earlier.

For specific applications, the rules would obviously have to be rewritten. Since, however, a rule is not arbitrarily restricted in what it may not do, the decision as to how to proceed with implementation, e.g., whether to emphasize SQL or the use of external, possibly preexisting programs, is up to the developer. For example, the Read Codes curation work of Schulz et al. (described earlier) employed a variety of UNIX utilities extensively, and a framework like ours could serve as scaffolding for controlling the invoking of these utilities. Further (while we have not actually attempted this task), the concept-subclassification algorithm described by Cimino et al. in the work cited earlier is expressible in a fairly straightforward manner using code that combines procedural control with SQL operations.

The ability to use *eval* might also make an approach such as ours worth exploring in knowledge-base systems, such as systems that store computable guidelines. The advantage of *eval* in languages that support it is that it saves the considerable effort of building an expression parser and provides extensibility with respect to being able to process arbitrary functions. Strictly speaking, while we built a rule condition parser (which still uses *eval* internally for function evaluation), this was primarily an issue of efficiency/scalability (as was the use of Rete) and secondarily a means of supporting static checking for expression correctness. The next version of the Microsoft .NET framework, which has a security architecture similar to that of Java, will support a restricted (secure) *eval* through the mechanism of anonymous ("lambda") functions, a feature first implemented in LISP.[40] The price of *eval* is modest computational inefficiency because of runtime interpretation. For database-intensive systems, however, the factor limiting performance may be disk I/O rather than rule parsing and evaluation per se.

Future Plans

We are currently experimenting with the Yukon beta. When the final release arrives, we intend to modify the engine and editor so that they generate back-end triggers in a .NET language. For example, the use case "Change Question Order" would be an update trigger on the "QuestionGroup_Questions" table, which contains serial-number information on the order of questions within a question group. The trigger would be fired by a change in the Serial_Number field of any record in this table. The knowledge base and the editor itself should not need to change, but the editor's "compilation" action will not need to create class wrappers for client hookup. Client application development will consequently be dramatically simpler because consistency maintenance is performed transparently behind the scenes.

As with any knowledge base, the process of creating and defining the use cases (knowledge) for a complex system is time-consuming and error prone, and maintenance is an issue.[41] Graphical user interfaces may assist in the process, but there must be diligence and effort put into correctly and adequately documenting and testing the constraints created so that errors such as redundancy, missing rules, dead-end rules, and incomplete and inconsistent rules can be prevented. We plan to increase our emphasis on alternative methods to improve validation and testing of the rules and use cases.

References ■

1. Johnson S. Generic data modeling for clinical repositories. J Am Med Inform Assoc. 1996;3:328–39.
2. Huff SM, Haug DJ, Stevens LE, Dupont CC, Pryor TA. HELP the next generation: a new client-server architecture. In: Proceedings of the 18th Symposium on Computer Applications in Medical Care, 1994. Washington, DC: IEEE Computer Press, 1994, pp 271–5.
3. Johnson S, Cimino J, Friedman C, Hripcsak G, Clayton P. Using metadata to integrate medical knowledge in a clinical information system. In: Proceedings of the 14th Symposium on Computer Applications in Medical Care, 1990. Washington, DC: IEEE Computer Press, 1990, pp 340–4.
4. Vaduva A, Vetterli T. Meta data management for data warehousing: an overview. Int J Cooperative Inform Syst. 2001;10(3): 273–98.
5. Marco D. Building and Managing the Meta Data Repository: A Full Lifecycle Guide. New York, NY: John Wiley & Sons, 2000.
6. Solbrig HR. Metadata and the reintegration of clinical information: ISO 11179. MD Comput. 2000;17(3):25–8.
7. Kuperman GJ, Gardner RM, Pryor TA. HELP: A Dynamic Hospital Information System. New York, NY: Springer-Verlag, 1990.
8. Friedman C, Hripcsak G, Johnson S, Cimino J, Clayton P. A generalized relational schema for an integrated clinical patient database. In: Proceedings of the 14th Symposium on Computer Applications in Medical Care, 1990. Washington, DC: IEEE Computer Press, 1990, pp 335–9.
9. GE. Logician Product Information, 2004. Available at: http://www.medicalogic.com/products/logician/. Accessed Feb 9, 2004.
10. 3M Health Information Systems. The 3M Clinical Data Repository, 1998. Available at: http://www.3m.com/market/healthcare/his/us/products/care_inno/. Accessed May 24, 2004.
11. Cerner Corporation. Information available at: http://www.cerner.com. Accessed May 24, 2004.
12. Forward P. Phase Forward, 2004. Available at: http://www.phaseforward.com/. Accessed Feb 4, 2004.
13. Oracle Corporation. Oracle Clinical Version 3.0: User's Guide. Redwood Shores, CA: Oracle Corporation, 1996.
14. Nadkarni PM, Brandt C, Frawley S, et al. Managing attribute-value clinical trials data using the ACT/DB client-server database system. J Am Med Inform Assoc. 1998;5:139–51.
15. Brandt CA, Nadkarni PM, Marenco L, et al. Re-engineering a Database for Clinical Trials Management: Lessons for System Architects. Control Clin Trials. 2000;21(5):441–61.
16. Stead WW, Hammond WE. Computer-based Medical Records: The Centerpiece of TMR. MD Comput. 1988;5(5):48–62.
17. Stead WW. Information management through integration of distributed resources. Bull Med Libr Assoc. 1988;76(3):242–7.
18. Stead WW, Miller RA, Musen MA, Hersh WR. Integration and beyond: panel discussion. J Am Med Inform Assoc. 2000;7:146–8.

