Transforming the Axiomisation of Ontologies: The Ontology Pre-Processor Language

Mikel Egaña[1], Robert Stevens[1], and Erick Antezana[2,3]

[1] Computer Science, University of Manchester
[2] Department of Plant Systems Biology, VIB, Gent, Belgium
[3] Department of Molecular Genetics, Ghent University, Gent, Belgium
mikel.eganaaranguren@cs.man.ac.uk
robert.stevens@manchester.ac.uk
erant@psb.ugent.be

Abstract. As ontologies are developed there is a common need to transform them, especially from those that are axiomatically lean to those that are axiomatically rich. Such transformations often require large numbers of axioms to be generated that affect many different parts of the ontology. This paper describes the Ontology Pre-Processor Language (OPPL), a domain-specific macro language, based in the Manchester OWL Syntax, for manipulating ontologies written in OWL. OPPL instructions can add/remove entities, and add/remove axioms (semantics or annotations) to/from entities in an OWL ontology. OPPL is suitable for applying the same change to different ontologies or at different development stages, and for keeping track of the changes made (e.g. in pipelines). It is also suitable for defining independent modelling macros (e.g. Ontology Design Patterns) that can be applied at will and systematically across an ontology. The presented OPPL Instruction Manager is a Java library that processes OPPL instructions making the changes to an OWL ontology. A reference implementation that uses the OPPL Instruction Manager is also presented. The use of OPPL has been demonstrated in the Cell Cycle Ontology.

1 Introduction

The use of OWL ontologies is rapidly increasing, especially in areas such as bioinformatics. As ontologies are more widely used, more tools are needed to fulfill the requirements of new users. One of those requirements is an abstract, straight-forward, high-level language for manipulating ontologies in a re-usable and efficient way.

This is particularly necessary when many of the ontologies currently written in OWL are axiomatically lean and increased computational inference will only arise with increased axiomatisation. For example, in many bio-ontologies, much of the ontology's semantics are bound up in the term names (`rdfs:label`) and these need to be made explicit so that reasoners can use those semantics. The Gene Ontology (GO) [1], one of the most used bio-ontologies, is a

good example of such a problem. In GO, we can find classes with labels like `alanine:sodium symporter activity` but with only some `is-a` and `part-of` relationships that can hardly be exploited by a reasoner. However, we can add new axioms based in the label; we can add, for example, the axiom `transports only (alanine or sodium)` and exploit that axiom in querying and structural maintenance[4]. The large size and repetitive nature of much of this axiomatic enrichment (for example, many term names share a similar syntactic structure [2, 3]) mandates the use of some form of transformation language that can be used to define reusable transformations.

Another justification for such a language comes from the fact that there are bio-ontologies that are built using automatic procedures (e.g. pipelines). For example, the Cell Cycle Ontology[5] (CCO) [4] is generated as a result of gathering data from existing ontologies and databases using a pipeline. Each version of CCO is generated automatically; any extra axioms that need to be added to enrich the existing knowledge can not be added by hand (they would be overwritten). Therefore, an OPPL implementation has been integrated into the CCO pipeline (see section 4), so the needed enrichment is defined as a set of OPPL instructions that are automatically applied in CCO.

A general justification for a high level language is based in how OWL ontologies are currently manipulated by the user. OWL ontologies can be manipulated in different ways. The most obvious and common method is via editors such as Protégé[6], but such manipulations are not reusable: a user makes changes through a graphical interface, and if another user wants to recreate the changes, they will need to be applied step-by-step (Swoop[7] allows for change sets to be reused by different users, but the ontology needs to be loaded and changes applied each time). Another method is to manipulate the ontology programmatically via APIs such as the OWL API[8]. When interacting with the ontology programmatically, the manipulations are reusable, but such access can only be performed by an API-familiar programmer, and each change of the ontology can represent a large amount of programming effort. The Protégé script tab[9] offers some level of abstraction but still full programming knowledge is required. Therefore a high level language for defining reusable actions is required.

The Ontology Pre-Processor Language[10] (OPPL) fulfills the requirements cited above. OPPL offers an abstract and straight-forward syntax that can be used to manipulate OWL ontologies. The OPPL manipulation actions can be easily re-used in different OWL ontologies, at different stages and by different users, offering most of the expressivity and re-usability of an API-level access to the ontology, with minimal notions of programming required. The intended

[4] The Gene Ontology Next Generation (GONG) workflow does precisely that: http://www.gong.manchester.ac.uk/
[5] http://www.cellcycleontology.org
[6] http://protege.stanford.edu/
[7] http://code.google.com/p/swoop/
[8] http://owlapi.sourceforge.net/
[9] http://www.med.univ-rennes1.fr/~dameron/protegeScript/
[10] http://oppl.sourceforge.net/

audience of OPPL is formed by ontology curators who need to do flexible and automatic ontology building but without necessarily having a strong computational background.

