Weaving OML in a General Purpose Programming Language

Nuno Carvalho[1], José João Almeida[1], and Alberto Simões[2]

[1] Departamento de Informática, Universidade do Minho
[2] Centro de Estudos Humanísticos, Universidade do Minho
{narcarvalho,jj}@di.uminho.pt, ambs@ilch.uminho.pt

Abstract. Most existing programming languages can be categorized as general purpose programming languages, meaning that they can be used to implement solutions for any given domain. They are not, in any way, optimized for a specific set of problems. In contrast, Domain Specific Languages (DSL) are used to solve specific problems in a well defined domain. DSL are optimized to a particular set of problems, but they lack support for a wide range of operations that are required when dealing with real world problems. So, in a perfect world, we would like to implement applications using a general purpose programming language, but use a set of different DSL to handle specific domains' tasks.
In this paper we describe a DSL named Ontology Manipulation Language (OML) [3], designed to describe operations over with ontologies [9]. Programs can be written using only the OML syntax and be executed independently. OML syntax was designed to deal with ontologies and the language itself is optimized to perform these tasks, which means that other relatively simpler tasks can not be easily done. To overcome this challenge a mechanism was developed so that you can weave small snippets of OML code inside Perl programs, meaning we have the power of OML to manipulate ontologies and, at the same time, all the paraphernalia of modules that Perl offers to handle everything else.

1 Introduction

Since programs exist, many flavors of languages and different paradigms, have been devised and adopted. Before starting to write a program, one valid and pertinent question is which language to choose or paradigm to use. Most of the times a general purpose programming language is adopted. These languages are general purpose in the sense that they were not designed to solve problems in any particular domain and can, theoretical, be used to perform any task [1].

This is a positive attribute, but what can quickly became obvious is that since these languages are so general and act a bit as a swiss army knife, they quickly become inefficient for some specific problems. Inefficient not only performance wise but also linguistic wise, i.e. a lot of complex code is required to perform a rather simple task in a specific domain. To overcome this inefficiency, Domain Specific Languages (DSL) [12, 10, 14] are usually used. These languages are designed and optimized to solve particular sets of problems, meaning that writing programs to solve real world problems can be difficult or even impossible.

The optimal solution would combine both approaches, programs could be in their grand majority written using a general purpose programming language, taking advantage of their usual tools and libraries, and glued together small snippets of code written in a well defined set of (small) DSL to perform tasks in very specific domains. This would ensure language efficiency throughout the code. But the compiler would have to cope with different languages in the same program, which means that probably different parsers will be needed, different intermediate representations will be used, or even different compilation workflows would be required. Some well known problems emerge when several programming languages are mixed together in a single program, for example: parsers need to handle more than one language efficiently.

OML is a DSL that can be used to describe operations that manipulate information represented in an ontology. There are many different ways to define and represent an ontology, this work assumes a rather simplistic definition: a set of terms and relations between these terms. More details on these definitions can be found in [2]. OML is a rule-based programming language, and a program is a set of rules that are executed in order. A rule uses as its left hand side a pattern that should be searched in the ontology, and as its right hand side, an action to be performed. Patterns describe information, terms, relations between terms, or any combination of these that can be found in the ontology. Execution blocks describe the arbitrary operations that are to be executed when patterns are found, this includes not only operations on the ontology itself (changing its terms, relations, etc.), but also any arbitrary side effect producing operation required.

Although many interesting programs can be written using only OML, real applications can be hard or impossible to implement. Given that this language was designed to solve problems in a very specific domain—ontologies—it does not provide any syntax or methods to perform many simple tasks required to solve real world problems.

This was the motivation to the weaving of OML snippets in the Perl programming language, allowing to implement a program in Perl, using its typical tools and libraries for performing tasks, but allowing the user to describe operations related with ontologies in OML.

To better explain OML and its weaving into Perl, we will start by discussing OML syntax and semantic in section 2 and its compiler in section 3. This will allow the reader to understand how the DSL works, and therefore, be able to better follow the discussion about its weaving into the Perl programming language (section 4). Section 5 will present some simple tools that were built based on the ability to mix these two languages and finally, in section 6 we will confer about OML relevance, and the advantages that arose from its weaving with a general purpose programming language.

