OML: A Scripting Approach for Manipulating Ontologies

Nuno Carvalho
Departamento de Informática
Universidade do Minho
Braga, Portugal
narcarvalho@di.uminho.pt

Alberto Simões
Instituto de Letras e Ciências Humanas
Universidade do Minho
Braga, Portugal
ambs@ilch.uminho.pt

José João Almeida
Departamento de Informática
Universidade do Minho
Braga, Portugal
jj@di.uminho.pt

ABSTRACT

There are different definitions for ontologies. Different knowledge areas tend to define ontologies in a different way. For computer science, an ontology can be used to describe, in a well defined and structured way, knowledge about a specific domain. These artifacts store rich information that can be reasoned about, this information can also be target of many structured processing functions. There is a diversity of programs that can be implemented to take advantage of these features and produce applications in every area of knowledge.

The Ontology Manipulation Language (OML) is a Domain Specific Language (DSL) designed to describe and execute operations that reason about ontologies. These reasoning operations can be used to manipulate and maintain the current information in the ontology, infer new knowledge or concepts, or even produce any kind of side effect. OML is a simple and descriptive language, yet it is powerful enough to implement complex operations or reasoning engines in a clear and efficient way.

To actually run programs written in OML a stand alone compiler is available, as well as a mechanism for embedding OML programs in a generic programming language. This allows the quick development of applications that make use of ontologies, by describing ontology related operations in wove OML snippets throughout the code. This mechanism has proven to be a very effective and clear approach for taking advantage of adopting ontologies to represent information, while maintaining the implicit advantages of using a general-goal programming language.

1. INTRODUCTION

Ontologies are a common approach to store knowledge. They are used in a wide range of applications in all areas of science. Many heterogeneous standards and options for storing these artifacts in a persistent and processable way are available. The Web Ontology Language (OWL) [5] is a good example of a well accepted family of schemas for publishing and sharing ontologies mainly for the World Wide Web (WWW). Topic Maps [7] and the Simple Knowledge Organization System (SKOS) [6] are other good examples of approaches for creating and maintaining ontologies. Since there are several ways to represent ontologies, there are also different approaches to manipulate them. Several software packages offer methods to change and manipulate information in a ontology, as well ways to infer new information and reason about the existing knowledge. Engines to infer new knowledge from the existing information are a very interesting feature of some of these languages [4].

Tools for these formats can be divided in two major groups: tools based on graphical interfaces, and tools that offer interaction based on Application Programming Interfaces (API). The tools that implement a graphical interface for manipulating information are suitable approaches to be used by humans in common situations. This is not exactly our aim since we plan on using our ontologies and reasoning rules to build more complex tools, and most of the manipulation operations should be decided in runtime. Tools that do not implement a graphical interface but rather expose an usable API for executing operations, allow a wider range of possible applications, since more complex tools can be built using the provided interfaces. This approach suits our needs best, providing a complete module to manipulate information through a well defined API.

Tools available today for manipulating information described in ontologies are very powerful and can easily deal with many common problems, however, sometimes, they still lack some properties that would greatly increase their efficiency and adoption. One of these would be more expressiveness or efficiency on their languages syntax: many times more effort is spent to comply the language standards and specifications than dealing with the problem itself. Another major problem is that most of these tools work at very low abstract levels, making it hard to develop simple tools that can be elegantly composed to solve complex problems.

With these concerns in mind some design goals for the OML were defined:

- It should be a simple and expressive language, easy to use and easy to extend, simple and clean but powerful enough to allow creating complex tools;

- The language needs to be compact and elegant, aiming at high-level languages, achieving more in less syntax, but maintaining readability.

- It should allow a declarative approach for defining operations, suitable for writing transformation and reasoning engines.

- Allow a modular approach, building small blocks that can be composed together to build more complex operations.
- While we aim an expressive and powerful language, there are details that should not be handled by OML. For example, we should not build an interface for relational databases in our languages. Therefore, OML should be possible to mix OML blocks in a general purpose programming language.
- Giving this modular approach, keep in mind a scripting flavor for the tools, allowing us to build tools that can be easily combined together with already existing programs in pipelines for example, *Linux style*.

