Ontology Pre-Processor Language (OPPL)

User's manual

Mikel Egaña Aranguren (1), Luigi Iannone (2), and Robert Stevens (2)

(1) Ontology Engineering Group, School of Computer Science,
Technical University of Madrid (UPM), Spain

(2) Bio-Health Informatics Group, School of Computer Science,
University of Manchester, UK

July 4, 2011

Contents

1 Introduction

1.1 What is OPPL?

OPPL is a scripting language for automating the manipulating of OWL[1] ontologies: The user defines queries and changes to be performed on the entities returned by the queries. A change is the addition/removal of axioms to/from the entities retrieved by the query. The queries and changes are all defined in an OPPL script written following the OPPL syntax: when the script is executed, if the query returns entities from the ontology the changes will be applied to those entities. For example, the translation of an hypothetical OPPL script into natural language would read as follows: "Retrieve all the entities that have the axiom `subClassOf part-of some car` and add the axiom `subClassOf part-of only car` to them".

1.2 When should I use OPPL?

The use of OPPL is worthy in (At least) the following situations (And combinations thereof):

- Ontologies with a lot of entities.
- Ontologies with repetitive axiomatic structures.
- Ontologies with complex axiomatic structures (Complex modelling).
- Ontologies with repetitive annotation values.

The advantages of using OPPL in the mentioned situations (And in other situations that the user considers it to be useful) can be summarised as follows:

- Store modelling: very complex modelling structures can be stored and applied at will, saving time, since such application is not performed manually.
- Do/Undo: very complex modelling can be tested by executing a script.
- The same modelling can be applied consistently across the whole ontology.
- Modelling can be shared between developers and applied at will.
- The modelling process is explicitly recorded in scripts.

1.3 Installation and requirements

OPPL is open source and licensed under the LGPL[2]; it can be freely downloaded [3]. In order to use OPPL OWL knowledge is required, specially Manchester OWL Syntax (MOS) [1]. The Manchester Protégé OWL tutorial[4] covers most of the OWL knowledge that is needed to work with OPPL, including MOS.

[1] http://www.w3.org/standards/techs/owl
[2] http://www.gnu.org/copyleft/lesser.html
[3] http://sourceforge.net/projects/oppl2/files/
[4] http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/

OPPL can be used through a graphical interface as a Protégé[5] plug-in (At least Protégé 4.1 RC 4 is needed). Protégé is a widely used open source ontology editor that is able to work with the OWL or OBO[6] ontology languages. OPPL can also be used programmatically, through the OPPL Java API; the OWL API[7] is needed in that case.

1.4 Contact

The OPPL web page[8] provides documentation[9] and a collection of OPPL sample scripts[10]. Any questions regarding using OPPL should be addressed to the public mailing list[11].

[5] http://protege.stanford.edu/
[6] http://www.geneontology.org/GO.format.obo-1_4.shtml
[7] http://owlapi.sourceforge.net/
[8] http://oppl.sf.net
[9] http://oppl2.sourceforge.net/documentation.html
[10] http://oppl2.sourceforge.net/taggedexamples/
[11] oppl2-user@lists.sourceforge.net

2 Basic OPPL

The basics of the OPPL syntax will be shown using an example script; further details and keywords of the syntax are provided in Section 4. The example script manipulates the Cell Type Ontology[12] (CL). CL describes cell types in a hierarchy, including their development path with the `develops_from` relation (*e.g.* `Glioblast subClassOf develops_from some neuroectodermal_cell`). Such existential relation indicates that there is at least one `develops_from` relation, but there could be more; therefore, it might be interesting to close every `develops_from some` relationship to assert that each cell that develops from a given cell develops only from that cell. Thus, to add the axiom, in the case of Glioblast, `Glioblast subClassOf develops_from only neuroectodermal_cell`. The following OPPL script does the job:

```
1 ?target:CLASS,
2 ?origin:CLASS
3 SELECT
4 ?target SubClassOf develops_from some ?origin
5 BEGIN
6 ADD ?target SubClassOf develops_from only ?origin
7 END;
```

This script shows the main parts of an OPPL script, summarised as follows:

```
1  Variable declaration,
2  Variable declaration,
3  ...
4  SELECT
5  Query,
6  Query,
7  ...
8  BEGIN
9  ADD/REMOVE Axiom,
10 ADD/REMOVE Axiom,
11 ...
12 END;
```

