White Paper v. 1.2

Speaking the same language: sharing clinical knowledge with GELLO expression language

[a]Margarita Sordo, Ph.D, [a]Robert A. Greenes, M.D., Ph.D.

[a]Decision Systems Group, Brigham & Women's Hospital, Harvard Medical School, Boston, MA

msordo@dsg.harvard.edu

December 2004

1. Context

As a developer/user of computerized clinical guidelines particularly, and clinical knowledge in general, you have come across valuable pieces of encoded clinical knowledge that you wish could be able to incorporate in your systems. However, the growth of multiple, incompatible clinical applications, each based on its own hardware architecture, platform, and operating system; each having patient records stored in ad hoc databases specifically tailored to satisfy the needs of a particular institution, precludes you from reusing valuable knowledge and incorporating it in your clinical applications without the painful process of restructuring and sometimes, rewriting it all to meet your particular specifications.

There is a better way for doing this: with GELLO, you can represent your clinical knowledge, e.g. decision rules, eligibility criteria, patient state specifications with expressions that are:

- Specifically targeted to clinical applications
- Object oriented
- Platform independent
- Vendor independent
- Intended to access data through an OO data model – an RMIM compatible with the HL7 RIM
- Robust, strongly-typed, declarative and side-effect free.

Thus use of GELLO:

a. provides platform-independent support for mapping to the OO data model used. Therefore eliminating the need for implementation-specific encoding methods for information retrieval as part of knowledge content (guidelines, alerts, etc.).
b. simplifies the creation and updating of clinical data objects, and their evaluation.
c. facilitates sharing of decision logic and other computer expressions.
d. reduces errors and discrepancies in knowledge representation

As shown in Figure 1 decision rules, eligibility criteria, patient state specifications, guidelines, alerts, etc can be written as GELLO expressions embedded in clinical applications.

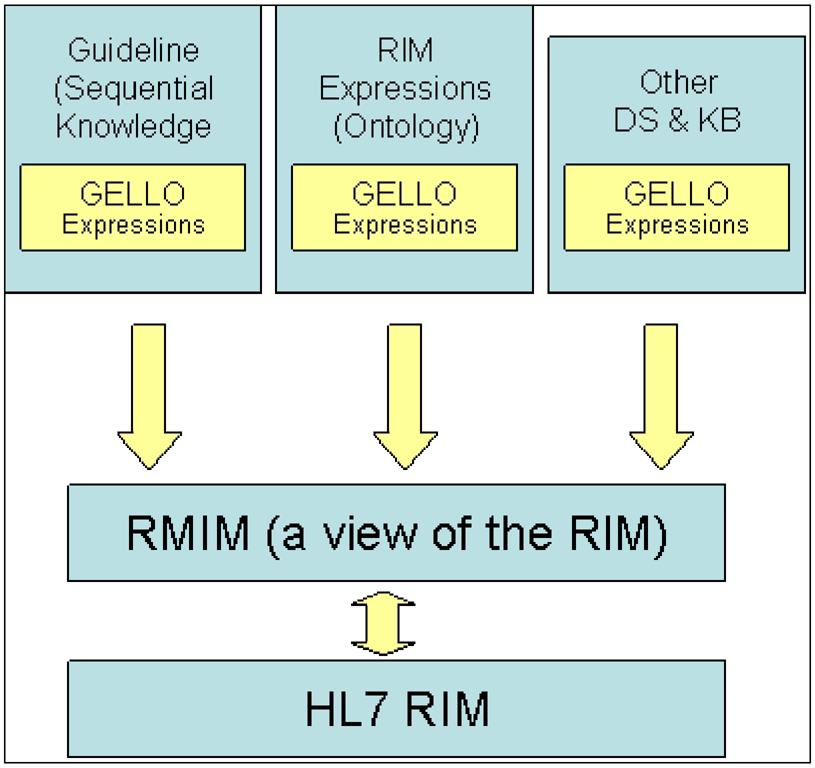

Figure 1: GELLO expressions across multiple platforms

With GELLO, clinical knowledge can be shared across platforms, systems and applications, facilitating true, seamless sharing of information.

