Getting Started with Clinical Quality Language: Technical Implementation for Vendors

Bryn Rhodes, ESAC
James Bradley, The MITRE Corporation

March 2018
Agenda

▪ Clinical Quality Language (CQL) Architecture
▪ Creating a Calculation Engine
▪ Using a CQL Execution Engine
▪ Lessons Learned
▪ Resources
▪ Questions
Assumptions

- Familiarity with CQL
- Background in language processing
  - Language translation and/or evaluation
- Familiarity with Clinical Data Representation
  - Clinical Data Models
  - Terminology
CQL Architecture
Components of Sharing Logic

Definitions:
SNOMED CT – Systematized Nomenclature of Medicine – Clinical Terms
LOINC – Logical Observation Identifiers Names and Codes
CQL Architecture

Definitions:
SQL – Structured query language

Authors use CQL to produce libraries containing human-readable yet precise logic.

ELM XML documents contain machine-friendly rendering of the CQL logic. This is the intended mechanism for distribution of libraries.

Implementation environments will either directly execute the ELM, or perform translation from ELM to their target environment language.
CQL-to-ELM Translation

Conceptual Level

CQL is defined at this level

Lexical Analysis

\[ x + y \times z; \]

Logical Level

Processing applications begin at this level

Parsing

\[ x \oplus \ y \ominus \ z \]

Semantic Analysis

\[ + (\text{int, int}) \]

\[ \times (\text{int, int}) \]

\[ \text{symbol}(z) \]

Physical Level

Compiling/Translation

001100...
Expression Logical Model (ELM)

- A “byte-code” representation of CQL logic: carries sufficient semantics to enable execution independent of the CQL that produced it
- A “canonical” representation in terms of more primitive operations: focused on supporting implementation use cases such as evaluation and translation
ELM Representation

- ELM expressions are built as trees of nodes, where each kind of expression is represented by a different node type.

- For example, \(2 + 2\) is represented as:

```
      Add
     /   \
   Literal (2)  Literal (2)
```
In general, operations and functions in CQL have an equivalent ELM representation

<table>
<thead>
<tr>
<th>CQL Operator or Function</th>
<th>ELM Node Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>and</td>
<td>And</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>Ceiling()</td>
<td>Ceiling</td>
</tr>
</tbody>
</table>
Type Categories

- **Primitive types**
  - Boolean
  - String
  - Integer
  - Decimal
  - DateTime
  - Time

- **Collection types**
  - List<T>

- **Structured types**
  - Class types (defined by a data model)
  - Tuple (anonymous class types)

- **Interval types**
  - Interval<T> (must be an ordered type)
Data Access

- All data access is done through Retrieve
  - Type information (data type and optional “template” identifier)
  - Code filter (a valueset or a set of codes)
  - Date filter (a date range)
  - Path information (id, code, date)
**Simple Retrieve**

- Pharyngitis Diagnoses:

  ```json
  {"Diagnosis": "Acute Pharyngitis"}
  ```

- ELM Retrieve:
Specifying Data Models

- Each data model is described with “model info”
- Describes the types available in the model
- Also defines “primary code path” for each retrievable type
- Specifies the “patient” type

**NOTE:** Model info is not required by ELM, it’s only required to translate CQL to ELM
Model Info Example

```xml
<ns4:typInfo xsi:type="ns4:ClassInfo"
  name="QDM.Diagnosis"
  identifier="Diagnosis"
  label="Diagnosis"
  retrievable="true"
  primaryCodePath="code"
  baseType="QDM.QDMBaseType">
  <ns4:element name="onsetDateTime" type="System.DateTime"/>
  <ns4:element name="abatementDateTime" type="System.DateTime"/>
  <ns4:element name="anatomicalLocationSite" type="System.Concept"/>
  <ns4:element name="severity" type="System.Concept"/>
</ns4:typInfo>
```
System Model