Each section is explained as follows:

- Variables declaration (Lines 1–2): In this section the variables that will be used in the script (In the query and the changes) are defined. The variables represent the entities of the ontology that should be retrieved by the query. Variables start with the `?` mark, and they are strongly typed with the keyword `CLASS`, `CONSTANT`, `OBJECTPROPERTY`, `DATAPROPERTY`, `ANNOTATIONPROPERTY`, or `INDIVIDUAL`. In this case there are two variables: `?target` and `?origin`, both OWL classes (`CLASS`).
- Selection of entities (Starting with keyword `SELECT`, Lines 3-4): In this section a query against the ontology is performed, referring to the variables defined in the previous section. The query can be a DL (Description Logics) query or a regular expression to filter entities by their annotation values, *e.g.* `rdfs:label`. In this case a DL query is performed, thus the query `?target SubClassOf develops_from some ?origin` is executed using the automated reasoner: `?target` will be matched by `Glioblast` and other cells, and `?origin` will be matched by `neuroectodermal_cell` and other cells. Note that `develops_from` is an entity of the CL ontology.

[12] http://purl.org/obo/owl/CL

- Perform actions (Lines 5-7, between keywords `BEGIN` and `END`): In this section axioms are added or removed to/from the entities retrieved by the querying, if any. In this case a new axiom is added (`ADD`) to the retrieved entities (Axioms can also be removed with the `REMOVE` keyword). In the case of `Glioblast`, the following axiom will be added: `Glioblast subClassOf develops_from only neuroectodermal_cell`. Many axioms can be added and/or removed, but the variables should always be the ones defined in the variables declaration section.

3 Using OPPL with Protégé

In order to use OPPL, the OPPL Protégé 4.1 plug-in should be downloaded and placed in the plug-ins directory of the Protégé installation. There are two ways of using OPPL *via* Protégé: using the OPPL tab or the OPPL macros tab. If the plug-in has been installed correctly, both can be found in `Window >> Views >> Ontology views >> OPPL | OPPL macros`: the user chooses where to put them within the Protégé interface.

Before using any of the OPPL plug-ins an automated reasoner should be chosen and synchronised with the ontology. To synchronise the reasoner reasoning should be performed at least once, by going to `Reasoner >> Start reasoner` or by pushing `CTRL-R`. This does not apply in the case of the asserted mode (See Section 4.2).

3.1 OPPL tab

Using the OPPL tab the user builds an OPPL script using either the OPPL builder or the OPPL text editor (Figure 1). Once built, the script can be evaluated (`Evaluate` button) in order to see beforehand how its execution will affect the ontology, in the `Affected axioms` pane. The user can later execute the script, if happy with the consequences.

3.1.1 OPPL builder

The OPPL builder guides the user through the different steps needed to build an OPPL script (Figures 1, 2 and 3). The beginning of the script, thus the variables declaration, can be found in the left side: the variables can be for entities already present in the ontology (`Input variables`) or for entities that will be created when the script is executed (`Generated variables`). Further to the right, the `Select` pane can be used to define the selection section of the script, including constraints in the `Where` pane (See Section 4.3). Finally the actions can be defined in the `Actions` pane. If the OPPL script has syntactic errors they are flagged in the middle pane.

3.1.2 OPPL text editor

The plain text editor can be used to directly type an OPPL script or paste it (Figures 4 and 5). If the OPPL script is directly typed in, the autocomplete functionality can be exploited.

3.2 OPPL macros tab

The OPPL macros tab can be used to record the steps that the user performs when working with the ontology (Figures 6 and 7). Such steps are recorded in an OPPL script: Later, such script can be pasted in the OPPL text editor and executed normally. In order to use this method the user only needs to push the record button (The big red circle on the left) and perform the desired changes in the ontology. The recording can be stopped at any time and the changes performed till that moment will be written down to an OPPL script.