2 OML Specification

This section briefly describes OML syntax and how semantic actions can be defined to produce different type of results. OML is a simple language. One of the major goals during its design was to make sure that it would be easy and intuitive to learn and use, even for people without any programming skills background.

OML programs are a sequence of statements (or rules) which are executed in order. Each rule consists of a *pattern* block, the left side of the *fat-arrow* operator (=>), and an *action* block, everything on the right side. Also, each rule terminates with a single dot.

$$program \rightarrow rule^{+}$$
$$rule \rightarrow Pattern \texttt{ => } Action \texttt{ .}$$

The following two subsections describe the syntax for patterns (section 2.1) and the syntax and semantics for action blocks (section 2.2).

2.1 Patterns

Patterns are used to describe information in the ontology. Typically some action needs to be performed when a specific pattern is found. Patterns can be used to represent simple terms, relations between terms, or any combination of these. Table 1 illustrates some patterns that can be written in OML[1].

#	Pattern	Explanation
1	`term(Buster)`	term *Buster*
2	`rel(ISA)`	relation *ISA*
3	`term($t)`	for all terms
4	`rel($r)`	for all relations
5	`Buster ISA cat`	a specific relation
6	`$pet ISA cat`	any pet that *is a cat*
7	`$pet ISA $animal`	any two terms related by *ISA*
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, illustrated in table 1, will match every term in the ontology named `Buster` (in fact, will match just one term, as our ontology definition currently does not support more than one concept identified by the same preferred term). A specific relation can also be searched, using pattern 2, it will match all concept pairs that are related with the `ISA` relation.

OML supports variables, mostly like Prolog. Variables can be used instead of terms or relations names. Pattern 3 searches for all the terms in the ontology and pattern 4 searches for all the relations.

[1] Note that there is no rule forcing relations to be written in uppercase. We just use that convention for clearness.

More interesting patterns can be obtained describing facts: relations between terms. A simple example of a pattern that describes a fact is pattern 5. It will match if the terms `Buster` and `cat` are linked by a relation named `ISA`.

Variables containers can also be used in patterns, which means that the pattern can match more than once for a given ontology. Pattern 6 is one possible example, that matches all terms related to `cat` by the `ISA` relation (that is, all concepts that are cats). Another example of using variable containers is pattern 7, representing all the possible combinations of facts relating terms using the `ISA` relation.

Patterns can be grouped together using the logical binary operators `AND` and `OR`, which have their traditional meaning. Patterns paired with the `AND` operator will be evaluated as found if both patterns match. Patterns paired with the `OR` operator will be evaluated as found is one of the patterns match in the ontology. 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 if that 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:

- *a built-in action*
  An operation from the predefined list of operations available. These include the common ontology manipulation methods, as adding or removing facts, terms or relations;
- *generic code*
  OML is able to interpret Perl code on action blocks. This way OML gives some power to the user without the need to weave OML into a general purpose programming language. Therefore, the user can write any code to produce any arbitrary side effect, from updating a database, to creating a PDF file.

Built-in Actions
An example, of using a pre-defined action block 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 matched, 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 that matches some specific pattern. The variable `$pet` is automatically replaced with the term (or relation) that matched.

Generic Code
As already mentioned, we can also produce any kind of 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 it needs to be executed as is. At the current time this block needs to be written in the programming language Perl [5].

Having the ability to run Perl code we can produce any side effect, for example adding information to a relational database:

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

This was just a brief overview of what can be written using OML. You can find a exhaustive description of the language in [2]. OML can be compared with similar work in the same area, OPPL [6] is a language for transforming ontologies written using OWL [11]. Although both languages are based on a pattern approach, OML has a much clear and concise syntax, but OPPL provides a greater range of operators that allow more complex transformations.

3 The OML Compiler

OML stand-alone compiler (or probably, more correctly, interpreter) architecture is illustrated in figure 1. Keep in mind that in this section we are talking about the OML specific compiler, which we feed a program completely written in OML.

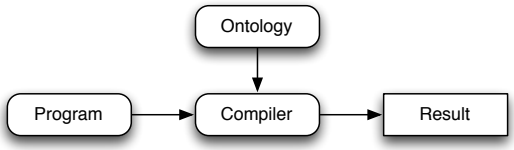

Fig. 1. OML compiler architecture overview.

