

# Logic and a Little Language for Heritage Resources on the Semantic Web

Andrew Green (Instituto Mora, Mexico City) ahg@servidor.unam.mx

## Summary

The work presented here is part of the interdisciplinary project *Rescuing Memory: Image Preservation, Information Systems, Access and Research*, which investigates the dissemination, study and management of heritage resources, and attempts to provide solutions to problems often faced in the realization of these tasks.

One component of this project is the development of a scalable digital library application based on a back-end of semantically modeled data. In designing the application, called *El Pescador*, we encountered (what we believe to be) novel ways around obstacles to using the Semantic Web as a mechanism for modeling catalogue data in applications of this type. Our work has focused on the following areas: (1) information flow between the model and the user interface; and (2) mechanisms used to establish or configure algorithms used in that process.

Our design proposal for *El Pescador* defines—among other things—a domain-specific language, has been partially implemented, and will soon be updated following feedback from the current implementation.

## Key Problems Encountered

- (1) The lack of adequate studies of algorithms for processing information as it moves from the model to the user interface, and, following user input, from the interface back into the model.
- (2) The lack of a unified, scalable mechanism for defining vocabulary, model structure and model-interface translation rules.

## Proposed Solutions

In response to problem (1):

We analyzed model-interface translation from a general perspective and designed configurable algorithms that may be applied to a wide range of semantically modeled catalogue data and record presentation formats. This proposal, called "Knowledge Repository Logic", applies information engineering best-practice principles, such as encapsulation and modularity, throughout.

In response to problem (2):

We created a domain-specific (or "little") programming language (as yet unnamed—see below) with which the programmer/model designer may establish vocabulary and model structure and set the parameters of model-interface translation algorithms. This "full-stack" solution allows related aspects of a semantic catalogue implementation to be established together, avoids the repetition of information in this process, and offers simplified access to recurrent model and interface structures, thus providing both easy deployment and scalability, and applying, again, said best-practice principles.

## More about the Project

For more information about the larger project this development is a part of, please see <http://durito.nongnu.org>. All the software we create is released as free/open source (see below for version control), and we strongly promote open access to heritage materials.

## Details

### Keep Stuff Together

Key principles in system design include the non-repetition of algorithms and keeping together stuff that goes together. Since definitions of vocabulary, model structure and model-interface translation rules must refer to the same objects, it doesn't make sense to define those objects repeatedly in different locations scattered throughout an application. As the language example shows, our proposal avoids this problem; note, however, that this does not mean departing from the standards: if elements definable by the language can also be established using an existing norm, in most cases it will be possible to implement the "export" of those elements to the appropriate standard format. This is already the case for vocabulary definitions, which are automatically exported to an RDF Schema in the current implementation.

Note also that the preceding example does not demonstrate everything we propose to define through the language. Other elements that are currently implemented or may be implemented in the future include: inference rules, more elaborate rules for graph generation and traversal, input validation rules, specialized ordering (comparison) rules, and rules for constructing what we call "variable descriptions" (essentially, shorter descriptions whose contents may be determined on-the-fly by the system in certain circumstances).

### Abstracted Statement Layer above SW-Conformant Graph

In the models we've worked with, certain low-level graph patterns become so routine that it makes sense to encapsulate and simplify them. This is the case of repeated properties that need to remember their order, and alternate versions of a literal for different languages, for example. In addition, existing norms do not provide certain facilities that we find useful, such as datatype hierarchies, the simultaneous assignment of a language and a datatype to a literal, or a mechanism to distinguish structured values from other resources. Thus, we've constructed an abstracted statement layer that provides these facilities and is operated on by our little language; beneath this abstracted layer lies a SW graph that reflects it, modeling everything in it in a conformant, though perhaps slightly less beautiful, manner.

(Point **(A)** in the language example shows this: the term `multipleOrdered` there means that, for nodes linked to the `ruleset person`, repeated `swv:hasFavoriteMovie` properties have an order that must be remembered and included in the model. In the SW-conformant graph this is implemented using an `rdf:Seq`; but the abstracted statement layer hides this detail, allowing simplified access to the order of the repeated properties.)

### Better Path Definitions

Current mechanisms for describing paths between nodes in a model are text-based relatives of SQL. When the complexity of a model increases and certain segments of complex paths appear repeatedly in several parts of an application, these mechanisms provide insufficient flexibility and scalability. Our proposal allows for the encapsulation and combination of definitions of segments of paths, thus avoiding repetition in, and augmenting the semantic value of, path descriptions. (See point **(B)** in language example.)