2. GELLO expression language

Based on the Object Constraint Language (OCL), GELLO expression language has been designed to meet the challenges of clinical knowledge in the context of heterogeneous clinical environments. Most important among these challenges is a platform independent knowledge representation that can be used and shared by any clinical application. GELLO follows the same conventions as OCL in relationship with the UML metamodel. In other words, every GELLO expression is written in the context of a specific type.

GELLO incorporates most of OCL functionality with the exception of some unneeded capabilities have been removed a)pre and post conditions for constraints -not used in GELLO and b) invariants -also for constraints. The UML ITS implements the semantics of the HL7 Abstract Data Types in such a way that HL7 data types are mapped into the core UML and OCL kernel data types where such mappings are appropriate. Such functionality compatible with the HL7 RIM is used by GELLO, hence enabling it to reference instances of patient-specific data independent of platform, vendor, and data model.

GELLO expressions can be 'translated' to other languages, e.g. SQL, Java by means of a compiler producing executable code compatible with institution-based applications. While at the same time, GELLO expressions can be shared among institutions with different systems without the need of rewriting.

Figure 2 shows how GELLO expressions embedded in applications are handled. The green rectangle on the left hand side is an application running in a particular setting. GELLO expressions appear wherever evaluation criteria are required. Such expressions may consist of a single expression as in Figure 3 or as 'chunks' of more complex logic as the examples presented in section 3.

When a GELLO expression is found, the GELLO evaluation module is invoked (yellow rectangle in Figure 2). As a first step, GELLO expressions are parsed and compiled; this generates a) executable code and b) the query request for data. Depending on the DBMS handling the RMIM, such queries can be in SQL –for relational databases; OQL – for object-oriented databases; XQL –for XML documents. Step 2 sends the request to the DBMS. The DBMS handles the transactions needed to retrieve the requested data from the RMIM. Extracted data is sent back for evaluation (step 3) where GELLO expressions are instantiated and evaluated. The result of evaluating the GELLO expression is sent back to the application and the execution flow continues.

The context where GELLO expressions are to be evaluated can be defined either explicitly by using the Context reserved word as is the case in the example presented in Figure 3, where the context refers to a specific patient in contrast to all the patients in the data repository.

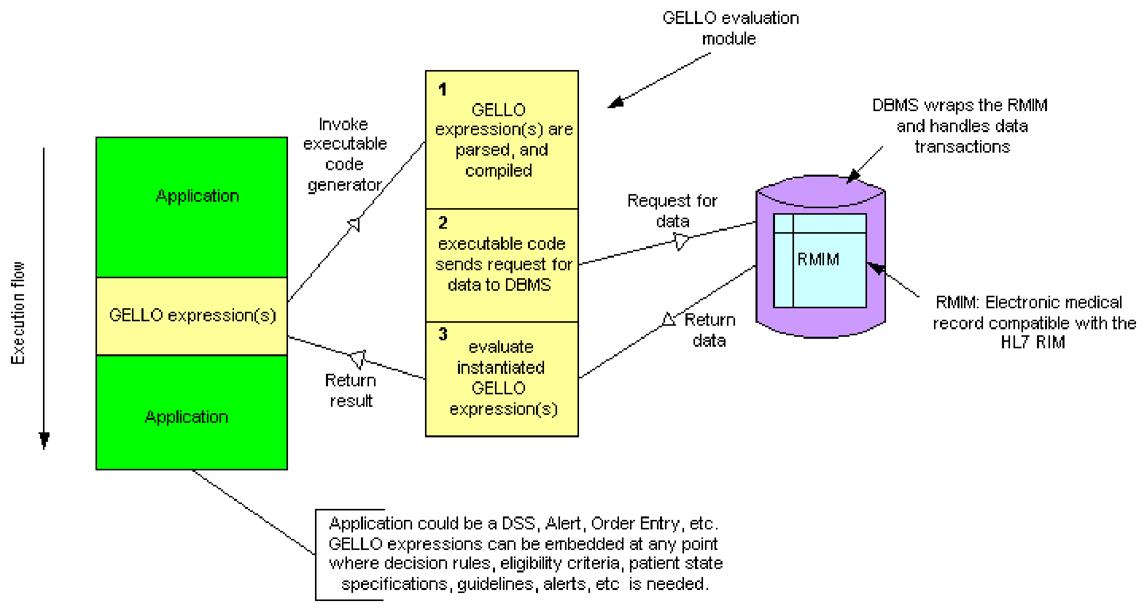
GELLO evaluation module
Execution flow
Application
GELLO expression(s)
Application
Invoke executable code generator
Return result
1 GELLO expression(s) are parsed, and compiled
2 executable code sends request for data to DBMS
3 evaluate instantiated GELLO expression(s)
Request for data
Return data
DBMS wraps the RMIM and handles data transactions
RMIM
RMIM: Electronic medical record compatible with the HL7 RIM
Application could be a DSS, Alert, Order Entry, etc. GELLO expressions can be embedded at any point where decision rules, eligibility criteria, patient state specifications, guidelines, alerts, etc is needed.