Besides the program itself, written in OML, an ontology is also required as input. Currently there is only one backend for ontology processing available, this implies the use of ISO 2788 to represent the ontology [4, 13]. Of course other backends can be implemented to allow ontologies stored in other formats. The compiler will interpret the rules in the program, try to match the patterns described against the ontology, and act accordingly with found matches and described actions producing an arbitrary result.

Although this compiler can be used by itself, its main purpose is to be used by other tools, like the weaving of OML into Perl described in the next section.

The work performed by the compiler when executing a program is divided in three main stages (illustrated in figure 2):

Parsing Stage
In this stage the OML *parser* is responsible for analyzing the source program and create a parsing tree (`pTree`). This tree contains the same information that is in the source program but in a structured way. A typical yapp based parser was implemented to perform this task [7].

Expansion 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 them looking for the needed information in the ontology, and storing possible variations of the pattern being searched for. All the instances of the pattern found are stored in a tree called *domain instantiated tree* (`diTree`).

Reaction Stage
Finally, the *reaction* engine is responsible for executing the actions described in the program. This engine uses the `diTree` to instantiate the variables found in the action blocks of each statement, and executes the associated methods or code.

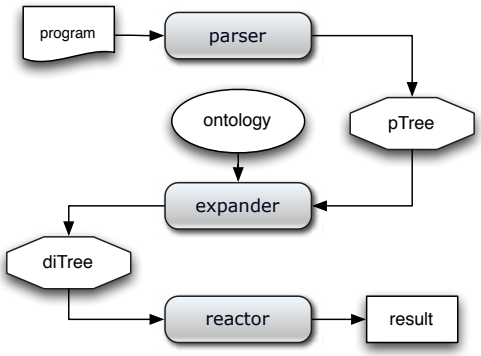

Fig. 2. OML compiler architecture overview.

This compiler is freely available for download [2] and use. Full documentation and example applications can also be found in the package. It is fully implemented in Perl and is ready to use OML programs to build full featured applications. Implementation and design details can be found, as already pointed out, in [2].

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

4 Weaving OML in Perl Programs

In this approach we consider one language to act as the host language, and the other as the hosted language: the DSL is the hosted language (OML in this specific case), and the general programming language is the hosting language (Perl in this specific case).

Instead of trying to handle several languages parsing and execution flows, we transform code written in the hosted language in a compilation unit that can be recognized and called by the hosting language. And then use the compiler for this language to run the program, while keeping this transformation hidden from the programmer. Note that we do not completely rewrite the OML code in Perl, we just transform it so that is valid Perl syntax, and the tasks implemented in OML can be called by the Perl compiler.

The main advantage of this approach is that in the end, the program that is executed, is written in a single well known programming language, and no special compiler or external tools are required to execute it. This workflow is illustrated in figure 3.

Fig. 3. Language weaving workflow overview.

To achieve this we take advantage of a library available for Perl that allows to perform source code transformations to programs before firing up the actual Perl compiler that will perform the execution. Using this approach, a function called `Weaver` was implemented that is responsible for the program filter transformation in three steps:

1. Identify in the program source code snippets written in the specific programming language (for that we needed to define some syntax sugar so the programmer can easily separate OML from Perl syntax).
2. Transform these snippets, individually, in code that can be executed by the general purpose programming language compiler. Each of these snippets is transformed in a function.
3. Combine these individual transformations, required initializations and library loading back into the original program.

The first step is to search the original program for snippets of code written in OML. This can be easily done with a simple pattern match approach, since the OML code is clearly tagged in the program with a special syntax. The result of executing this function is a list of all the snippets of OML found in the program. The rest of the code can also be partitioned into snippets, although these do not require any transformation this detail can be useful later. The resulting list can be a list of pairs, where one element of the pair is the language in which the second element of the pair, the code, is written. We call the function responsible for doing this work the `split` function.