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

19. Lindberg DAB, Humphreys BL, McCray AT. The Unified Medical Language System. Method Inf Med. 1993;32:281–91.
20. Date CJ. What Not How: The Business Rules Approach to Application Development. Reading, MA: Addison-Wesley Publishing Co., 2000.
21. Ross RG. Principles of the Business Rule Approach. Boston: Pearson Education Inc., 2003.
22. von Halle B. Business Rules Applied: Building Better Systems Using the Business Rules Approach. New York, NY: John Wiley & Sons, 2003.
23. Vaduva A, Dittrich KR. Metadata management for data warehousing: between vision and reality. In: Proceedings of the 2001 International Database Engineering and Applications Symposium. Grenoble, France: IEEE, 2001:129–35.
24. Schulz EB, Barrett JW, Price C. Read code quality assurance: from simple syntax to semantic stability. J Am Med Inform Assoc. 1998;5:337–46.
25. Cimino JJ, Clayton PD, Hripcsak G, Johnson SB. Knowledge-based approaches to the maintenance of a large controlled medical terminology. J Am Med Inform Assoc. 1994;1:35–50.
26. Nadkarni PM, Marenco L, Chen R, Skoufos E, Shepherd G, Miller P. Organization of heterogeneous scientific data using the EAV/CR representation. J Am Med Inform Assoc. 1999;6:478–93.
27. Genesereth MR. Knowledge Interchange Format Draft Proposed American National Standard (dpANS) NCITS.T2/98-004, 1998. Available at http://logic.stanford.edu/kif/dpans.html. Accessed November 28, 2003.
28. Grosof BN, Labrou Y. An approach to using XML and a rule-based content language with an agent communication language. Proceedings of the IJCAI-99 Workshop on Agent Communication Languages (ACL-99); August 1, 1999; Stockholm, Sweden.
29. Boley H. The Rule Markup Initiative. Available at: http://ruleml.org/>. Accessed Nov 28, 2003.
30. Riley G. CLIPS: A tool for building expert systems, 2003. Available at: www.ghg.net/clips/CLIPS.html. Accessed Nov 29, 2003.
31. Friedman-Hill E. JESS, The Rule Engine for the Java Platform, 2003. Available at: http://herzberg.ca.sandia.gov/jess/. Accessed Nov 29, 2003.
32. YASU Technologies. QuickRules. Net Standard Edition, 2003. Available at: http://www.yasutech.com/products/quickrulesdotnet/index_NET.htm. Accessed Nov 25, 2003.
33. Rumbaugh J, Jacobson I, Booch G. The Unified Modeling Language Reference Manual. Reading, MA: Addison-Wesley, 1999.
34. Forgy CL. Rete: a fast algorithm for the many pattern/many object pattern match problem. Artif Intell. 1982;19:17–37.
35. Levine J, Mason T, Brown D. lex & yacc. 2nd ed. Sebastopol, CA: O'Reilly Press, 1992.
36. 21 CFR Part11.com. Available at: http://www.21cfrpart11.com. Accessed July 6, 2004.
37. Shortliffe EH, Axline SG, Buchanan BG, Merigan TC, Cohen SN. An artificial intelligence program to advise physicians regarding antimicrobial therapy. Comput Biomed Res. 1973;6(6):544–60.
38. van Bemmel JH, Musen MA. Handbook of Medical Informatics. Bonn: Springer, 1997.
39. Microsoft Corporation. SQLServer Yukon, 2003. Available at: http://www.microsoft.com/sql/evaluation/yukon.asp. Accessed Nov 28, 2003.
40. Microsoft Corporation. Microsoft Developer Tools Roadmap 2004–2005, 2003. Available at: http://msdn.microsoft.com/vstudio/productinfo/roadmap.aspx. Accessed Dec 1, 2003.
41. Coenen F, Bench-Capon T. Maintenance of Knowledge-Based Systems. London: Academic Press Ltd., 1993.