2 OPPL syntax

The OPPL syntax is based in the Manchester OWL Syntax [5], with some extensions: mainly the keywords `ADD`, `REMOVE` and `SELECT`. The OPPL syntax is case sensitive.

The central unit of the OPPL syntax is the so-called **OPPL instruction**. It describes one or more actions to be performed upon an entity or groups of entities[11]: each action, or **OPPL statement**, is delimited by a semicolon (`;`). See Fig. 1 for details.

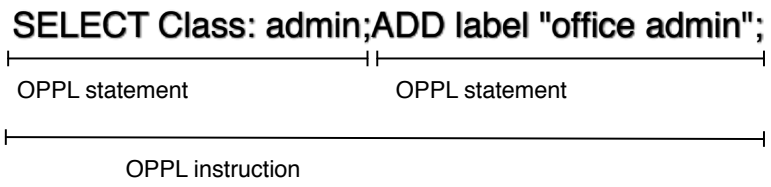

Fig. 1. An example of an **OPPL instruction**, composed of two **OPPL statements**.

There are two types of OPPL instructions which are explained in detail in subsections 2.1 and 2.2.

2.1 Single statement OPPL instructions

This OPPL instruction is formed by only one OPPL statement that adds/removes an entity to/from an ontology. For example, if we want to add a class named `undergraduate` to the ontology, we would use the following instruction:

```
ADD Class: undergraduate;
```

To remove a class from the ontology, we would use, for example, the following instruction:

```
REMOVE Class: undergraduate;
```

The OWL API (which is used by OPPL to access the ontology, see section 3) only allows the addition of axioms, in conformance to OWL semantics. As a result, it is not possible to add a class *per se*, instead, only an axiom

[11] An entity is assumed to be a named class, a named individual, or an object property. Currently OPPL does not support data properties nor datatypes.

that references the class can be added. Therefore the cited OPPL statement (`ADD Class: undergraduate;`), when processed, adds an axiom stating that the class `undergraduate` is a subclass of `owl:Thing`. This assumption is made to make the OPPL syntax more simple. A `REMOVE` statement deletes the axioms that reference entities, not the entities as such.

2.2 Multiple statements OPPL instructions

This type of OPPL instruction is composed of at least two OPPL statements. The first statement is always a `SELECT/ADD` statement, followed by one or more `ADD/REMOVE` statements.

In the case that the first statement is a `SELECT` statement, it selects entities from the ontology and the following `ADD/REMOVE` statements add/remove axioms (semantic axioms such as restrictions or annotations) to/from the selected entities. The selection is made according to a condition, which can be a semantic axiom (e.g. having a concrete restriction like `part-of some nucleus` as a necessary condition) or an annotation value. Any entity that matches the condition will be selected and the next `ADD/REMOVE` statements will be applied to it. For example:

```
SELECT equivalentTo participates_in
only (intellectual_dinner and party);
ADD label "professor";
REMOVE subClassOf lives_on only (not campus);
```

This instruction does the following: selects any class that has the necessary and sufficient restriction `participates_in only (intelectual_dinner and party)`. Then, adds the `rdfs:label` "professor" to it. Finally, removes the necessary restriction `lives_on only (not campus)` from it.

Annotation values can be used to select entities, and regular expressions support for annotation values[12] is included for this purpose. In the following example, any class in which the `rdfs:label` matches the regular expression "(.+) (development)" (e.g. "cell development") will be selected. The selected class(es) will become subclass(es) of `development`, and the necessary restriction `acts_on some cell` will be added (`<1>` refers to the first group of the matching string):

```
SELECT label "(.+) (development)";
ADD subClassOf development;
ADD subClassOf acts_on some <1>;
```

The `SELECT` statement need not be a condition to fulfill; a single entity can be selected:

[12] The regular expressions are Java style:
http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html

```
SELECT Class: admin;
ADD label "administrator";
```

Conditions can also be defined to select object properties:

```
SELECT inverse participates_in;ADD range student;
```

In the case that the first statement is an `ADD` statement (instead of `SELECT`), an entity will be added and the next `ADD`/`REMOVE` statements will be applied to it. For example:

```
ADD Class: professor;
ADD label "staff";
ADD equivalentTo participates_in only
(intellectual_dinner and party);
```

This OPPL instruction adds a class `professor` first. Then, it adds the `rdfs:label` "staff" to it. Finally, it adds the necessary and sufficient restriction `participates_in only (intelectual_dinner and party)`.