OML is a domain specific language that can be used to write programs that act on knowledge stored in a ontology. OML is a domain specific language, it can be used to write programs. Programs can be executed using a special program, typically this program is called a compiler. Figure 1 illustrates the execution of an OML program. We feed the *compiler* a program (written in OML) and an ontology, and a result is produced. The *result* can be changes on the ontology itself, or any kind of arbitrary side effect. With current implementation, the ontology that is feed to the compiler needs to be concretized using the `Biblio::Thesaurus` framework. This framework allows storing information in a ontology using an information structure based on ISO 2778, initially created to allow the description of monolingual *thesaurus*. It has already been use for building applications with success in previous work, see [1] and [9].

The principle for writing programs in OML is simple, we define a pattern and an action. Then we search the ontology for that pattern, if the pattern is found the action is executed. Note that a pattern can be found once or more that once, in this later case the action block is executed once for each instance of the pattern found.

In the next chapters of this article we start by introducing the OML language specification and how to write programs, we briefly describe the current implementation of the OML compiler. And finally describe some applications and tools created with OML that illustrate its' use.

2. OML SPECIFICATION

OML is a simple language, one of the major goals during design was to make sure that it would be easy and intuitive to use, even for people without any programming language background. In this section we illustrate a glimpse of what OML can do.

In a nutshell OML programs are a sequence of statements which are executed in order. Each statement consists of a `<pattern>` block, everything on the left side of the *fat-arrow* operator (`=>`) and an `<action>` block, everything on the right side. A statement always ends with a single dot (`.`), as shown here:

```
<pattern> => <action> .
```

2.1 Patterns

Patterns are used to describe knowledge in the ontology. Typically some action needs to be performed when this pattern is found. Patterns can be used to represent simple terms or relations between terms, or any combination of these. Table 1 illustrates some patterns that give an idea of what can be done.

#	**Pattern**	
1	`term(Buster)`	term *Buster*
2	`rel(ISA)`	relation *ISA*
3	`term($t)`	for all terms
4	`rel($r)`	for all rels
5	`Buster ISA cat`	
6	`$pet ISA cat`	
7	`$pet ISA $animal`	
8	`Buster $rel $term`	for all related do Buster
9	`Buster ISA cat ∧ Twitty ISA bird`	
10	`Buster ISA cat ∨ Twitty ISA bird`	
11	`$c ISA cat ∧ $b ISA bird`	

Table 1: Example patterns.

The most simple pattern that can be defined is a single term or a single relation. Pattern 1 shown in Table 1 will evaluate as found if there is a term in the ontology named `Buster`. A single relation relation described in a pattern is shown in Pattern 2 in Table 1. This pattern will evaluate as found if there is at least one relation named `ISA` in the ontology. Variables can be used instead of terms, or relations names. So, Pattern 3 shown in Table 1 describes all the terms in the ontology, and Pattern 4 represents all the relations. Of course that more interesting would be to describe facts, relations between terms, a very simple example of a pattern that describes a fact is Pattern 5 in Table 1. This pattern is considered found if the term `Buster` and the term `cat` are linked by a relation named `ISA`.

Variable containers can also be used in patterns, which means that the pattern can be found more than once for a given ontology. Pattern 6 in Table 1 is one possible example. This pattern represents all the facts that relate the term `cat` with any other term by a relation named `ISA`. Another example of using variable containers is Pattern 7 in Table 1. This pattern represents all the possible combinations of facts that relate terms with the `ISA` relation.

Patterns can be grouped together using the binary operators `AND` and `OR`, which have their traditional meaning. Patterns paired with the `AND` operator will be evaluated as found if both patterns are found, and if they are paired with the `OR` operator only one needs to be found in the ontology for the pattern be evaluated as found. Patterns 9, 10 and 11 in Table 1 illustrate this.