Figure 2: GELLO and applications: How data needed to instantiate GELLO expressions is retrieved.

In cases where the context is implicitly defined by the application, e.g. an alert about an abnormal lab result –of a particular patient- such information must be provided to the GELLO evaluation module (yellow rectangle in Figure 2).

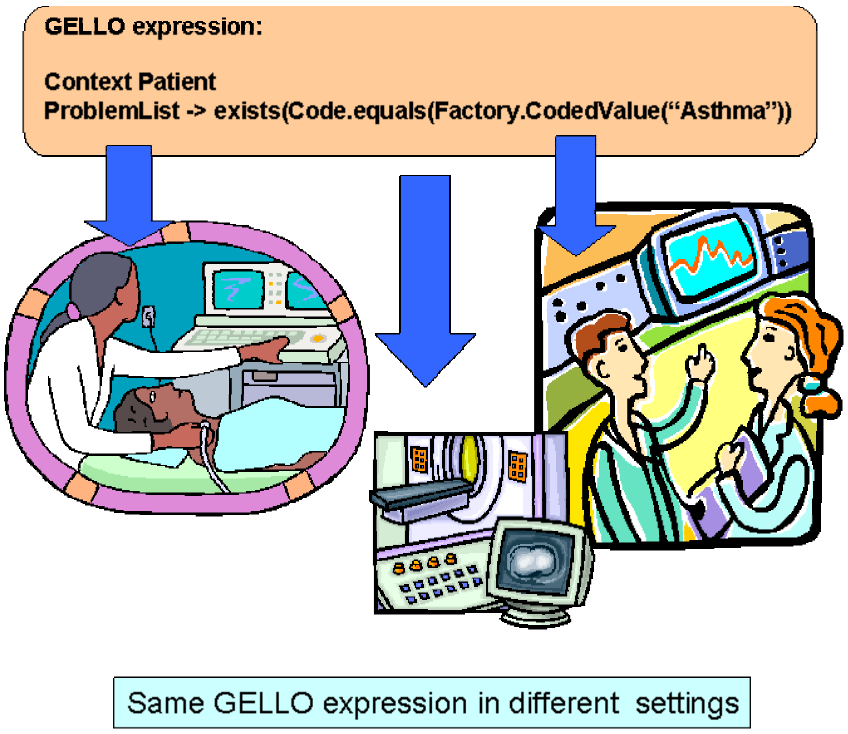
GELLO expression:
Context Patient
ProblemList -> exists(Code.equals(Factory.CodedValue("Asthma"))
Same GELLO expression in different settings

Figure 3: Sharing knowledge between applications: One GELLO expression, three different settings.

1.1 GELLO Types

GELLO language consists of basic data types, tuple and collection types, and a series of operators for manipulating expressions (the full description of GELLO language can be found [1]). Also, GELLO object-oriented notation accesses classes and their attributes in the underlying data model, triggering the execution of class-related methods. The result is seamless integration with institution systems while preserving shareability of knowledge.

GELLO is a strongly-typed language. This means that every expression must have a known type. There are two main type categories in GELLO: predefined types and model (user-defined) types.

Predefined types include basic types, collection types and tuple type.