2.3 OPPL syntax design

The development of the OPPL syntax has been driven by the authors' daily needs while working with bio-ontologies (mainly enrichment of the Gene Ontology in the GONG workflow and enrichment of CCO), and therefore some features that would make the OPPL syntax more complete have been left aside as they did not provide an immediate benefit. However, there are pointers that OPPL could follow to provide a more complete syntax. For example, more programming-like features (loops, conditional control, subroutines, etc.) would be desirable. A `SET` statement that simply changes axioms (instead of only removing or adding them) will probably also be part of the future OPPL syntax. Some segmentation capability is also needed: for example, the ability to extract parts of an ontology to introduce them in a new ontology.

For the same reason, there has also been an equilibrium between the needs of the authors to introduce more and more tailored instructions and the aim to keep OPPL simple and clean. Some examples of tailored instructions include the ability to distinguish between primitive/defined classes:

```
SELECT_PRIMITIVE descendantOf domain;
ADD label "primitive";

SELECT_DEFINED descendantOf domain;
ADD label "defined";
```

Another example of such tailored instructions is the `disjointWithSiblings` statement, that can be used, for example, to make all the classes of a subtree disjoint with their siblings:

```
SELECT descendantOf person;ADD disjointWithSiblings;
```

There are also two instructions (`assertedSubClassOf` and `assertedSuperClassOf`) that allow to query for super/subclasses without using the reasoner, which means that, for example, inconsistent classes can be selected:

```
SELECT assertedSubClassOf participates_in some sport;
ADD label "is this a student?";
```

The complete OPPL syntax can be learned by following the examples provided in the OPPL web site[13]. Other instructions not reviewed in this paper include `disjointWith`, `differentFrom`, `sameAs`, `type`, `descendantOf`, `ancestorOf`, `subPropertyOf`, etc.

3 OPPL design and implementation

The core OPPL is provided as an stand-alone Java library (OPPLInstructionManager) that can be used to write applications (Fig. 2). The OPPLInstructionManager processes each OPPL instruction that is provided to it and performs the changes in an OWL ontology chosen by the user. The changes to the ontology are made by the OPPLOWLManager class, a wrapper for the OWL API, when prompted to do so by the OPPLInstructionManager. The OPPLOWLManager performs all the actions related to OWL: accessing the model, querying the model via reasoners, changing the model, and loading or writing ontologies.

Each OPPL instruction is executed independently, therefore, if, for example, an OPPL instruction adds an axiom the following OPPL instruction can safely remove it. For the same reason, if an OPPL instruction removes an entity and a later OPPL statement of an OPPL instruction needs to operate upon it, the execution will fail, but the execution will continue with the next OPPL statements of that OPPL instruction. Similarly, if a `SELECT` statement is unable to select an entity, the rest of the OPPL statements in that OPPL instruction will not be executed.

The parsing of the Manchester OWL Syntax expressions is made by the parser provided by the OWL API, through the OPPLOWLManager, which returns an OWLDescription object (from the OWL API) that is used to query the reasoner. The user can choose which reasoner to use: Pellet[14], FaCT++[15] or any reasoner via the DIG interface[16]. Errors are flagged at different levels: OPPL syntax, reasoning, Manchester OWL syntax, OWL model changes, etc.

The OPPLInstructionManager is independent of the OPPL instructions provider, which is anything that implements the OPPLInstructionsProvider interface. In

[13] http://oppl.sourceforge.net/test.oppl
[14] http://pellet.owldl.com/
[15] http://owl.man.ac.uk/factplusplus/
[16] http://dig.sourceforge.net/

the provided reference implementation the OPPL instructions provider is a flat file parser (by convention, files with the suffix **.oppl** are used), but it would be simple to program any other OPPL instructions provider (e.g. to include the OPPLInstructionManager in another program). In the reference implementation, **.oppl** flat files allow comments starting with hash (#) and the OPPL instructions are divided by white lines (the OPPL instructions can be multi-line). See Fig. 3 for an example OPPL file.