2.2 Actions

After being able to specify the patterns we are looking for in the ontology we need to describe the operations that are going to be executed when the pattern is actually found. Any number of operations can be executed in an action block. Operations are executed in order and can be one of the following types:

- an operation from the predefined list of operations available, this is typically used to add or change the current knowledge of the ontology, for example adding or removing facts or relations;
- or we choose to define our own operation, and write the complete code, this is typically used to produce any arbitrary side effect, updating a data base, printing, creating a PDF file or anything else.

An example, of using a predefined operation can be:

```
add(Buster ISA Mammal)
```

This adds new information to the ontology, specifically relating the term `Buster` with the term `Mammal` using the relation `ISA`. Variables found in the pattern can also be used in the action side of any statement, having their values instantiated according to the pattern found, which means we can write an action block that looks something like:

```
add($pet ISA Mammal)
```

This action would be executed an arbitrary number of times, one time for each instance found in the ontology. The variable `$pet` is automatically replaced with the term (or relation) that matched in the pattern.

As advertised before we can also produce any side effect, by executing any arbitrary action, for example:

```
sub { print $name; }
```

The `sub` keyword has a special meaning, it means that the following action block is a user defined operation and that needs to be executed as is. At the current time this block needs to be written in the programming language Perl [3]. Remember that any side effect can be produced with this, approach, for example adding information to a relational database:

```
sub {
  $db->execute(
    'INSERT INTO terms (name) VALUES ($term)'
  );
}
```

Putting everything together we can write statements that look like:

```
$ci CAPITAL $co ⇒ add($ci ISA city)
                  add($co ISA country).
```

This statement says that for every two terms linked by a relation named `CAPITAL` add two new relations linking the first term with the term `city` by a relation `ISA`, and the second term with the term `country` also by a `ISA` relation. Imagine a geographical ontology describing information about cities and countries, in more loosen English this statement reads: *for every city which is a capital, add a fact stating that city is a city, and country is a country.*

This is just a brief overview of what can be written in OML, a more exhaustive and complete introduction to the language can be found in [2].

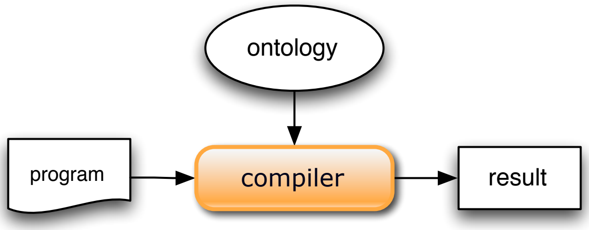


Figure 1: OML architecture overview.

3. IMPLEMENTATION

After being able to write a program in this DSL the next natural step, as in any other programming language is to actually execute the program. An compiler was implemented to allow the execution of programs written in OML. An architecture overview of the system is illustrated in figure 1. In simple terms we feed a program and an ontology to the compiler and after some intermediate stages a final result is produced. This final result can be translated in information changes in the original ontology, any kind of side effects, like updating a external database or producing LaTeX code for example, or even a combination of these.

Figure 2 illustrates the compiler work-flow. Executing a program is divided in three main stages:

The Parsing Stage In this stage the *parser* is responsible for analyzing the source program written in OML, and create a parsing tree (`pTree`). This tree contains the same information that is in the source program but in a more structured way.

The Expanding Stage After creating a `pTree` the control is handled to the *expander* engine, which is responsible for looking at the patterns described in the `pTree`, and expand the pattern by looking for the information in the ontology and storing possible variations of the pattern being searched for. All the instances of the pattern found are stored in an `diTree`.

The Reaction Stage Finally the *reaction* engine is responsible for actually executing the actions described in the initial program. This engine uses the `diTree` to instantiate the variables found in the action blocks of each statement.

These tools are available for download [1]. Full documentation and example applications can also be found there. All the tools were implemented in Perl and are ready to use OML program to build full featured applications. Implementation and design details can be found in [2].