- System.Any – Base type for all types
- System.Boolean
- System.Integer
- System.Decimal
- System.String
- System.DateTime
- System.Time
- System.Quantity – e.g., 3 'gm'
- System.Code – code, system, version, display
- System.Concept – codes, display
CQL Library

- Named, versioned groupings of CQL components

```
library CMS55 version '1'
using QDM

valueset "Inpatient": '2.16.840.1.113883.3.666.5.307'

parameter "Measurement Period" default Interval[@2014-01-01T00:00:00.0, @2015-01-01T00:00:00.0]

context Patient

define "Inpatient Encounters":
  ["Encounter, Performed": "Inpatient"] &
  where E.lengthOfStay <= 120 days
  and E.dischargeDateTime during "Measurement Period"
```
Library Example

```xml
<library xmlns="urn:hl7-org:elm:r1"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:t="urn:hl7-org:elm-types:r1"
         xmlns:qdm="urn:healthit-gov:qdm:v4_2">
  <identifier id="CMS55" version="1"/>
  <schemaldentifier id="urn:hl7-org:elm" version="r1"/>
  <usings>
    <def localidentifier="System" uri="urn:hl7-org:elm-types:r1"/>
    <def localidentifier="QDM" uri="urn:healthit-gov:qdm:v4_2"/>
  </usings>
  <parameters>
    <def name="Measurement Period" accessLevel="Public"/>
  </parameters>
  <valueSets>
    <def name="Inpatient" id="2.16.840.1.113883.3.666.5.307" accessLevel="Public"/>
  </valueSets>
  <statements>
    <def name="Patient" context="Patient"/>
    <def name="Inpatient Encounters" context="Patient" accessLevel="Public"/>
  </statements>
</library>
```
Parameter Definition

```xml
<def name="Measurement Period" accessLevel="Public">
    <default lowClosed="true" highClosed="false" xsi:type="Interval">
        <low xsi:type="DateTime">
            <year valueType="t:Integer" value="2014" xsi:type="Literal"/>
            <month valueType="t:Integer" value="1" xsi:type="Literal"/>
            <day valueType="t:Integer" value="1" xsi:type="Literal"/>
            <hour valueType="t:Integer" value="0" xsi:type="Literal"/>
            <minute valueType="t:Integer" value="0" xsi:type="Literal"/>
            <second valueType="t:Integer" value="0" xsi:type="Literal"/>
            <millisecond valueType="t:Integer" value="0" xsi:type="Literal"/>
        </low>
        <high xsi:type="DateTime">
            <year valueType="t:Integer" value="2015" xsi:type="Literal"/>
            <month valueType="t:Integer" value="1" xsi:type="Literal"/>
            <day valueType="t:Integer" value="1" xsi:type="Literal"/>
            <hour valueType="t:Integer" value="0" xsi:type="Literal"/>
            <minute valueType="t:Integer" value="0" xsi:type="Literal"/>
            <second valueType="t:Integer" value="0" xsi:type="Literal"/>
            <millisecond valueType="t:Integer" value="0" xsi:type="Literal"/>
        </high>
    </default>
</def>
```
Patient Context

```
10 11 12
context Patient

<def name="Patient" context="Patient">
  <expression xsi:type="SingletonFrom">
    <operand dataType="qdm:Patient" templateId="Patient" xsi:type="Retrieve"/>
  </expression>
</def>
```
Expression Example

define "Inpatient Encounters":
  ["Encounter, Performed": "Inpatient"] E
  where E.lengthOfStay <= 120 days
  and E.dischargeDatetime during "Measurement Period"