The next step of this approach consists on iterating over this list and converting the OML snippets of code in Perl code, using the corresponding compiler, in this case

the compiler described in the previous section. The smallest unit of code that we are processing here is a function, meaning that every snippet of OML converted will be realized in a function, the main difference is that this function will be written in Perl and will implement a call to the OML native compiler in order to execute the code in runtime. All these subroutines are declared in the main namespace of the Perl compilation unit, this will make sure that the task implemented in OML will be available in the entire program. The main magic trick is performed inside this newly created function, where a call to the OML compiler with required data is automatically created. The job of doing this transformation is performed by the `compiler` function. The result of this stage is a list of code snippets all implemented in Perl syntax.

The next and last stage of the overall process is to assemble a complete and syntax correct program. This is achieved by joining all the snippets together, plus adding some initialization and library loading code. Since the smallest compilation unit that we are using is a function the order in which the snippets is glued is not relevant in this specific case, but if another general purpose programming language was being used maybe this function would need to have other details in consideration. We call the function that implements this stage `combine`. The final program produced by this function is the actual program that is feed to the compiler for execution.

Putting everything together we can say that `Weaver` is a function that transforms a `program` into another `program`, the first one written in OML and Perl, and the second one written in Perl only. This high order function, besides the original `program` to process receives as arguments the `split`, `compile` and `combine` functions in order to process the `program` has described earlier. The method is summarized in the following algorithm:

Function Weaver(program,split,compile,combine) : program

Input: program – original program to process
Input: split – function that separates the code parts by language
Input: compile – tranforms OML into Perl
Input: combine – joins parts of code together

$parts \leftarrow split(program)$
forall the $(lang, code) \in parts$ **do**
  if *lang is DSL* **then**
    $code \leftarrow compile(code)$
  $push(parts2, code)$
return *combine(parts2)*

A closer look at this algorithm reveals other interesting details:

– First, if we consider the compiler function a finite function where the keys are DSL and the values are functions that are able to compile code from that specific language to Perl, little changes would be need to the algorithm for processing more that one hosted language in the source program.

– This algorithm is so generic that a careful implementation of this approach would allow to build a framework that could automatically allow the weaving of arbitrary languages inside a Perl program, as long as the described functions could be provided.
– Also this technique can be perfectly extended to other hosting languages besides Perl, as long as some particular features are available in this target language, the ability to do a simple processing task on the code before handling the execution to the compiler for execution, for example.

After considering these details we already started working on this framework to allow arbitrary programming languages to be mixed together in a single program without much additional effort from the programmer. But this new framework is still under development and will not be introduced in this particular article.

Before entering the next section, where a real word program using this technique is presented, a small and quite academic example follows, for better understanding of this transformation.

To write a Perl program using OML snippets, the only requirement is to use the module in which this was implemented [3]. To tell Perl we want to use a module, the `use` keyword is used. So, once we use this module we can immediately write code in OML and Perl mixed together.

We can write a simple program just to illustrate how this works. This program creates a `dot` notation file, used by the GraphViz[4] tool [8]. This tool, named `term2dot`, takes a term and an ontology and creates a graph that represents all the relations for that term. A simplified version of the code for this tool is shown in program 1.

Program 1 *term2dot*: Create a Graphviz dot file based on an ontology.

```
use Biblio::Thesaurus::ModRewrite::Embed;

my $term = $ARGV[0];
my $ontology = thesaurusLoad($ARGV[1]);

printTerms($ontology,$term);

OML printTerms(term)
   begin => sub { print "digraph new {"; }.
   term $r $t => sub { print "$term -> $t [ label = $r ]"; }.
   $t $r term => sub { print "$t -> $term [ label = $r ]"; }.
   end   => sub { print "}"; }.
ENDOML
```

Note that we are calling the function `printTerms` in our code, just as if it was a standard Perl function.

[3] Biblio::Thesaurus::ModRewrite::Embed
[4] http://www.graphviz.org

This approach has proved to be very useful while building larger applications, because it allows the use of typical Perl tools and frameworks to develop applications and use small and concise OML programs to handle ontology specific operations.

5 Application Example: OntoMap

We used the approach described earlier, of weaving two programing languages (OML and Perl) to implement several applications that have to deal with information stored in ontologies. We chose to present in this article one of them: OntoMap.