3.1 Embedded OML

We also developed tools that allow the use of OML inside other programs. In this case we also used the Perl programming language to develop a proof of concept. This means that you can write something like this in a traditional Perl program:

```
OML function(arguments)
  ...
ENDOML
```

[1] http://search.cpan.org/perldoc?Biblio::Thesaurus::ModRewrite

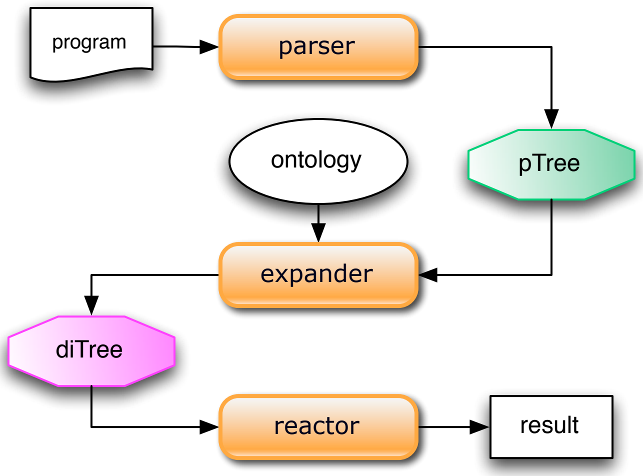


Figure 2: OML compiler architecture overview.

This lets you call `function` as a normal Perl routine, passing needed `arguments`, in a Perl script, but executes OML programs. This has proved to be very useful while building larger applications, because it allows the use of typical Perl tools and frameworks to build applications and use OML programs to handle ontology information and operations.

4. USING OML

We have been using OML to implement several applications that have to deal with information stored in an ontology. A brief introduction to a couple of these applications follows.

4.1 term2dot

The tool *term2dot* given a term and an ontology, creates a graph that represents all the relations for the given term in the given ontology. The graph is created using GraphViz[2]. For example, executing the following command:

```
$ term2dot Portugal geography.onto \
    | dot -Tpng > Portugal.png
```

will create a graph for every relation for the term `Portugal`. The actual graph created is illustrated in figure 3.

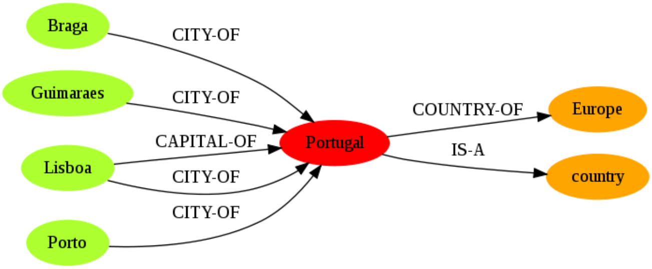


Figure 3: Example of using the tool *term2dot*.

The following source code snippet is the entire code required, to implement the *term2dot* tool.

```
my $term = $ARGV[0];
my $ontology = $ARGV[1];

use Biblio::Thesaurus::ModRewrite::Embed;
process($ontology,$term);

OML process(t)
  begin     => sub{ print "digraph t {\n" }.
  t $r $t1 => sub{ print "t->$t1 [label=$r]"}.
  $t1 $r t => sub{ print "$t1->t [label=$r]"}.
  end       => sub{ print "}\n" }.
ENDOML
```