<def name="Inpatient Encounters" context="Patient" accessLevel="Public">
  <expression xsi:type="Query">
    <source alias="E">
      <expression dataType="qdm:EncounterPerformed" templateId="EncounterPerformed" codeProperty="code" xsi:type="Retrieve">
        <codes name="Inpatient" xsi:type="ValueSetRef"/>
      </expression>
    </source>
    <where xsi:type="And">
      <operand xsi:type="LessOrEqual">
        <operand path="lengthOfStay" scope="E" xsi:type="Property"/>
        <operand value="120" unit="days" xsi:type="Quantity"/>
      </operand>
      <operand xsi:type="In">
        <operand path="dischargeDatetime" scope="E" xsi:type="Property"/>
        <operand name="Measurement Period" xsi:type="ParameterRef"/>
      </operand>
    </where>
  </expression>
</def>
Creating a Calculation Engine
Evaluation Approaches

Clinical Quality Language (CQL)

Expression Logical Model (ELM)

Native | Java Script | Drools | SQL

Authors use CQL to produce libraries containing human-readable yet precise logic.

ELM XML documents contain machine-friendly rendering of the CQL logic. This is the intended mechanism for distribution of libraries.

Implementation environments will either directly execute the ELM, or perform translation from ELM to their target environment language.
CQL Evaluation Architecture

Calculation of CQL involves the following conceptual components:

- **Calculation Engine** is what performs the measure calculations.
- **Logic** is the description of how the measure calculates against the clinical information, for example, patient records.
- **Model** is the structured representation of clinical information that is used to calculate the measure.
- **Data Access** is how records of clinical information are retrieved from the underlying system, for example, an Electronic Health Record (EHR).
- **Terminology** is concerned with determining whether clinical information is related to the measure logic through looking at coded values.
- **Libraries** enable the reuse of measure logic across measures and decision support artifacts.
Java-Based CQL Evaluation

**Calculation Engine** – Generic Java implementation

**Logic** – is implemented as a deserialization of the ELM into a tree of “executable” nodes

**Model** – The model for this generic engine is effectively any Java classes, supported with a Runtime to model the “System” types

**Data Access** – Modeled with a “DataProvider” interface that corresponds to the “retrieve” in ELM

**Terminology** is – Modeled with a “TerminologyProvider” interface that provides basic terminology services (expand, isMember, lookup)

**Libraries** – Modeled with a LibraryLoader interface
FHIR-CQL Execution

- Uses HAPI FHIR classes for the Model Implementation
- FHIRDataProvider to communicate with a FHIR data endpoint
- FHIRTerminologyProvider to use a FHIR terminology endpoint
HAPI FHIR JPA Plugin

- Uses a FHIRJPAPProvider to “loop back” into the HAPI FHIR Server
- Provides a FHIRLibraryLoader that can load libraries from the HAPI FHIR Server
JavaScript-Based CQL Evaluation

Data Types – Built-in clinical models for this engine

Patient Source – An interface class provided on execution, that implements "get" functions to access data from the passed in patient model

Code Service – Interface that provides basic terminology services (expand, isMember, lookup)

Repository – Loads multiple libraries in for execution
Using a CQL Execution Engine
Introduction

- Software Developer on the Bonnie eCQM Testing Tool
- Replaced existing QDM-Logic Execution Engine with CQL Execution Engine
- Contributed to the JavaScript CQL Execution Engine
Open Source JavaScript Execution Engine

- Easily run Server-side or Client-side
- Developed in CoffeeScript. Transpiled to JavaScript
- Only uses data passed in at execution
- Expects patients to be QDM-based
  - Defined in the QDM ModelInfo files
- Returns a list of patients and their calculation values for each named expression
Execution Engine Requirements and Example

**Requirements:**

- Measurement Period
- ELM JSON
- Value Sets
- Patient Source

```java
parameters = {
    "MeasurementPeriod": new cql.Interval(
        new cql.Date_time(2016, 1, 1, 0, 0, 0, 0),
        new cql.Date_time(2017, 1, 1, 0, 0, 0, 0)
    ),
}
lib = new cql.Library(measureElm)
codeService = new cql.CodeService(valueSets)
exector = new cql.Executor(lib, codeService, parameters)
results = executor.exec(patientSource)
```
Execution Engine Results

- All definition expressions passed in will be executed
- Results return 2 objects:
  - A list of Patients and their calculation values for each named definition expression
  - A list of Patients and their calculation values for each sub-component of a definition

```
define "Initial Population":
  "In Demographics" and exists "Qualifying Encounters"
```
Execution Engine Results (cont.)