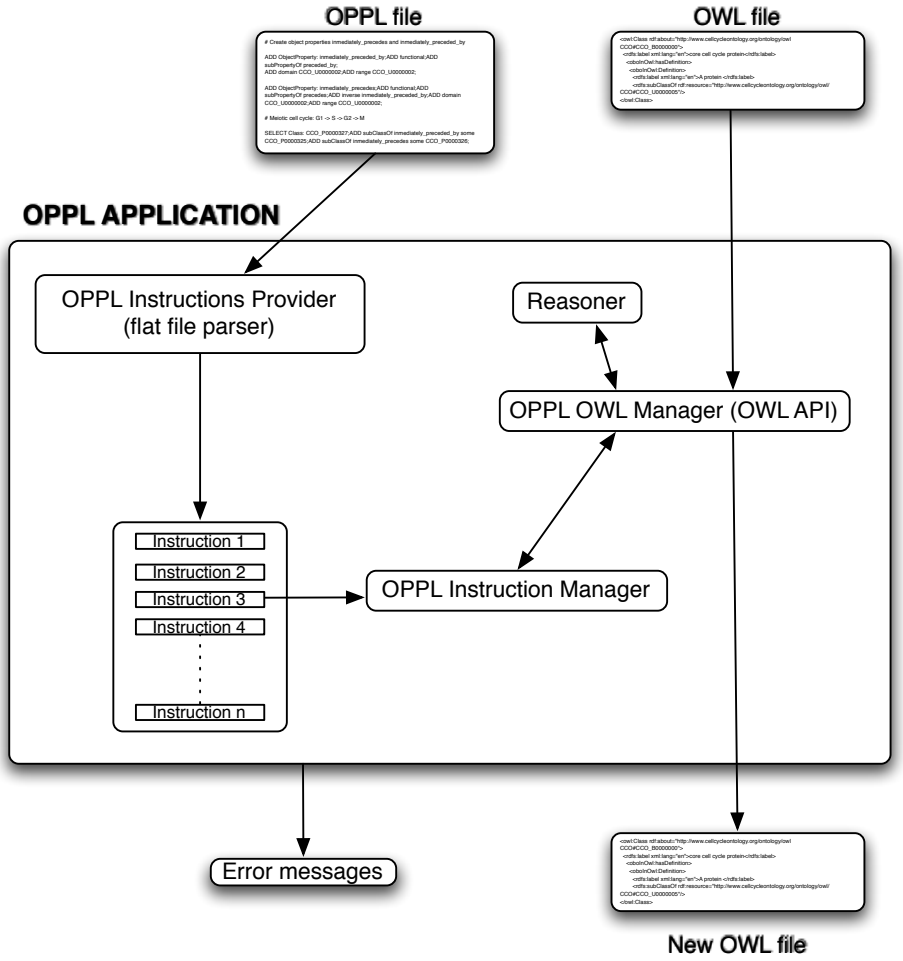

Fig. 2. Simplified design of the current OPPL implementation.

In the reference implementation, the OPPL application is executed through the command line: the path to the flat file with the OPPL instructions and the path to the OWL ontology are passed as arguments.

```
# Create object property inmediately_precedes

ADD ObjectProperty: inmediately_precedes;ADD functional;
ADD subPropertyOf precedes;ADD inverse inmediately_preceded_by;ADD domain
CCO_U0000002;ADD range CCO_U0000002;

# Meiotic cell cycle: G1 -> S -> G2 -> M

SELECT Class: CCO_P0000327;ADD subClassOf inmediately_preceded_by some
CCO_P0000325;ADD subClassOf inmediately_precedes some CCO_P0000326;

# Query 1: Proteins acting in the mitotic S phase (At)

ADD Class: query_1;ADD subClassOf query;REMOVE subClassOf Thing;
ADD comment "Proteins acting in the mitotic S phase";

SELECT subClassOf participates_in some (CCO_P0000014 or (part_of some
CCO_P0000014));ADD subClassOf query_1;
```

Fig. 3. An extract of an OPPL file applied to CCO.

4 Application on the Cell Cycle Ontology

The cell cycle is the process by which a new cell comes into existence and divides into two cells, and all the steps in the middle. The cell cycle is a very important research field of life sciences, as its malfunction is the cause of diseases such as cancer. The Cell Cycle Ontology gathers the current scientific knowledge about the cell cycle. This system composes five ontologies: an ontology for each considered model organism (*H. sapiens*, *S. cerevisiae*, *S. pombe* and *A. thaliana*) and a central ontology that includes the four ontologies plus relationships across proteins from the different model organisms. An automatic pipeline retrieves and manipulates data from different ontologies and databases that is finally included into the ontologies, thus, the five ontologies are created anew each time the pipeline is executed.