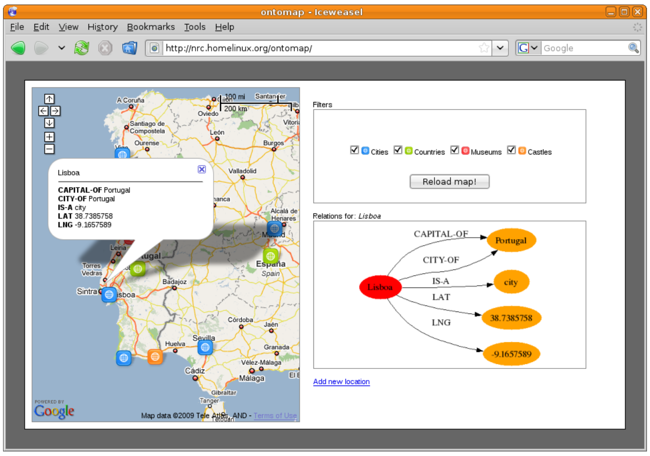
In this small example we can see that a simple OML snippet was included in the source code to query the ontology and print the required information to build the graph with GraphViz. The Perl code here is only used to process the argument passed to the tool. First what is the OML code doing exactly, in the first and last line, where the patterns are `first` and `last` respectively, these patterns are always evaluated as found, so the aassociated operations block are always executed. In this simple case we are using these blocks to print the GraphViz notation header for the graph, and the closing curly bracket at the end. The second line in our OML code has a pattern that evaluates as found for every arbitrary term `$t1` (which acts as a container) by an arbitrary relation named `$r` (which acts as another container) with the term `t` which is passed as argument to this code. In more loosen english this tells the compiler to look for every relation between the term passed as argument and any other term in the ontology. The next line in the snippet is doing the same thing, the only diference between these two lines is that in the first the term given as argument is used as the source term for the relation, and in the second as the target term for the relation. This pattern will evaluate as found for every relation in the ontology for the given term and produce the code in GraphViz notation required to represent that in the graph. In sum this tool produces code in the GraphViz notation that can be later used to build an image illustrating the graph of relations for a given term.

4.2 OntoMap

OntoMap is a very interesting application, it was written in Perl using small embedded OML programs. It manages points of interest in a map and the information is stored in an ontology. It provides a web interface that can be used to display and add new information in a actual map. Perl modules where used to implement everything related to the interface itself, to handle retrieving and updating knowledge from our ontology OML was used. Since embedded programs where used, the overhead of using another language to accomplish some tasks was zero. Figure 4 illustrates the application. One interesting point that can be seen is that we are already using the tool (*term2dot*) described in the earlier section. All the information seen in the application is stored in the ontology.

Every detail regarding anything besides the ontology in handled by typical Perl code, every task concerning the ontology is implemented using OML snippets, mixed inside the application. An example of one of these snippets is:

[2] http://www.graphviz.org/



Lisboa

CAPITAL-OF Portugal
CITY-OF Portugal
IS-A city
LAT 38.7385758
LNG -9.1657589

Figure 4: Example of using the *OntoMap* application.

```
OML find_points
  $p lat $x ∧ $p lng $y
  ⇒ sub {
      to_json({name=>$p, lat=>$x, lng=>$y"})
    }.
ENDOML
```

This is used to answer a AJAX request for the list of points of interest to highlight on the map. To mark points in the map the only thing we need is the points' latitude and longitude, that is exactly what we are looking for in our pattern: all the terms that have a latitude and a longitude, and return a JSON object to pass the information along. This pattern will evaluate as found in the ontology more than onde, because many points are defined there, and each point is related with a value for by a latitude relationa name `lat` and a longitude relation name `lng`, for each of these points we create a string, JSON enconded, with the name of the term found has its' latitute and longitude. This information is enough fot the application interface display the points in a map.

The application allows to filter the points in the map by type. This means that the AJAX request that queries the ontology for the points list to display can also suply an aditional argument which is a list of types of points to show. This illustrates another propriety of OML embebed code, it is dynamic, i.e. this code can be changed in runtime to adapt to new variables. In this specific case we want to change the pattern that was last illustrated to cope with this filter. So the new pattern would be something like

```
  $p lat $x ∧ $p lng $y ∧ $p ISA Castle
    ⇒ sub { ... }
```

This way we have the same behaviour as before but now we are only interested in listing points that are castles, because the user changed that behaviour using the provided interface.

We use OML to handle the information in our ontology and use other language to do everything else. Plus, we use dynamic code to adapt our OML snippets to the requested behaviour of the application.