- Does not calculate final population results
  - Definitions of the Populations are specified in the HQMF file
  - Ex: IPP, DENOM, NUMER…

- Functions in CQL are not executed
  - To execute, call in a define statement
  - Measure Population Observations are functions in CQL
    - Not included in define statements by default
Bonnie Execution Engine Workflow

**Engine Parameters:**
- ELM (JSON)
- Patient Source (JSON)
- Value Set Code Service
- Measurement Period

**Execution Engine**

**Result Object:**
- Named definition
  - Patient Results
- Sub-component
  - Patient Results

**Match Results to Populations:**
- HQMF Definitions
- Patient Result Map
Wrapping Patient Data Model to match ModellInfo QDM

```xml
<ns4:typeInfo xsi:schemaLocation="ns4:ClassInfo" name="QDM.MedicationAdministered">
  <ns4:element name="authorDateTime" type="System.DateTime"/>
  <ns4:element name="relevantPeriod" type="interval&lt;System.Interval"/>
  <ns4:element name="dosage" type="System.Quantity"/>
  <ns4:element name="supply" type="System.Quantity"/>
  <ns4:element name="frequency" type="System.Code"/>
  <ns4:element name="route" type="System.Code"/>
  <ns4:element name="reason" type="System.Code"/>
  <ns4:element name="negationRationale" type="System.Code"/>
</ns4:typeInfo>
```

```java
class CQL_QDM.MedicationAdministered extends CQL_QDM.QDMDatatype
{
  @param {Object} entry - the HDS data criteria object to convert
  constructor: (entry) ->
    super @entry
    @authorDateTime = CQL_QDM.Helpers.convertDateTime(@entry.start_time)
    @dosage = @entry.dose
    @frequency = @entry.frequency
    @negationRationale = @entry.negationReason

  @returns {Date}
  authorDateTime: ->
    @authorDateTime
}
```
The Measure Authoring Tool (MAT) Package

- Human readable .html page
- HQMF .xml file
- Measure .cql file
- Support Library .cql files
- ELM files in XML representation
- ELM files in JSON representation
Processing the MAT Package

The MAT Package

- HQMF XML

HQMF Parser

- Data Criteria
- Measure population definition mappings
- Observations

ELM Value Set Parser

- CQL FILES
- ELM JSON

VSAC API

- Value Set OIDS
- Value Sets

Bonnie Database

- ELM JSON

MAT Export Package: The HQMF File

HQMF Document Structure

HQMF Header
- HQMF Document Attributes
- Author
- Measure Processing Metadata
- Measure Period

HQMF Body
- Measure Description
- Data Criteria
- Population Criteria
  - Stratifier Criteria Entry
- Measure Observations

MAT Export Package: The HQMF File

Using HQMF to Map Populations to Define Statements

def "Exclusions":
  exists ( ["Procedure, Performed": "Chemotherapy"] Chemotherapy
      where Chemotherapy.relevantPeriod overlaps "Measurement Period"
  )
  or exists ( ["Procedure, Performed": "Bone Marrow Transplant"] BMT
      where BMT.relevantPeriod overlaps "Measurement Period"
  )
  or exists ( ["Diagnosis": "Anaphylactic Reaction to Pneumococcal Conjugate Vaccine"] Reaction
      where Reaction.prevalencePeriod before end of "Measurement Period"
  )
  or exists ( ["hospice or using hospice services"]