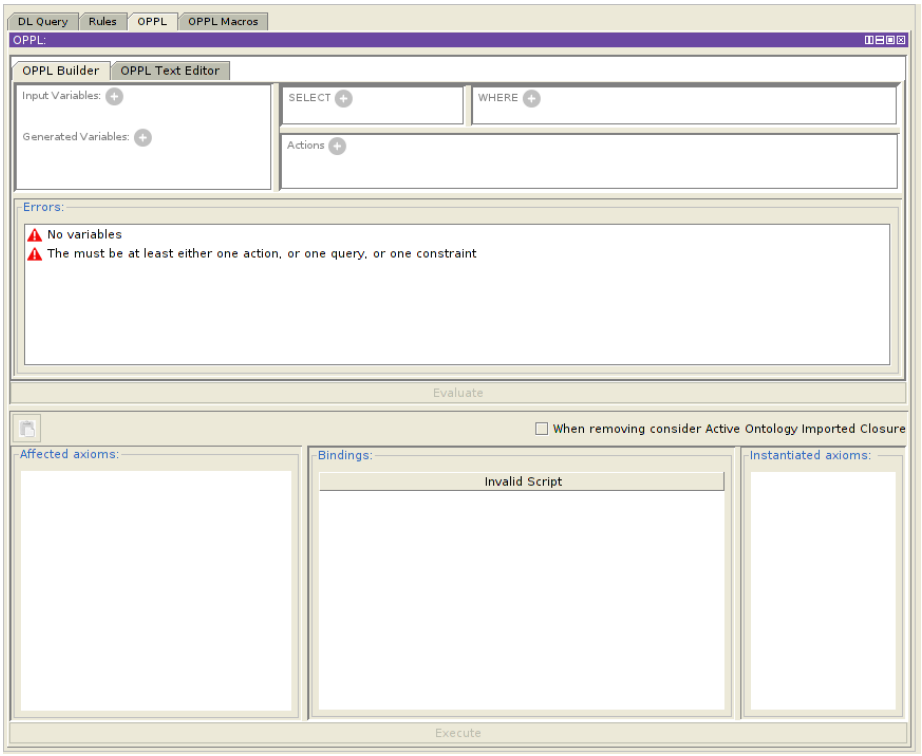

Figure 1: OPPL tab. The OPPL tab is further divided into the OPPL builder and the OPPL text editor. In this case the OPPL builder is selected.

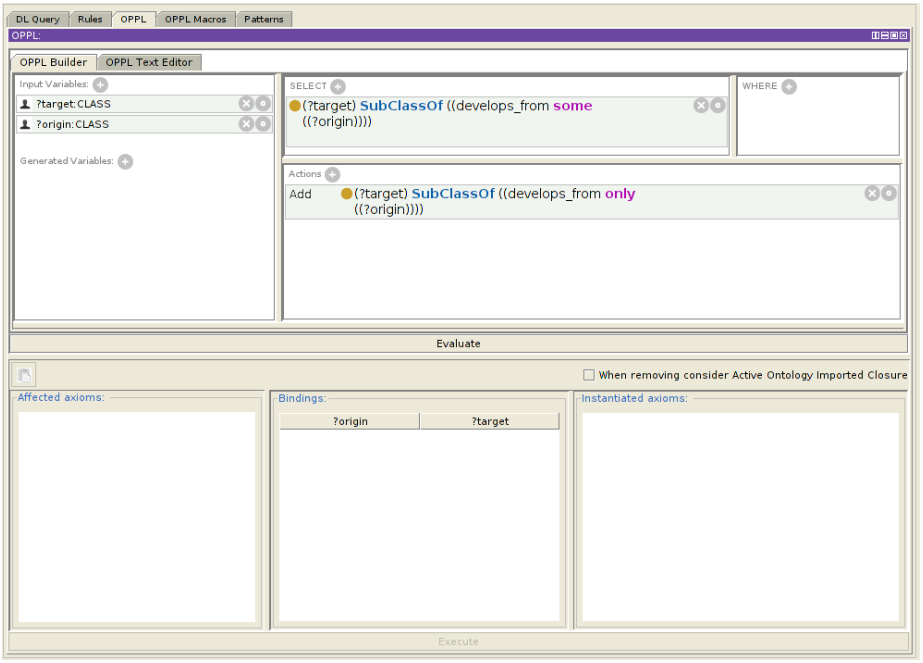

Figure 2: OPPL builder: the user has entered the example script from Section 2.