### Model Everything? No!!!

Semantic modeling is fun and good for you, too, but all the world is not a semantic model. Call us old-fashioned, but we think some things are still best expressed by plain old code. You can see how we use blocks of executable code in points **(C)** and **(D)** in the language example.

### Levels of Presentation Logic

Some schools of thought in Cognitive Linguistics posit a strong distinction between the rules governing expression and those that apply to human mental models. Following the (limited) analogy that may be drawn between knowledge representations and mental models, we have taken a cue from this hypothesis, and have integrated in our work the assumption that a user should not see a direct encoding of the contents of a model, but rather a complex transformation that follows the rules of human communication. In other words, while other Semantic Web projects are rooted in the assumption that language is compositional, we work from the idea that—as Gilles Fauconnier puts it—"expression is not compositional formal encoding that mirrors a compositional conceptual construction."

Concretely this view is most reflected in our description templating system (see point **(D)**), which allows a programmer to use default templates when he or she knows they would produce appropriate results, or tweak the generation of descriptions extensively, a task that is often necessary to provide the user with concise, friendly output.

At the same time, however, the templating system remains media-agnostic: all templates may be used for output to diverse formats, including Web, print, or even spreadsheet. Therefore, this part of the system constitutes an initial, generic layer of presentation logic, whose output is manipulated by another layer that transforms the non-specific descriptions it receives into concrete user-consumable resources.

### Default Model Organization: Convention over Configuration

The current implementation of *El Pescador* provides a default global organization of the model, which we expect will be appropriate for nearly all foreseeable uses of the system. (See point **(E)**.)

### Original Design Goals

Following is a summary of our original design goals:

- to allow the reuse of operations, rules and configurations in different parts of the KR logic;
- to provide a basis for the implementation of a wide variety of user functions;
- to allow said functions to be at once complex, flexible and easy-to-use;
- to allow a complex and expressive knowledge representation back-end;
- as appropriate, to keep configuration information in one place (or few places);
- to provide simple, intuitive, unrepeatative and modular configuration mechanisms;
- to use the right configuration mechanisms for each part of the KR logic;
- to provide a basis for the optimization of operations for large repositories; and
- to support the inclusion of the KR logic system in an appropriate global application architecture.

### Status and Roadmap

The implementation being demonstrated uses a previous, more cumbersome version of the language's syntax. The system will soon be modified to accept the syntax that appears on this poster. The current packaging mechanism is also different from that shown on the poster.

A website that uses *El Pescador* to disseminate and search a small collection of 19th century photographs will soon go online. After that is completed, a major refactoring of both the language and the system is in order. Possible changes or additions *may* include:

- An improved mechanism for defining inference rules. (The original proposal for this was never implemented.)
- Inheritance among rulesets.
- Global rule definitions.
- More flexible and elegant mechanisms for determining how nodes are linked to rulesets.
- Embedding of description templates in other description templates.

### What's in a name?

Our little language needs a name! If you have any suggestions, please let us know.

Download this poster from <http://durito.nongnu.org/docs/innsbruck.pdf>!

Poster content provided under the Creative Commons Attribution-Non-Commercial-No-Derivs license.

Original (now outdated) documentation of our work is available, in Spanish, from

[http://200.67.231.185/mediawiki/index.php/Pescador:L%C3%B3gica\\_de\\_RC](http://200.67.231.185/mediawiki/index.php/Pescador:L%C3%B3gica_de_RC).

Developer resources, including version control, are at

[http://200.67.231.185/mediawiki/index.php/Pescador:Recursos\\_para\\_desarrolladores](http://200.67.231.185/mediawiki/index.php/Pescador:Recursos_para_desarrolladores).

Many thanks to the Instituto Mora and the Open Digital Library Network for their support. Thanks also to the Audiovisual Social Research Lab and the other programmers involved in this effort: José Antonio Villarreal, Sandra Luz Aguirre and Alejandro Martínez.

