GELLO vs OCL

Overview

Why would we choose to use the GELLO language instead of using OCL directly?

While GELLO has originated from OCL and contains a large part of the syntax of OCL expression, there are a number of good reasons for defining GELLO as a language independent of OCL. Some differences are subtle, whilst others are more obvious.

- **GELLO is designed as a query language, whilst OCL is designed as a declarative language.**

The ultimate goal of GELLO is to be able to take a data model, apply a query to that model and to normally return a result to an external infrastructure, typically in an embedded manner. In this respect it functions in a simlar way to SQL, but has much richer operations than SQL in that it is fully object oriented, and could possibly be a future successor to SQL.

Conversely, OCL has been designed as a language to construct data models using a model driven architecture. It is typically used to generate a constrained data model, but any rules it generates would be hard coded into the data model being generated. While it has query facilities through the use of the body: clause of a Context statement, these only apply to the internal data model being described - there is no generic mechanism to pass the result of a query outside the context of a given class or function call.

Compare these two pieces of code to retrieve all sodium (Na) observations for a given patient context.

in GELLO

```
Context SinglePatient

  Observations->select(obs | obs.code.value = 'Na')
```

in OCL

```
Context SinglePatient::GetSodiumObservations()

 body: Observations->select(obs | obs.code.value = 'Na')
```

Now at a cursory glance, these two snippets of code appear to be much the same. However the difficulty is that in OCL, we have to define an arbitrary new method and bind it to our data model to be able to achieve the desired result, and we also need to specifically call this method from our host infrastructure. This might make sense when OCL is being used in the context of an existing externally defined data model which is hard wired into an application framework, but this becomes somewhat unworkable in a typical query oriented environment where arbitrary queries need to be made on a pre-existing data model.

The piece of code written in GELLO is more portable, makes no change to the data model, and is clear in its intent.

- **GELLO has specific needs which OCL has difficulty dealing with.**

Important additions to GELLO are the ability to create instances of classes with properties set to specific values. This is done through the use of the factory construct.

Let us extend our example to also select only observations where the observation value is greater than a specific value. We assume that PQ in the data model has the operator ">" implemented.

in GELLO

```
Context SinglePatient
  Let NaMax: PQ = factory.PQ(145.0,'mmol/L')
  Observations->select(obs | (obs.code.value = 'Na') and (obs.value >
NaMax))
```

in OCL we can't do this easily. The best we can do is compare the individual properties of the PQ we wish to test, but we can't specifically use the predefined operators of PQ or any other methods of PQ.

```
Context SinglePatient::GetSodiumObservationsGreater()

 body:
    Let NaMaxValue: Real = 145.0
    in
    Observations->select(obs | obs.code.value = 'Na' and (obs.value.value
> NaMaxValue and obs.value.units = 'mmol/L')
```

Note that the two pieces of code are NOT the same in that the OCL form will not count any observations which do not have the correct units. The best we can do to improve this would be to convert the PQ to the specific units during our calculation, however it can be seen that our example becomes somewhat more complicated and possibly error prone than the GELLO equivalent

```
Context SinglePatient::GetSodiumObservationsGreater()

 body:
    Let NaMaxValue: Real = 145.0
    in
    Observations->select(obs | obs.code.value = 'Na' and (obs.value.convert
('mmol/L').value > NaMaxValue)
```

Also note the use of the "in" reserved word in the OCL examples. GELLO R2 permits the "in" operator also but has an specialized grammar which allows the "in" to be ommitted.

The issue of not being able to use a factory become even more apparent when developing more complicates queries such as the following.

```
 Let IBD: CodedValue = factory.CodedValue('24526004', 'SNOMED-CT') --IBD
 Let Colitis: CodedValue = factory.Codedvalue('64226004', 'SNOMED-CT') --
Colitis

 Let ColitisCount: Integer = FamilyHistory.Relative -> select(x |
                  (x.ClinicalGenomicChoice.clinicalObservation ->Select
(code.implies(Colitis)or code.implies(IBD)).size() > 0)
                  and x.LivingEstimatedAge < 20).size()

 ColitisCount > 0
```

In this case, it is practically impossible to write equivalent OCL code without the use of a specific constructor for the class CodedValue. One cannot simply expand the properties of the class CodedValue since we need to call the specific operation CodedValue.implies() which only operates on two coded values. The only workaround available in OCL would be for every class in the Data Model to export constructor method for instances of those class.