The vMR Data Structure

Notes about the Virtual Medical Record (vMR), its structure, and how to use it

Last Updated: 3/28/2014 7:30 PM ~~5/16/2013 8:47 AM~~

Change Tracking

Date	Content Changes	Author
2011-11-29	First Draft	David Shields
2011-12-01	Second Draft	David Shields
2012-06-14	Updates to clarify the role of the vMR in OpenCDS	David Shields
2013-05-13	Renamed to The vMR Data Structure, removing OpenCDS specific material to make the discussion more general	David Shields
2013-05-16	More clarifications	

Contents

I. Purpose of a virtual Medical Record

A virtual Medical Record (vMR) is a data structure designed specifically and solely for decision support. It is NOT intended for persistence in anything more than a processing log file, and it is NOT intended to carry complete provenance as a medical document. It is lightweight and engineered for computability in preference to human readability. It is sometimes described as a clinical document turned inside out to make the actionable data elements more prominent.

The vMR is a compromise between the polarities of complete detailed expressivity, and broad generalization. We often say that it follows the 80-20 rule, where 80% of the work is done by 20% of the elements. The vMR tries to include the 20% of the elements that do most of the work, and then provides a means of extension to specify any other element that may be needed in less common use cases.

One major goal was to create a data structure that would be stable for long periods of time. Changing data structures in CDS is an expensive process, involving changes to core software, and sometimes changes to balloted standards for data use. We tried to create a vMR that will be as useful as possible for as long as possible before new medical knowledge and practices require a new version.

Some features of the vMR are:

- It bases its data element definitions on the ISO 21090 standard, which was adopted by HL7 as Datatypes R2.
- The vMR bases its structure very loosely on relevant components of the CCD and the HL7 Clinical Statements (which are both based on the HL7 v3 Reference Information Model – the RIM),
- It structures its data elements from those standards in a way that makes the "computable" elements of those structures more accessible.
- It also tries to present some of the trickier concepts in all HL7 v3 structures in a format that is more understandable to the clinical experts who need to design the content of knowledge for clinical decision support. This includes explicitly placing the following HL7 v3 concepts into separately named classes:
    - "negationInd" (which means that the presented data is not true), and
    - "mood" (which determines whether we are talking about an event that has happened, or an event that has been proposed, or scheduled, etc.).
- It can be extended to describe clinical data that is not built into it without changing the model.
    - There is an element in the base ClinicalStatement class that allows the addition of new attributes to any ClinicalStatement by describing the element and its datatype.
    - Both Entities and ClinicalStatements can be related to each other in whatever structure is useful to fully describe complicated concepts. This allows for the expression of a wide variety of useful collections of data for clinical decision support purposes.
- It is designed in such a way that the data contained in a vMR can be flattened during processing into a set of lists. Flattened lists are a structure that software can scan much faster than it scans deeply nested data structures (such as most HL7 v3 structures, e.g., CDA and CCDA), which lends

itself to very high-performance evaluation of large datasets associated with ongoing patient care.

The implementable vMR schema provided as an informative addendum to the logical model provides a common model to write rules against. This implementable schema provides two optional structures to submit data relationships:

1. "nested" (as is typical in XML or object structures), and
2. "listed" with related ids (as is typical in relational database structures).

The external vMR schema places the location where **nested** structures are attached to the parent class within the leaf nodes, so that nested data in a clinical statement comes at the end of the enclosing data element. Optional **listed** data relationships are placed at the end of all other data for an "Evaluated Person". External vMR data may be provided in either structure, or a combination of those two structures.

The internal data structure of the vMR data within an implementation is expected to normalize all of the **nested** data into the **listed** structure, so that all data of the same class is in a single list.The relationships between data structures are also maintained in **listed** structures. This makes it possible to write rules that look for a clinical concept as a code within a certain class of data without having to walk an object tree to look for all instances of that data structure.