Library name
Definition name
Filtering Execution Engine Results

- localIdPatientResultsMap: {5a74ad8008fa18119b3566c2: {...}}
  - patientResults:
    - 5a74ad8008fa18119b3566c2:
      - Denominator: true
      - Denominator Exceptions: false
      - Initial Population: false
      - Most Recent Delivery: ProcedurePerformed {entry: {...}, _codes: {...}}
      - Most Recent Delivery Overlaps Diagnosis: [ProcedurePerformed]
  - Numerator: true
    - Patient: {birthDatetime: DateTime, gender: "M"}
    - SDE Ethnicity: []
    - SDE Payer: []
    - SDE Race: []
    - SDE Sex: [PatientCharacteristicSex]
      - __proto__: Object
      - __proto__: Object
    - populationResults: {}
Bonnie Execution Engine Workflow Recap

**Engine Parameters:**
- ELM (JSON)
- Patient Source (JSON)
- Value Set Code Service
- Measurement Period

**Execution Engine**

**Result Object:**
- Named definition
  - Patient Results
- Sub-component
  - Patient Results

**Match Results to Populations:**
- HQMF Definitions
- Patient Result Map
Lessons Learned
Lessons Learned

- Engine Architecture
- “Direct” deserialization
- DataProvider interface
- TerminologyProvider interface
- Data Access Layer and “context”
- Patient Model
- Function Execution
Lessons Learned: Engine Architecture

- **Problem**
  - How can we quickly **build a performant and flexible native evaluation engine**?

- **Success**
  - By **“direct” deserialization** of an ELM document **into a graph of “executable” nodes**

- **Discussion**
  - Considered a **“visitor” pattern**, but that would mean **run-time determination of execution**. For the number of node types in ELM, that would be **significant**
  - Considered **transpiling to Java or JavaScript**, but would require **significant Runtime libraries and significant effort**. Would result in **better performance** though

- **Recommendation**
  - **Performance** using this approach is **favorable**, but also **benefits from caching**
  - There are **some places** in ELM that **could provide better typing** (STU comments in progress)
Lessons Learned: “Direct” Deserialization

- **Problem**
  - Using “direct” deserialization, we want to be able to use generated classes, but also provide override implementations of the “execute” method

- **Success**
  - Generate all classes as descendants of a base “Executable” class
  - Read ELM directly into descendants of generated ELM classes
    - Provide engine implementation

- **Discussion**
  - Requires a JAXB override to tell the factory what class to use for each node type in ELM
  - Create an ObjectFactoryEx descended from generated ObjectFactory
    - Override “create” methods to return descendant classes
  - Set the ObjectFactoryEx in the JAXB context
Lessons Learned: DataProvider Interface

- **Problem**
  - How do we implement an engine that can **deal with arbitrary data models** as required by the CQL specification?

- **Success**
  - Define a **DataProvider interface** that completely **characterizes the data access layer**:
    - The DataProvider Interface is **based on the ELM Retrieve**
      - Context – What patient is being evaluated
      - DataType – What data type is being requested (and profile)
      - Code Filter – What code filter to apply
      - Data Filter – What date-range to apply
    - Results in a **clean interface for data access**
Lessons Learned: DataProvider Interface (cont.)

- **Considerations**
  - **Requires an abstraction layer between it and the data source** (e.g., Object Adapter)
    - i.e., an interface through which the engine can perform all get/set functionality
  - The **engine must be able to uniquely determine the DataProvider** responsible for any given instance at run-time
Lessons Learned: TerminologyProvider Interface

- **Problem**
  - How can we **enable the engine to use any terminology service?**

- **Success**
  - Define a very simple **Terminology interface**
    - In – Is a given code a member of a given value set
    - Expand – Retrieve the codes in a given value set
    - Lookup – Retrieve the code for a given code and code system
  - Based on the FHIR Terminology Service CapabilityStatement

- **Discussion**
  - **Valuesets should be cached locally for performance**
    - Implications for maintenance
Lessons Learned: Data Access Layer and “context”

Problem

- How do we implement Patient context as required by the CQL specification?
- CQL Context: all expressions in a given context are evaluated with respect to that context
  - e.g., Medications returns only medications for a given patient

Success

- The Data Access Layer is responsible for enforcing this expectation
  - FHIR is straightforward: translates to a “patient” or “subject” parameter in the URL
  - Other models: the data access layer needs to understand the relationship between each data type and the patient
    - ModelInfo doesn’t currently represent this. May need to extend ModelInfo to capture this information
Lessons Learned: Patient Model

- **Problem**
  - The execution engine expects a patient model aligned with the QDM model specified in the ModellInfo file (a calculable representation of the expected model).