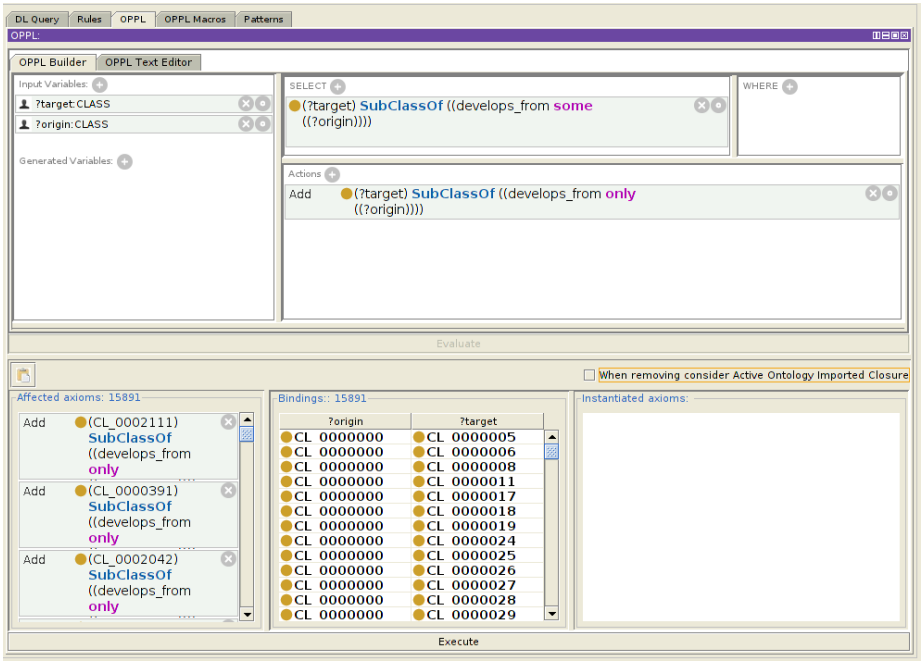

Figure 3: OPPL builder: evaluation of the example script from Section 2. The script can be executed afterwards by pushing the `Execute` button.

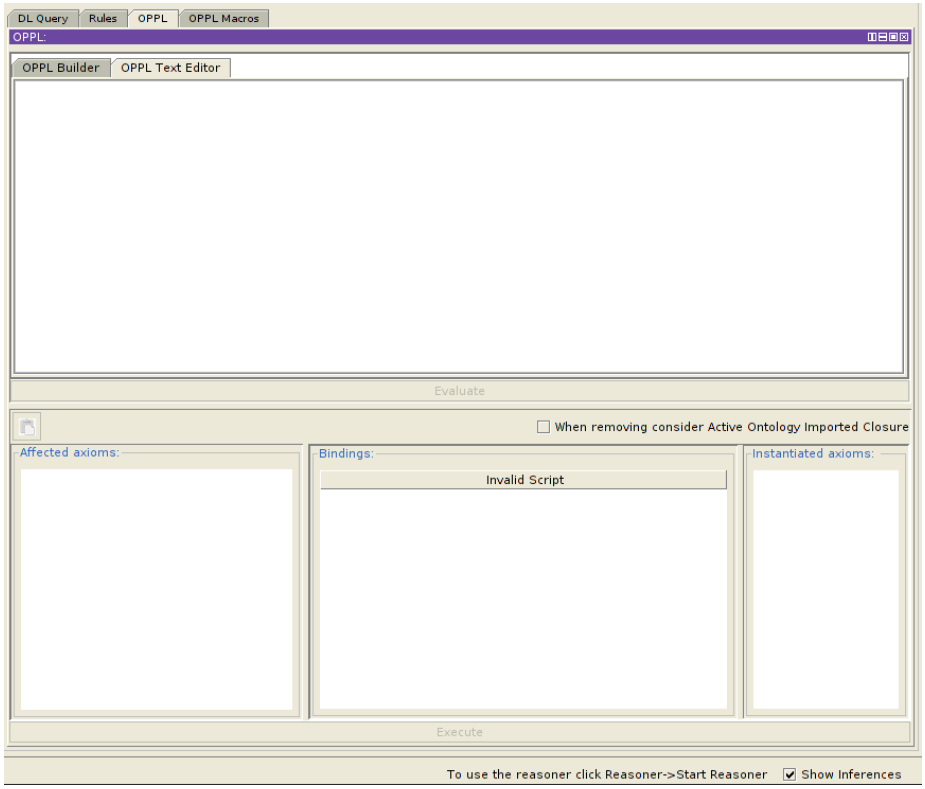

Figure 4: OPPL text editor.