Note that it is up to the user of the vMR not to duplicate data between **nested** and **listed** structure. Either or both structures may be used, but an individual element must only be provided in one structure or the other.

Here is a sample vMR construction (with all details omitted), shown in both nested and listed structures so you can see the differences:

1. **Nested** (related entities and clinical statements are nested within the source clinical statement):

```
<vmr>
        <templateId root=""/>
        <patient>
                <id root=""/>
                <demographics/>
                <clinicalStatements>
                        <adverseEvents>
                                <adverseEvent>
                                        . . . (details of the adverse event)
                                        <relatedEntity>
                                                <targetRole/>
                                                <administrableSubstance>
                                                        . . .
                                                </administrableSubstance>
                                        </relatedEntity>
                                        . . . (possibly more related Enitites)
                                        <relatedClinicalStatement>
                                                <targetRelationshipToSource/>
                                                <observationResult>
                                                        . . .
                                                </observationResult>
                                        </relatedClinicalStatement>
                                        . . . (possibly more related ClinicalStatements)
```

```
                    </adverseEvent>
                </adverseEvents>
            </clinicalStatements>
        </patient>
</ vmr>
```

2. **Listed** (each clinical statement and entity is grouped and listed separately, as are their relationships – no nesting)**:**

```
<vmr>
        <templateId root=""/>
        <patient>
            <id root=""/>
            <demographics/>
            <clinicalStatements>
                <adverseEvents>
                    <adverseEvent>
                        <id root=" "/>
                        . . .
                    </adverseEvent>
                </adverseEvents>
                <observationResults>
                    <observationResult>
                        <id root=" "/>
                        . . .
                    </observationResult>
                </observationResults>
            </clinicalStatements>
            <clinicalStatementRelationships>
                <clinicalStatementRelationship>
                    <sourceId root=" "/>
                    <targetId root=" "/>
                    <targetRelationshipToSource code=" ">
                </clinicalStatementRelationship>
            </clinicalStatementRelationships>
            <clinicalStatementEntityInRoleRelationships>
                <clinicalStatementEntityInRoleRelationship>
                    <clinicalStatementId root=" "/>
                    <entityId root=" "/>
                    <role code=" "/>
                </clinicalStatementEntityInRoleRelationship>
            </clinicalStatementEntityInRoleRelationships>
            <entityLists>
                <administrableSubstances>
                    <administrableSubstance>
                        <id root=" "/>
                        . . .
                    </administrableSubstance>
                </administrableSubstances>
            </entityLists>
        </patient>
</vmr>
```

As you can see, the **nested** structure is more compact and easier for humans to follow, since related data is all in one place. Different types of ClinicalStatements might be found as relatedClinicalStatements anywhere in the structure.

On the other hand, the **listed** structure is flatter and more amenable to analysis by writing rules. For example, all ClinicalStatements of the same type are in the same list, where a rule can iterate over them to locate data that is of interest.

II. The Internal vMR Structure in OpenCDS

Datatypes

Internal datatypes began as a clone

We started the definition of the internal datatypes as a clone of the datatypes used in the logical model and the implementable schema. Some changes were then made to all of the datatype classes to turn them into Java Beans useable in Drools and OpenCDS:

> All of the generated comments from JaxB were removed (except for the description of the class), along with the imported JaxB classes, and all methods of the generated code.
>
> Using Eclipse, we added the following:
>
> - Getters and setters
> - Hash code and equals methods
> - toString() method (including the super class, where one exists)
> - Apache License as comments at the top of the class.

Some internal datatypes are modified

Some internal datatypes were modified or optimized to suit the way we needed to use them. Specifically, the following were changed:

- "CD" datatype (ConceptDescriptor) structure was changed from its Datatypes R2 structure to move the nested CD element **displayName** to be an attribute of the CD. This is done for ease of use in writing rules and producing outputs.
- "II" datatype (InstanceIdentifier) structure was changed from the external root and extension structure to a single string, expected to be of the format "root^extension" which makes it possible to use it in rules as a simple unique identifier. Note that instances of II that do not have an extension (such as a UID) are implemented as a string just consisting of "root".
- "TS" datatype (TimeString) was changed to java.util.Date structure for internal use. For computing purposes, we assume that all HL7 datetime strings have at least the year, month, and day elements populated.