Ontomap is a very interesting application. It is built as a web site that manages points of interest in a map, storing all the relevant information in an ontology. Note that we are not presenting a novel application. Our claim, here, is the relevance of weaving different programming languages to more quickly and efficiently obtain a running application.

- Perl is handling all the typical tasks for creating web-sites: generating HTML, parsing URL for arguments, etc.
- OML is being used to retrieve and add new information into the ontology.

The weaving of programming languages details are all handled by the embedding mechanism described in section 4, and everything is completely transparent not only for the developer but also to the entire infrastructure supporting it. The programmer simply needs to decide in which language to write particular sections of the code.

There are two main components: the **interface**, a web site, that implements all the functionalities required for the end-user to interact with the application; and the **ontology** that is used to represent all the information required by the application (points of interest attributes, as their longitude, latitude, type, description, etc.).

OntoMap UI

The interface is acting clearly as a proxy between the user and the data stored in the ontology. To achieve this, many tasks need to be performed besides retrieving the information from the ontology. These tasks include the generation of the presented HTML, with all the required calls to draw the map, etc.

After the interface is built, the application draws all the requested points of interest in the map. This is where OML gets useful. To gather this information an AJAX request is made to the server, and the answer is built accessing the ontology using OML.

This is implemented in a simple OML snippet as is shown in program 2. Note how clean, simple and efficient the code is, and how both languages used are clearly separated in a very natural and elegant way. A function called `get_points` is defined in OML, that collects all the points in the ontology that have at least the latitude and longitude relations. The Perl part of the program just prints the required headers and the information gathered by OML in a suitable format.

There is a variable that is being passed to the function, named `filter`. It is used by the interface to filter the type of locations shown (filter locations by a specific type,

Program 2 *locations.cgi*: Answer AJAX request for points of interest's data.

```
use CGI;
use JSON;
use Biblio::Thesaurus::ModRewrite;

my $filter = param('FILTER') or 'ANY';
print header, "{ markers: ", get_points($filter), "}";

OML get_points(filter)
  $point LAT $x ∧ $point LNG $y ∧ $point ISA $filter
    ⇒  sub {
      print to_json({name=>$point, lat=>$x, lng=>$y});
    }.
ENDOML
```

like searching for castles). Location types are also represented in the ontology, which means that this extra filtering option is handled by the pattern defined in the OML code. The OntoMap interface is shown in figure 4. When the user clicks any point in the map, a new request will be sent to the application, asking for more detailed information on that particular location. Again, this is an AJAX call, and the results are presented in the interface.

Another interesting request that is fired up is the one shown on figure 4, that includes a graph of all the available relations for the given point in the interface. This graph is generated using the *term2dot* tool described earlier.

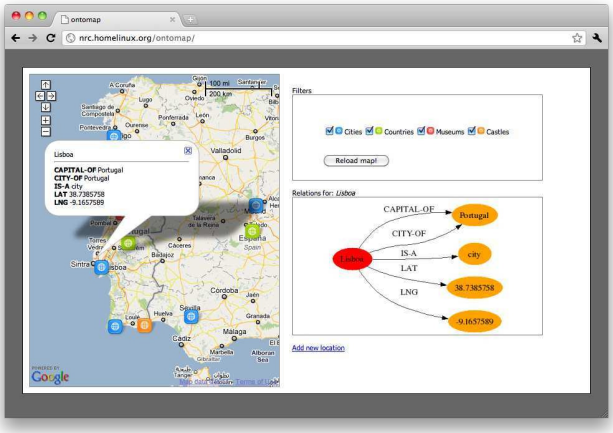

Fig. 4. OntoMap UI showing information about a specific location.

Producing Information

The application is not only responsible to serve information. It can also be used to learn new knowledge: the user is capable of adding new locations to the ontology. The information is collected via the interface and an AJAX request calls a program to add new knowledge to the ontology. To process this request a new program was created, using OML, to add the new data in the ontology. This is illustrated in program 3.