OPPL is used to add new axioms to the CCO ontologies. OPPL has been chosen because it is very inefficient to manually add the axioms to five newly created ontologies each time the pipeline is executed (they would be overwritten in the next execution) and may also be error prone. Therefore, OPPL flat files have been devised with some new enriching axioms[17] and the OPPL reference implementation is executed as part of the pipeline, adding the axioms to each newly created ontology (Fig. 3). The defined axioms can be regarded as independent "modelling modules" or "modelling libraries" (e.g. Ontology Design Patterns[18]) to be applied or re-used.

Using OPPL in CCO also means that the design decisions become explicit (the rationale behind each added axiom is documented in the flat files via comments) and flexible (OPPL instructions with very complex semantics can be

[17] ftp://ftp.psb.ugent.be/pub/cco/oppl/cco.oppl
[18] http://odps.sourceforge.net

tested and rejected/accepted by simply commenting/uncommenting lines on the OPPL flat files). OPPL can also be used for querying; sample queries against CCO are stored in OPPL flat files[19] and executed via the OPPL reference implementation (Fig. 3).

The fact that OPPL instructions could be applied on demand has eased the development and maintenance of the Cell Cycle Ontology. CCO is automatically built monthly which implies that a careful maintenance policy is needed, not only for keeping suitable identifiers, but also for enriching the semantics via a predefined set of OWL axioms. The implementation of that set of axioms as part of the CCO automatic building pipeline by means of OPPL has demonstrated many features of OPPL such as re-usability, modularity, and maintainability while dealing with huge and complex ontologies such as CCO. Currently, CCO has more than 54000 classes (more than 30000 proteins) which are connected by about 10 different types of properties resulting in a relatively highly connected network of concepts. Moreover, the file size of CCO composite ontology in OWL is over 90 MB which clearly suggests the need for a suitable tool like OPPL for modifying it automatically.

5 Conclusion

OPPL offers a straight-forward syntax for manipulating OWL ontologies. Using OPPL ontologies can be manipulated in a repeatable manner (for example OPPL complex instructions can be shared amongst users), and complex modelling can be done in a one-step fashion (define the axioms once, apply many times); OPPL increases the efficiency of ontology maintenance and makes development time shorter. OPPL has been used in the development of the Cell Cycle Ontology and has demonstrated its utility.

In a near future, the development of a Protégé plugin for enabling a user-friendly development with OPPL is expected, with functionalities such as auto-complete for writing OPPL instructions. In this way, the best of both paradigms (normal access through graphical interface and access through OPPL instructions) will be available in ontology development.

Regarding the syntax, role chains, data types and dataproperty support will be added. Regular expressions in class URIs will also be supported in the short term. In a longer term, a BNF grammar (once the syntax has become stable) and SWRL support are expected.

The provided OPPL Instruction Manager and reference implementation are licensed under the LGPL[20].

[19] ftp://ftp.psb.ugent.be/pub/cco/oppl/cco.query.oppl
[20] http://www.gnu.org/licenses/lgpl.html

Acknowledgements

Mikel Egaña is funded by the University of Manchester and EPSRC. Erick Antezana is funded by the European Science Foundation (ESF) for the activity entitled "Frontiers of Functional Genomics".

References

1. Ashburner, M., Ball, C.A., Blake, J.A., Botstein, D., Butler, H., Cherry, J.M., Davis, A.P., Dolinski, K., Dwight, S.S., Eppig, J.T., Harris, M.A., Hill, D.P., Issel-Tarver, L., Kasarskis, A., Lewis, S., Matese, J.C., Richardson, J.E., Ringwald, M., Rubin, G.M., Sherlock, G.: Gene ontology: tool for the unification of biology. The Gene Ontology Consortium. Nat Genet **25**(1) (May 2000) 25–29
2. Ogren, P., Cohen, K., Acquaah-Mensah, G., J. Eberlein, L.H.: The Compositional Structure of Gene Ontology Terms. In: Pacific Symposium on Biocomputing. Volume 9. (2004) 214–225
3. Egana, M., Wroe, C., Goble, C., Stevens, R.: In situ migration of handcrafted ontologies to reason-able forms. Data and Knowledge Engineering, in press
4. Antezana, E., Tsiporkova, E., Mironov, V., Kuiper, M.: A cell-cycle knowledge integration framework. In Leser, U., Naumann, F., Eckman, B.A., eds.: DILS. Volume 4075 of Lecture Notes in Computer Science., Springer (2006) 19–34
5. Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., Wang, H.H.: The Manchester OWL Syntax. In: OWL: Experiences and Directions 2006 Athens, Georgia, USA, November 10-11 2006. (2006)