## Language Example

*File: person.def*

```
# Stuff related to people
realm swv

vocabulary {
  class swv:Person {
    "Person"@en
    "Persona"@es
    subclassOf swv:Agent
  }

  property swv:hasGivenNames {
    "Given Names"@en
    "Nombres"@es
    comment "All of a person's given names."@en
    domain swv:Person
    range xsd:String
  }

  property swv:hasFamilyNames {
    "Family Names"@en
    "Apellidos"@es
    comment "All of a person's family names."@en
    domain swv:Person
    range xsd:String
  }

  property swv:hasFavoriteMovie {
    "Favorite Movie"@en
    "Película preferida"@es
    domain swv:Person
    range swv:Movie
  }
}

ruleset person {
  baseRule has_given_names {
    withProperty swv:hasGivenNames
    minCardinality 1
    maxCardinality 1
  }

  baseRule has_family_names {
    withProperty swv:hasFamilyNames
    minCardinality 1
    maxCardinality 1
  }

  baseRule has_favorite_movie {
    withProperty swv:hasFavoriteMovie
    multipleOrdered
  }

  pathRule favorite_movie_genre {
    has_favorite_movie->movie.has_genre
  }
}

default textFunction full_name_fg {
  {
    given_names = (@root->has_given_names).text
    family_names = (@root->has_family_names).text
    "#{family_names}, #{given_names}"
  }
}

fullDescription {
  block {
    title {
      "Basic personal data"@en
      "Datos personales básicos"@es
    }
    df has_family_names
    df has_given_hames
  }

  block {
    title {
      "Movie tastes"@en
      "Gustos en películas"@es
    }
    <% if ((@root->has_favorite_movie).size > 0) %>
    df has_favorite_movie
    uniqueDF {
      label {
        "Number of genres of favorite movies"@en
        "Número de géneros de películas preferidas"@es
      }
      value {
        textFunction
        { { (@root->favorite_movie_genre).size.to_s } }
      }
    }
    <% else %>
    text {
      "No favorite movies!"@en
      ";Ninguna película preferida!"@es
    }
    <% end %>
  }
}

repository {
  secondary SOC people {
    "People mentioned this catalogue"@en
    "Personas mencionadas este catálogo"@es
    hasGroupDomain swv:Person
    bindToRuleSet person
  }
}
```

Definitions of related things can be placed together in a single file.

A `realm` is something like a package in general-purpose programming languages. Normally all the vocabulary defined in a given `realm` goes in the same ontology.

Here we define some vocabulary.

This will produce labels using `rdfs:label`.

As you'd expect, this line creates an `rdfs:comment` statement.

In this example, we'll assume that the class `swv:Movie` is defined in another file in the same `realm`.

A `ruleset` is a bunch of rules used in graph generation and model-interface translation. In our proposal, nodes are *linked to* one and only one `ruleset`, which determines how they are processed in the system.

`baseRules` establish which properties may be used in statements whose subject is linked to this `ruleset`.

Here we are setting no limits on how many times this property may appear on a node linked to this `ruleset`. (See below for the meaning of `multipleOrdered`. **(A)**)

This rule is for traversing a part of the graph. By defining it we're encapsulating some complexity and optimizing the traversal of matching subgraphs.

Here the identifier `movie.has_genre` refers to the rule `has_genre` defined in the `ruleset movie`; the `pathRule favorite_movie_genre` will thus traverse the graph from any node linked to this `ruleset` to all nodes that may be reached via a path that traverses first a statement with the property `swv:hasFavoriteMovie` and then whatever path is defined by the rule `movie.has_genre`. **(B)**

An upcoming versions of the language will allow the attachment of additional information to `pathRules`, specifically to help the search subsystem generate clearer, more relevant results.

Here we define a block of executable code using a general-purpose programming language. Currently the application can process blocks written in Ruby, though other languages could also be implemented.

The purpose of this particular block is to generate a snippet of text that may "represent", in the UI, nodes linked to the `ruleset`. The special variable `@root` refers to the node currently being processed. **(C)**

This template controls the generation of full descriptions of nodes linked to this `ruleset`. (Full descriptions are like the full records of objects in a catalogue.) The template's elements are media-agnostic. Label-value pairs are called "description fragments" and are generated using either `dfs`, which refer to rules defined elsewhere in the `ruleset`, or `uniqueDFS`, which can embed text-generating code directly in the template.

Code between these symbols `<% %>` controls the conditional execution of parts of the template. **(D)**

Here we establish a collection of people and bind all of its members to the `ruleset person`. **(E)**