Appendix 2: The Versioning Subschema

The rule engine records metadata changes in the versioning subschema. The Meta_Classes and Meta_Attributes tables are used for various purposes in TrialDB and store information respectively on every table and every field in a table in TrialDB's relational schema, thus serving the purpose of system self-documentation. These tables have numerous columns; we describe only a few that are essential to understanding of their use in versioning. The Meta_Version table records versions of a Case Report Form, Meta_Revisions segregates atomic changes by Use Case, and Meta_Field_Difference records the atomic changes themselves.

The Meta_Version Table

This table stores the explicit version of a Case Report Form, and the user who created the version or has locked the version.

VERSION_NUMBER: Determined by the user when a new version occurs.
CLUSTER_ID: the ID of the Case Report Form.
USER_ID: the ID of the user who is creating the version.
VERSION_SHORT_DESCRIPTION: a short description entered by the user.
VERSION_DESCRIPTION: a long description created by the rule engine from the revisions and reverse deltas.
DATETIME_LASTMODIFIED: self-explanatory.
DATETIME_CREATED: self-explanatory.
LOCK_OWNER: the user who has the version locked.
Primary key: VERSION_NUMBER,CLUSTER_ID.

The Meta_Revision Table

This table is used by the rule engine to store revisions that occur for a given set of actions.

REVISION_UID: Primary Key.
USER_ID: ID of the user who caused the rule to fire.
REVISION_NUMBER: created by the rule engine for each change.
VERSION_NUMBER, CLUSTER_ID: Foreign key to Meta_version table.
REVISION_DATETIME: self-explanatory.
REVISION_DESCRIPTION: self-explanatory.

The Metafield_Difference Table

This table is used to store the atomic changes into individual form elements that comprise the version. Reverse deltas that occur when changes are made to the metadata of the CRF.

REVISION_UID: Foreign key into the Meta_Revision table.
META_ATTRIBUTE_ID: The ID of the table column whose value was changed. This is a foreign key into the Meta_Attributes table.

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019

DIFFERENCE: a text that indicates the type of reverse delta difference. This is one of Added, Modified, or Deleted.

OBJECTID: Unique identifier of a row in the table whose value was changed.

FIELD_OLD_VALUE: Old value of the table column.

META_CLASS_ID: Unique identifier of the table whose attribute value is changed. This is a foreign key into the Meta_Classes table.

Example: If a user changes the data type of Question ID #6000 from string to integer, the rule engine would fire the use case "ChangeQuestionDataType." The row recording the change in this table would have the following information:

Meta_Class_ID: would point to the entry for the Questions table.

Meta_Attribute_ID: would point to the entry for the "DataType" field of the Questions table.

Object ID: The ID of the question affected (6000).

Difference: a code that indicates "Modified."

Field_Old_Value: Old value of the "DataType" field for the above question ("string").

Primary key: REVISION_UID, META_ATTRIBUTE_ID, DIFFERENCE, OBJECTID.

Meta_Classes and Meta_Attributes

The primary key of Meta_Classes is a machine-generated Class ID. For the purpose of this narrative, the only fields that are relevant are a class name (the name of the table), a caption (the name presented to the user), and a brief description of the table's purpose.

Meta_Attributes similarly has a machine-generated Attribute ID. There is a foreign key (Class_ID) into the Meta_Classes table, an attribute name (the physical column name), a caption and brief description, a data type, and a serial number indicating the physical column number in the table. Attribute_Name, Datatype and Serial_Number are obtained automatically by a query of the DBMS catalog.

Downloaded from https://academic.oup.com/jamia/article-abstract/11/5/380/819794 by guest on 23 March 2019