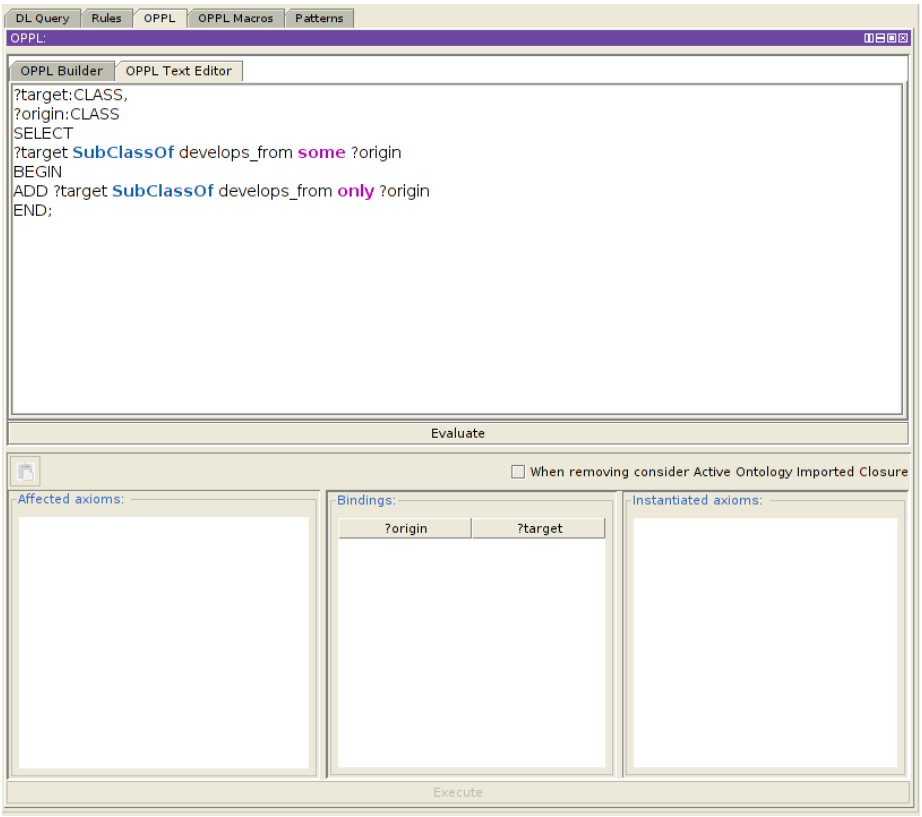

Figure 5: OPPL text editor: the user has typed the example script from Section 2. The evaluation and execution is exactly the same as in Figure 3.