- **Success**
  - Created a wrapper library around the Bonnie data model to map it to the specified QDM Model

- **Recommendation**
  - Build a wrapper library around the application’s data model to map it to the specified QDM Model
Lessons Learned: Function Execution

▪ Problem
  – Functions are not automatically executed by the execution engine
  – Measure Population Observations are represented as functions
  – Bonnie needed the Measure Population Observations to execute automatically

▪ Solution
  – Created a CQL definition snippet that called the Measure Population Observation to ensure it gets executed by the execution engine

▪ Recommendation
  – If it’s necessary to have a CQL function execute automatically, wrap it in a CQL definition
Resources
Resources

- CQL Specification - CQL Release 1, Standard for Trial Use (STU) 2
- eCQI Resource Center
  - CQL Space, including the QDM v5.3 and v5.3 Annotated
    - [https://ecqi.healthit.gov/cqI](https://ecqi.healthit.gov/cqI)
  - Check the eCQI Resource Center Events page and CQL Educational Resources page for more information
    - [https://ecqi.healthit.gov/ecqi/ecqi-events](https://ecqi.healthit.gov/ecqi/ecqi-events)
    - [https://ecqi.healthit.gov/cql/cql-educational-resources](https://ecqi.healthit.gov/cql/cql-educational-resources)
- CQL Formatting and Usage Wiki
- CQL GitHub Tools Repository
  - [https://github.com/cqframework/clinical_quality_language](https://github.com/cqframework/clinical_quality_language)
- Jira for tracking CQL Related Issues
  - [https://oncprojecttracking.healthit.gov/support/projects/CQLIT](https://oncprojecttracking.healthit.gov/support/projects/CQLIT)
Resources (cont.)

- JavaScript Open Source execution engine:
- Java Open Source execution engine:
  - https://github.com/DBCG/cql_engine
- HQMF Parser:
  - https://github.com/projecttacoma/cqm-parsers
- QDM ModelInfo Files
- Measure Authoring Tool
  - https://www.emeasuretool.cms.gov/
- Bonnie Testing Tool
  - https://bonnie.healthit.gov/
Other Related Presentations at HIMSS 2018

<table>
<thead>
<tr>
<th>Bonnie: Tips for Debugging Clinical Quality Language eCQMs</th>
<th>Cypress: New Features, Clinical Quality Language Support, and Burden Reduction</th>
<th>CMS eCQM Strategy: The Future of eCQMs (Gallery Walk)</th>
<th>CMS eCQM Strategy: The Future of eCQMs (Gallery Walk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/7/18</td>
<td>3/7/18</td>
<td>3/7/18</td>
<td>3/8/18</td>
</tr>
<tr>
<td>9:30-10am PT</td>
<td>10-10:30am PT</td>
<td>2-3:30pm PT</td>
<td>10:30am-12pm PT</td>
</tr>
<tr>
<td>Zeno 4608, MITRE Government Relations Room</td>
<td>Zeno 4608, MITRE Government Relations Room</td>
<td>Zeno 4608, MITRE Government Relations Room</td>
<td>Zeno 4608, MITRE Government Relations Room</td>
</tr>
</tbody>
</table>
Follow up

To download a copy of this presentation, visit:

https://health.mitre.org/himss18

Data Interoperability to Reduce Clinician Burden

Follow us on social media: @MITREhealth
Questions?