Program 3 *add_location.cgi*: Add new location to the ontology.

```
use CGI;
use Biblio::Thesaurus::ModRewrite;

my $name = param('NAME');
my $isa = param('ISA');
my $lat = param('LAT');
my $lng = param('LNG');

my $onto = thesaurusLoad('geo.iso');
add_location($onto,$name,$isa,$lat,$lng);
$onto->save('geo.iso');

OML add_location(name,isa,lat,lng)
  do ⇒ sub {
    add(name ISA isa) add(name LAT lat) add(name LNG lng)
  }.
ENDOML
```

In this program, Perl is only being used to retrieve the HTTP variables. The knowledge acquisition is all performed using OML. This snippet uses the special keyword `do` that executes the actions block without the need of searching for a pattern. In this action block, we are using the built-in function `add` to add the new term and corresponding relations to the ontology.

As has been shown, the entire application was built with small modular and elegant programs, written in both Perl and OML, that are easy to read and maintain. And can still be enriched or composed in different workflows to add more features to the application. These were the main goals that we were aiming to when adopted the usage of OML weaved inside Perl programs to implement applications.

6 Conclusion

General purpose programming languages are used to write programs, that describe how a set of tasks can be performed and used to solve an arbitrary problem. Since the syntax of these languages is intended to be as general as possible, the process of writing code to deal with very specific domains can be complicated and generate hairy code.

A DSL is an excellent approach for devising a language to solve problems in a very specific domain. But a problem arises: when using a DSL, most trivial tasks for a general purpose programming language can not be described in their syntax.

So, a possible and excellent solution is to write programs in general programming languages and use one or more DSL for describing tasks for specific domains. This approach would produce programs that are written in a mix of several languages.

In our particular case, we designed a very simple way for embedding small OML programs in Perl source code, without the need of any change to the Perl compiler itself. This was achieved by adding an extra step to the normal Perl execution workflow that takes a program written both in OML and Perl, and transforms it in a program written entirely in Perl, that can be normally executed by the Perl compiler. This extra step is completely transparent to the user, which simply writes the code and fires up the normal Perl compiler.

This approach provided an elegant and simple way of implementing applications that make use of ontologies, because OML was used to describe the tasks that have to deal with knowledge stored in the ontology. The application described in the previous sections of this article is a good example of programs that perform operations on ontologies and take clear advantage of the synergy between the two languages.

Further work is needed mostly on the OML language than in the languages weaving. OML includes methods for most ontology manipulative tasks, but misses some features and needs speed improvements.

Regarding languages weaving, further work would be interesting to analyze how the weaving of well known DSL languages into Perl or other general programming language can be obtained. This includes further developments to the general weaving mechanism described in section 4.

Acknowledgments

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

We would like to thank the reviewers for their valuable insight and detailed comments, which aided in improving this paper.

References

1. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic Programming. *Advanced Functional Programming*, pages 28–115, 1999.
2. N. Carvalho. OML - Ontology Manipulation Language. Master's thesis, University of Minho, 2008.
3. Nuno Carvalho, Alberto Simões, and José João Almeida. Oml: A scripting approach for manipulating ontologies. In *CISTI'11 - 6ª Conferência Ibérica de Sistemas e Tecnologias de Informação, Chaves, Portugal*, June 2011. (forthcoming).
4. ISO2788 Documentation. Guidelines for the establishment and development of monolingual thesauri, 1986.
5. M.J. Dominus and ScienceDirect (Online service). *Higher-order Perl: Transforming Programs with Programs*. Morgan Kaufmann Publishers, 2005.

6. M. Egana, E. Antezana, and R. Stevens. Transforming the axiomisation of ontologies: The ontology pre-processor language. *Proceedigns of OWLED*, 2008.
7. C. Frenz. *Pro Perl Parsing*. Apress, 2005.
8. E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Journal of Software — Practice and Experience*, 30(11):1203–1233, 2000.
9. A. Gómez-Pérez, M. Fernández-López, and O. Corcho. Ontological Engineering. *AI Magazine*, 36:56, 1991.
10. Tomaz Kosar, Pablo Martinez Lopez, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.*, 50(5):390–405, April 2008.
11. D.L. McGuinness, F. Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10:2004–03, 2004.
12. M. Mernik, J. Heering, and T. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316 – 344, 2005.
13. Alberto Manuel Simões and José João Almeida. Library::* — a toolkit for digital libraries. In *ElPub 2002 - Technology Interactions*, 2002.
14. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.