4.3 OntoMerge

A typical problem nowadays is about merging knowledge on more that one ontology. We wrote a program in OML to verify, given a set of assumptions, if we can merge ontology X with ontology Y. A simple verification can be:

```
OML exists(tA,rel,tB)
  tA rel tB =>
    sub {
      print "relation already exists!";
    }.
ENDOML
```

that simply verifies if a given relation named between two given terms already exists in the ontology. But we can do more complex andtrickier things like, for example: after merging two ontologies we may want to remove the direct transitive closures for a variable number of given relations since we can consider that redundant information. A simple implementation of this can be:

```
$the = thesaurusMultiLoad(ont1,ont2,...)

for $rel ("NT","BT",...){
  rem_trans($the,$rel); }

OML rem_trans(rel)
  $a rel $b ∧ $b rel $c ⇒ del($a rel $c).
ENDOML
```

These tools can be easily composed together to build powerful reasoning engines that act on ontologies. Either for creating to information, or helping in maintaining information.

5. CONCLUSION

Our main objective was to develop a set of tools to manipulate information described in ontologies. Although there are already tools to manipulate ontologies in standard formats, like the SPARQL [8] language, they miss the ability to easily integrate in other programming languages. Therefore, we specified a new domain specific language, called OML, and developed the tools required to run programs written in OML.

As presented in the our example scenarios, the process of writing and using programs with OML is simple, but powerful. Although the language itself is very compact and simple, the programs tend to be very expressive and can be used to implement a wide range of heterogeneous operations.

The ability of embedding OML programs inside a generic programming language was the step that allowed to build richer and fully featured applications. It lets the user to focus on the task he is dealing with, instead of trying to reduce and hook his problem to a set of basic library API functions. This approach makes it possible to do all kind of generic operations one want to perform with nowadays applications (access databases, web services, etc) easily, keeping the ability to manipulate ontologies using a specific designed language.

6. FUTURE WORK

OML is already an usable tool, and much can be accomplished with its current state. But it is yet a work in progress, and there are some points we have to deal with to improve it:

- Software engineering works well with prototyping and evolutionary development. This means that finding more problems where OML can be applied will be crucial to validate the language, to verify its completeness in terms of functionalities, its efficiency and adequacy.

- OML is currently implemented in Perl, using a Perl module that manages interfaces with the ontology data structures. Although Perl is a robust and versatile language it is not not what might be regarded as one of the most efficient language. Therefore, performance tests are relevant. Check how current implementation handles big ontologies or very complicated expressions. Reimplementation of some functionalities can be needed.

7. REFERENCES

[1] José João Almeida and Alberto Simões. T_2O — recycling thesauri into a multilingual ontology. In *Fifth international conference on Language Resources and Evaluation, LREC 2006*, Genova, Italy, May 2006.

[2] N. Carvalho. OML - Ontology Manipulation Language. Master's thesis, University of Minho, 2008.

[3] M.J. Dominus. *Higher-order Perl: Transforming Programs with Programs*. Morgan Kaufmann Publishers, 2005.

[4] A. Gómez-Pérez, M. Fernández-López, and O. Corcho. Ontological Engineering. *AI Magazine*, 36:56, 1991.

[5] D.L. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10:2004–03, 2004.

[6] A. Miles, B. Matthews, M. Wilson, and D. Brickley. SKOS Core: Simple Knowledge Organisation for the Web. In *Proceedings of the International Conference on Dublin Core and Metadata Applications*, pages 12–15, 2005.

[7] S. Pepper. The TAO of Topic Maps: Finding the Way in the Age of Infoglut. In *Proceedings of XML Europe 2000 Conférence*.

[8] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. pages 30–43. 2006.

[9] Alberto Manuel Simões and José João Almeida. Library::* — a toolkit for digital libraries. In *ElPub 2002 - Technology Interactions*, 2002.

Acknowledgments

This work was partly supported by project CROSS (PTDC/EIA-CCO/108995/2008), funded by the Portuguese Foundation for Science and Technology.

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

COMPETE

QREN QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL

UNIÃO EUROPEIA
Fundo Europeu de Desenvolvimento Regional