Mapping Utility

There is a MappingUtility which performs mapping between internal and external datatypes. Names of mapping methods are structured as "{DT}2Internal{DT}" or "Internal{DT}2{DT}" where {DT} represents the name of the datatype.

vMR Classes

All VMR classes defined in the external schema are expected to be present in the internal structure except for the following:

- RelatedEntity
- RelatedClinicalStatement
- RelatedEntityInRole

Data in the above three classes is normalized and split into two parts. The first one contains (and exactly matches) the original Entity or ClinicalStatement contained in one of the Related classes, and is moved into the corresponding ClinicalStatement or EntityList. The second part contains the nature of the relationship (typically its role in relationship to the parent class), and the ID of the source and the target. These relationship classes are moved from the nested source structure into the following corresponding internal classes in a list structure:

- EntityRelationship
- ClinicalStatementRelationship
- ClinicalStatementEntityInRoleRelationship

NOTE: The vMR classes all get the same treatment as the datatypes, but they also have an associated "Mapper" class which implements pullIn() and pushOut() methods.

We also define a few additional internal classes that are not part of the vMR to enhance the writing of rules. The most important of these additional internal classes are the "concept" classes. Every data element in the vMR that is a **concept descriptor** (i.e., the datatype is "**CD**"), or a **templateId** has an associated internal concept class.

For example, ObservationResult data has an **observationFocus** data element. This element has an associated concept class named **ObservationFocusConcept**, and OpenCDS populates this class with all OpenCDS concepts which map to the actual data present in the element. An OpenCDS concept might include things like "microbiology lab result", "discharge diagnosis", "Blood Pressure", and "Chlamydia Trachomatis organism". OpenCDS might create those concepts by mapping a submitted code for a data element to the OpenCDS concepts. These concept mappings may be maintained in Apelon DTS, the CDS Administration Tool (CAT) furnished with OpenCDS, or other vocabulary maintenance systems.

By the way, there is nothing special about an **OpenCDS concept**. It is simply a concept which someone writing a clinical rule determines to be useful for expressing the elements involved in the logic of the rule. It is created in a terminology system by mapping a list of standard (or proprietary) codes (aka **concept descriptors**, or **CD** data) to a single **concept descriptor** for the OpenCDS concept. This means that we have a single reference that we can use to write a rule about any of the codes in the source data that match the desired concept. It separates the writing of the logic of the rule from the maintenance of

the data codes that are constantly evolving. The rules don't need to change when codes are added or deprecated. Mappings don't necessarily need to change to adjust the logic of a rule.

III. Mapping *from* External Structure

Interfaces: org.opencds.vmr.v1_0.mappings.in

IPayloadUnmarshaller.java

The implementation of this interface accepts the payload from the DSS message as a base64 string, and converts it into a structured JaxB element for the CDSInput. This class then calls an implementation of the IPayloadUnmarshaller interface to produce the internal VMR structure for processing by an inferencing engine.

IBuildFactLists.java

The implementation of this interface accepts the JaxB CDSInput element as produced by the implementation of the IPayloadUnmarshaller interface, and parses it into the internal VMR structure as lists of facts. This is where the heavy lifting is done to normalize the submitted data to be processed by an inferencing engine.

We provide an implementation to support Drools fact lists. Users interfacing to a different inferencing engine requiring a different internal structure, but still based on the VMR, can replace this module as required.

IV. Mapping *to* External Structure

Interfaces: org.opencds.vmr.v1_0.mappings.out

IBuildResultSet.java

The implementation of this interface accepts the processed fact lists from the inferencing engine, and produces a structured output in the CDSOutput schema structure. This structure is then further processed by the implementation of the IMappingOutbound interface.