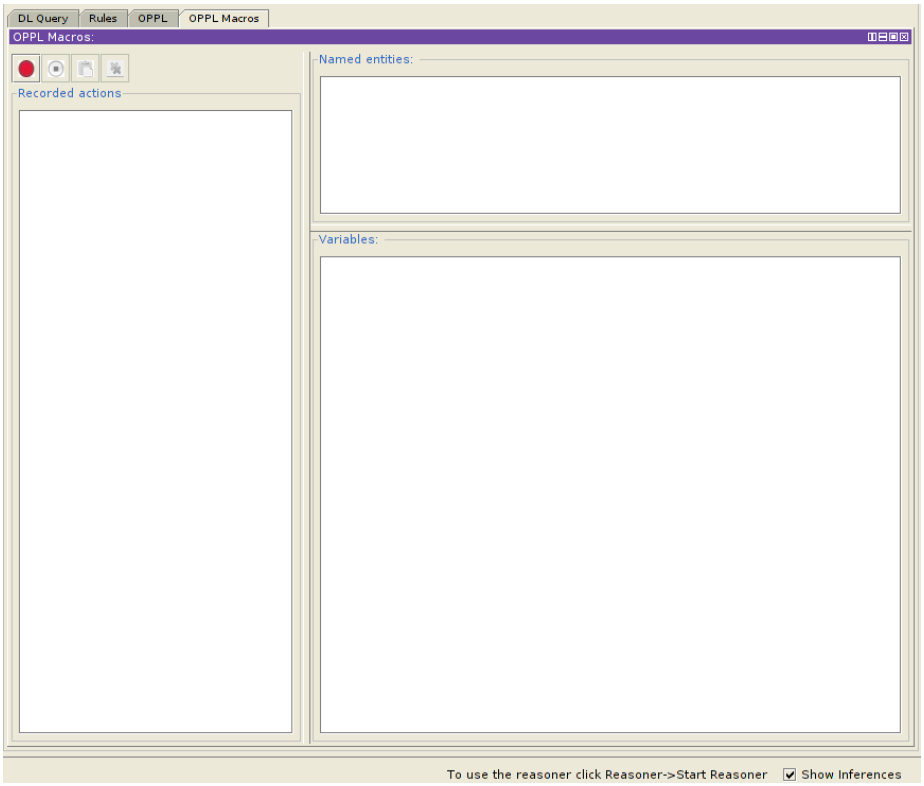

Figure 6: OPPL macros tab.

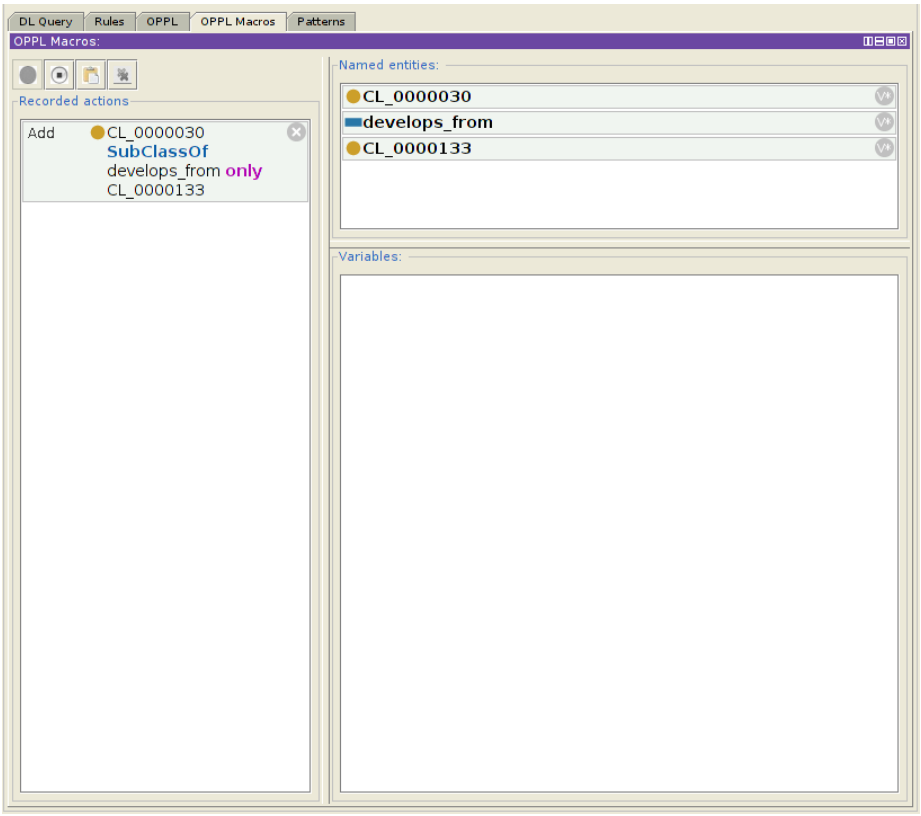

Figure 7: OPPL macros tab. Some actions have been recorded: if the user pushes the `Stop recording` button, the recording will stop and the actions will be available to be copied to an OPPL script, by pushing the `Copy OPPL` button (The script can be pasted in any plain text editor).

4 Advanced OPPL

This section expands the concepts introduced in Section 2 to cover the whole OPPL expressivity.

4.1 Working with variables

The whole syntax for working with variables is depicted in Figure 8 and expanded in the following subsections.