Basic Types

A primitive data type is named by its reserved word. They are: Boolean, integer, real, string. Having basic data types allows us to create literals of the form:

- `let aNum : integer = 5`

- let xNum: real = 5.6
- let aString: string = ‘abc’ (note single quotes)
- let aBoolean: Boolean = true

The Boolean type in GELLO is a three-valued type. A GELLO Boolean type can have one of three possible values: true, false, and unknown. The arising discrepancy between GELLO unknown and HL7 RIM Null as third Boolean value has been resolved by the UML ITS data types, which map unknown into Null and vice versa.

Each GELLO types has each own set of built-in operators that can be utilized in GELLO expressions to handle information (the full description of GELLO built-in operators can be found [1]).

Model Types

Model types refer to user-defined classes in the underlying data model. References to such types include either a full description name, e.g. PhysicalQuantity or its alias, e.g. PQ. Both examples refer to the model type PhysicalQuantity as defined in the HL7 RIM.

In both Basic and Model types hierarchy is observed. Hence, for Basic types, Integer conforms to Real class. As for any given class in the data model, if class B is a subtype of class A, then class B conforms to class A

A little explanation on Classes and their attributes

A *class* is a construct that defines the fields and methods of the specific concrete objects that are subsequently constructed from that class. Fields are the data for the class. Methods operate on the data.

A class is like a blueprint that defines how an object will look and behave when the object is created or *instantiated* from the specification declared by the class. Concrete objects are obtained by instantiating a previously defined class.

Variables

A variable declaration declares a variable name and binds it to a type. A variable can only hold a value of the same type. A declaration defines the name for a variable inside a program. Each variable declaration must have a scope.

Variables are declared using the **let** OCL expression. The type of the variable can be either one of the basic built-in GELLO data, collection or tuple types, or a class from the underlying data model. The type of the return value of an expression must match the type of the variable to which such a value is to be assigned. Once an expression is bound to a variable, it cannot be changed. The generic syntax for let expressions is as follows:

let variableName: type = Expression.

An example:
let threshold_for_osteodystrophy : integer = 70

Collection operators and the ‘arrow’ → notation

All operations on collections in GELLO are denoted with the ‘arrow’ → notation. The ‘arrow’ notation distinguishes a collection operation from a model type operation. Hence the notation is as follows:

collection → collectionOperator(parameters)

3. Examples in GELLO

1.1 An MLM into GELLO

From a MLM:

```
   maintenance:
      title: Screening for elevated calcium-phosphate product;;
   library:
      purpose: provide an alert if the product of the blood calcium and
         phosphorus exceeds a certain threshold in the setting of renal
failure;;
      explanation: An elevated Ca-PO4 product suggests a tendency toward
renal
         osteodystrophy and predisposes to soft-tissue calcification;;
```

Example in GELLO

```
   let lastCreatinine : Observation = Observation→ select(code=
         (”SNOMED-CT”, “xxxxxx")).sortedBy(efectiveTime.high).last()

   let lastCalcium : Observation = Observation→ select(code =
         (“SNOMED-CT”, “yyyyy”)).sortedBy(efectiveTime.high).last()

   let lastPhosphate : Observation = Observation→ select(code=
         ("SNOMED-CT", “zzzzz”)).sortedBy(efectiveTime.high).last()

   let renal_failure_threshold : PhysicalQuantity =
         Factory.PhysicalQuantity( "2.0, mg/dl")

   let threshold_for_osteodystrophy : int = 70

   let renal_failure :Boolean =    if lastCreatinine <> null and
         lastCreatine.value.greaterThan(renal_failure_threshold)
      then
         true
      else
         false
      Endif
   let calcium_phosphate_product : real = if lastCalcium
         <> null and lastPhosphate <> null
      then
         lastCalcium.value *  lastPhospate..value
      else
         -1
      endif

   if renal_failure and calcium_phosphate_product >
         threshold_for_osteodystrophy then
      whatever action or message
   else
      whatever action or message
   endif
```

1.2 Example of an iteration over more than one collection at a time

This example shows how collection operators can be nested in expressions as long as they comply with the notation