IMappingOutbound.java

The implementation of this interface accepts the CDSOutput structure produced by the implementation of the IBuildResultSet interface, and produces a base64 string for return as a response by the DSS service.

V. Using the vMR in OpenCDS

This is intended to be an overview of the general organization of the vMR, and a big-picture guideline to how you might use it. First, here is a somewhat organized view of the structure of the VMR, which will be followed by some suggestions on how to use it in OpenCDS, both for input, and for output.

Abstract Base Classes

There are two groups of data structures in the vMR: Clinical Statements describe "activities" of interest, and Entities are the people, places, and "things" that have a role in those medical activities. These abstract classes cannot be separately instantiated, but you can refer to them in rules. For example, you could examine the common elements in a base class to draw broad conclusions about whether a particular activity happened, did NOT happen, was proposed, or was ordered.

ClinicalStatement

We have distilled medical activities into 9 groups, and there is a "base" class for each of those groups. ClinicalStatement is itself a "base" class for all of the following specialized groups, and contains the elements that are in common with all of them. More details about the elements in these classes are in the detailed vMR documentation.

- AdverseEventBase
- CommunicationBase
- EncounterBase
- GoalBase
- ObservationBase
- ProblemBase
- ProcedureBase
- SubstanceAdministrationBase
- SupplyBase

EntityBase

This contains the common elements for all persons, places, organizations, and other "things." It is the basis for both EvaluatedPerson, and for the 7 other groups of entities.

Instantiatable Classes

The classes listed here are not abstract. You can instantiate them to send data to OpenCDS, and you can instantiate them in rules to return inferred data from OpenCDS to your application.

EvaluatedPerson

This is a root class for the vMR. All ClinicalStatements and related Entities are related to an EvaluatedPerson (most commonly known as the "Patient"). It is also possible for the vMR to contain

OtherEvaluatedPersons related to the patient, in the cases of a family medical history, or infectious disease contacts, or other similar use cases. These OtherEvaluatedPersons have two attributes: 1.) their medical data is relevant to the Patient, and 2.) they may have any sort of data about health in the same way as the Patient.

Nevertheless, there is always one person, the Patient, who is the focus of the vMR. The rules written in OpenCDS are intended to assist in the medical care of the Patient.

While an EvaluatedPerson inherits from the EntityBase class, it is treated differently than all other "lesser" entities. The Patient is the subject of the vMR, and OtherEvaluatedPersons have a clinical relationship to the Patient, whereas other entities participate in healthcare activities related to the Patient.

Entity Classes

These are the "lesser" entities that play a role in healthcare activities. They may be anything from medications to medical providers to lab specimens to healthcare protocols.

- AdministrableSubstance (dressings, medications, nutritional supplements, etc.)
- Entity (any "thing" not otherwise listed here)
- Facility
- Organization
- Person (other than an evaluated person, i.e., a person whose health has not relationship to this vMR)
- Specimen

Event Classes

These classes are used to describe a healthcare activity that has already happened, or is currently in existence.

- AdverseEvent
- CommunicationEvent
- EncounterEvent
- Goal
- ObservationResult
- Problem
- ProcedureEvent
- SubstanceAdministrationEvent
- SubstanceDispensationEvent
- SupplyEvent

Denied Event Classes

These classes describe a healthcare activity that did NOT happen. For example, an "UnconductedObservation" of blood pressure means that the BP was not taken (and possibly should have been). A "DeniedProblem" is a statement that the patient does NOT have the described problem or condition.

- DeniedAdverseEvent
- UnperformedCommunication
- MissedAppointment
- UnconductedObservation
- DeniedProblem
- UndeliveredProcedure
- UndeliveredSubstanceAdministration
- UndeliveredSupply

Proposal Classes

Proposals are most commonly produced as output from OpenCDS (e.g., "patient needs an MMR", or an HbA1c test, etc., but they might also represent input. For example, there might be a proposal to give a patient a certain medication, and OpenCDS might do a drug-drug interaction evaluation with the current medications taken by the patient.