<u>Statement in English (many thanks to Samson Tu):</u>
"There exists (for a patient) an anti-hypertensive prescription (?drug) such that there exists (for the patient) a problem (?problem) such that ?problem is a compelling indication for ?drug".
Where:'
- a patient' is the current patient
- ?drug is any drug in the drug database
- ?problem is a patient's problem

```
   Presence of Azotemia Observation within last three months :
Assumptions:
         1. The data model has as code a generic term such as
             SNOMED "finding" ("246188002") and the value slot has
             the code for Azotemia.
         2. For a diagnosis such as azotemia, the effective time is the
time
             interval during which the disease is thought to be present.
         3. A PointInTime.NOW() function returns the current time
```

<u>Example in GELLO</u>

```
      Let month : CodedValue = Factory.CodedValue(""SNOMED-CT",
"258706009"")
      Let finding : CodedValue = Factory.CodedValue("SNOMED-CT",
"246188002")
      Let azotemia : CodedValue = Factory.CodedValue ("SNOMED-CT",
"371019009")
      Observation → exists(code.equal(finding) and value.implies(azotemia)
and
              effective_time.intersect(ThreeMonthsAgo, PointInTime.NOW()))
```

1.3 Example: Number of current anti-hypertensive Medications > 1

<u>Statement in English (many thanks to Samson Tu):</u>

Number of current anti-hypertensive Medications > 1

Example in GELLO

```
      Let hypotensive_agents : CodedValue =
            Factory.CodedValue("SNOMED-CT, 1182007")

      MedicationOrder→ select(code.(hypotensive_agents) and
            effectiveTime.high = null)→ size() > 1

      MedicationOrder→select(code.implies(hypotensive_agents) and
      effectiveTime.high = null)→size() > 1
```

1.4 3rd Td dose before 12 months of age

Statement in English (many thanks to Samson Tu):
3rd Td dose before 12 months of age

Example in GELLO

```
      Let month : CodedValue = Factory.CodedValue(""SNOMED-CT",
"258706009"")

      Let DOBcode : CodedValue = Factory.CodedValue ("SNOMED-CT",
"184099003")

      Let DateOfBirth : Observation= Factory.Observation→ select(
            code.equal(DOBCode)).sortedBy(effectiveTime.high).last()

      Let TwelveMonthsOfAge : PointInTime = Factory.PointInTime(
            DateOfBirth.effectiveTime.high.plus(12, month))

      Let Td :CodedValue = Factory.CodedValue("SNOMED-CT", "59999009")

      Let ThirdTdDose : SubstanceAdministration =
Factory.SubstanceAdministration→
            select(code.implies(Td)).sortedBy(effectiveTime.high)).third()
            ThirdTdDose.effectiveTime.high.before(TwelveMonthsOfAge)
```

4. Conclusions

For clinical applications that depend on cross-platform, cross-application share of knowledge, GELLO expression language solution overcomes the inherent challenges of traditional encoding and sharing of knowledge across heterogeneous clinical environments. By eliminating the need of ad hoc encoding to satisfy specific requirements, easing the sharing of clinical knowledge and reducing the risk of inconsistencies and errors of "same knowledge source –multiple encodings", GELLO is challenging the way healthcare entities communicate and share information. . To find out more about GELLO expression language, see [1]-[3].

5. Acknowledgements

Many thanks to Dongwen Wang for his valuable comments.

6. References

[1]. Margarita Sordo, Omolola Ogunyemi, Aziz A. Boxwala, Robert A. Greenes, Samson Tu. Software Specifications for GELLO: An expression language for clinical decision support. http://www.hl7.org/v3ballot/html/foundationdocuments/welcome/index.htm

[2]. **Sordo M**, Ogunyemi O, Boxwala A, Greenes R. GELLO: An Object-Oriented Query and Expression Language for Clinical Decision Support. Symposium of the American Medical Informatics Association (AMIA), November, 2003

[3]. **Sordo M**, Boxwala A, Ogunyemi O, Greenes R. Description and Status Update on GELLO: a Proposed Standardized Object-Oriented Expression Language for Clinical Decision Support. Proceedings of Medical Informatics Association (MedInfo), September, 2004