A proposal is less authoritative than a request or an order, and is meant to convey information to be considered by a medical professional.

- AppointmentProposal
- CommunicationProposal
- GoalProposal
- ObservationProposal
- ProcedureProposal
- SubstanceAdministrationProposal
- SupplyProposal

Request and Order Classes

Requests and Orders are generally created by a medical professional to address a medical condition or problem. They are more authoritative than a proposal, and they convey a medical request for something to be done, generally directed to an ancillary employee.

- AppointmentRequest
- CommunicationOrder
- ObservationOrder
- ProcedureOrder

- SubstanceAdministrationOrder
- SupplyOrder

Populating Data for submission to OpenCDS

Choose your structure

The vMR is very flexible, and there is more than one way to do just about anything. In general, it is probably a good idea to choose the general structural approach that most closely matches the structure of your data.

If your data is in an XML or Object structure, then the nested structure of the vMR might more closely fit with your input data.

On the other hand, if you are pulling data from a relational database, then the listed structure might work better for you.

However, in most cases, if you are not the first person to populate a vMR for a particular purpose, you will want to do it in exactly the same way everyone else does it, so that you can use the same rules, and that requires using a template.

Use a Template

A "template" is a collection of constraints and requirements on the input data for a particular purpose. Since there is more than one way to represent just about anything, it is good to create and share templates for a particular purpose, so that rules can be shared.

OpenCDS will be collecting and publishing templates as time goes on. If a template doesn't exist for your particular need, then write down the requirements as you develop your rules, and we will work with you to create a template for that need.

Once a template is documented in OpenCDS, then we will assign a "templateID" value to it, and that templateID can be referenced within the rules to ensure that the data submitted adheres to the requirements of the author of the rule.

Note that the same template can apply to data submitted in the object / xml structure, or the listed / relational structure, because the template is addressing the logical structure of the data, regardless of how you physically structure it.

Use Best Practices

We will be collecting "Best Practices" for representing various common healthcare data structures, and the elements within those data structures. We will also try to do something similar for authoring rules.

These "best practices" will be posted in a separate document.

Populating Output Data in Rules for Return from OpenCDS

OpenCDS Output Structure

A portion of the output structure of OpenCDS is pre-defined, but there is still more than one way to create output. There are two major choices: 1.) single result element; or 2.) CDSOutput result containing vMR structured data.

Single Element Result

If you choose this type of result, your output data is a single element, using any of the supported ISO 21090 datatypes. For example, you might return a Boolean "true", or you might return an integer value of "23".

CDSOutput Containing a vMR Structured Output Result

If you choose this type of result, your output will resemble the structure of the input you sent to OpenCDS. It always contains all of the patient demographic data that was submitted, and the rest of the content can be created by the rules. In some cases, it might even resemble the input very closely. In fact, we provide a simple rule with OpenCDS called the "Bounce" rule that simply copies all of the input data to the output data structure. It is provided primarily for testing purposes, and to demonstrate how to copy input data to the output data structure.

Some useful output patterns are:

1. ObservationResults with information inferred from the input data, e.g., "patient has a reportable disease of <whatever>", or "patient has medications which can interact with each other" etc.
2. Treatment proposals, such as ProcedureProposals, AppointmentProposals, SubstanceAdministrationProposals.
3. Goal proposals
4. Care plans, combining any of the above outputs into an integrated package.

Use a Template for Output

The use of a template for the output serves most of the same purposes as a template for input. It allows the rules to be shared, and tells everyone who wants to use the rule what the output will look like.

Use Best Practices for Creating Output

We will be collecting "Best Practices" for representing various common output data structures, and the elements within those data structures.

These "best practices" will be posted in a separate document.