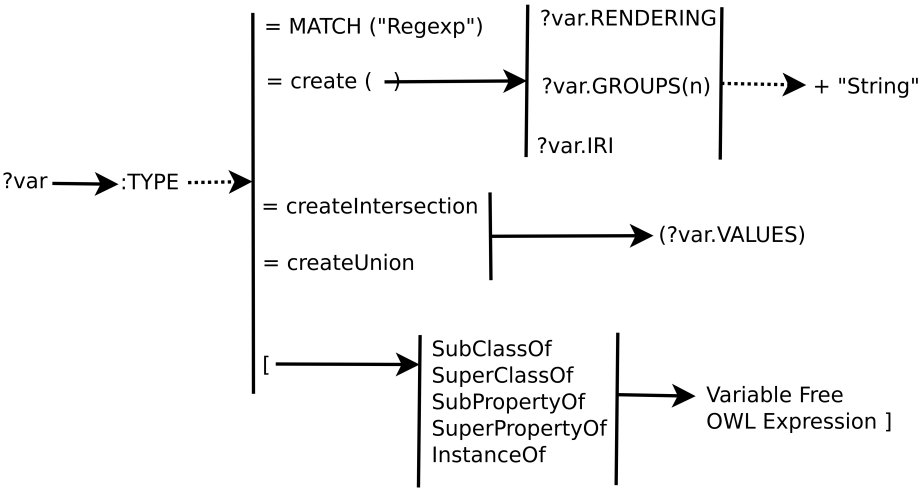

Figure 8: Graphical representation of the OPPL syntax for variables. Solid lines indicate compulsory elements, dotted lines optional elements. Vertical lines indicate different options on the same element. `?var` indicates any variable. `TYPE` should be one of `CLASS`, `CONSTANT`, `OBJECTPROPERTY`, `DATAPROPERTY`, `ANNOTATIONPROPERTY`, or `INDIVIDUAL`. `Variable Free OWL expression` is a pure OWL expression, with no OPPL variables or keywords, like `hasPart some (partOf only body)` or `Pizza`.

4.1.1 Matching strings (`MATCH`)

Regular expressions can be matched against the `rdfs:label` value of entities with the `MATCH` keyword. Such regular expressions follow the Java syntax for regular expressions. For example, the following script[13] selects all the classes whose `rdfs:label` value ends with `_binding` and adds the axiom `subClassOf molecular_function` to them:

```
1 ?y:CLASS=Match("((\w+))_binding")
2 SELECT ?y subClassOf Thing
3 BEGIN
4 ADD ?y subClassOf molecular_function
5 END;
```

[13] http://miuras.inf.um.es/~mfoppl/

4.1.2 Creating variables (`create`, `create Union|Intersection`)

Variables can be created anew. Such variables represent entities that will be created as the result of processing other variables that represent entities that do exist in the ontology. Two types of processes can be used to generate variables: `create` or any of the functions `createUnion`, `createIntersection`.

The `create` function creates entities from strings, and three processes can be used to do so: `?var.RENDERING`, `?var.GROUPS` and `?var.IRI`. A string can be added to any of the three processes using the `+` symbol, as in Java. The following script transforms, for example, the axiom `car subClassOf hasPart some engine` into `car subClassOf hasFeature some (Part and hasValue some engine)` [2]:

```
1 ?x:CLASS,
2 ?y:OBJECTPROPERTY = MATCH("has((\w+))"),
3 ?z:CLASS,
4 ?feature:CLASS = create(?y.GROUPS(1))
5 SELECT ASSERTED ?x subClassOf ?y some ?z
6 BEGIN
7 REMOVE ?x subClassOf ?y some ?z,
8 ADD ?x subClassOf !hasFeature some (?feature and !hasValue some ?z)
9 END;
```

The `?y` object property will be any object property whose `rdfs:label` value matches (`MATCH`) the regular expression `has((\w+))`. The `GROUPS` keyword is used to refer to the groups of the matched string; the `?feature` class will be a newly created class (`create`) whose name is the first group of the matched string (`GROUPS(1)`). In the actions section two new entities will be created using the `!` keyword: `hasFeature` and `hasValue`.

The `createUnion` and `createIntersection` functions can be used to create new entities from the values that have been bound to a prior variable. Both functions must be used with the `?var.VALUES` parameter, that is, they take the values (entities) of the variable as parameters. For example the following script creates a copy of the class `Margherita` (`MargheritaCopy`) by making a copy of the union of the toppings of `Margherita` (The script can be used with the Pizza Ontology[14]):

```
1 ?topping:CLASS,
2 ?allToppings:CLASS = createUnion(?topping.VALUES)
3 SELECT
4 Margherita subClassOf NamedPizza,
5 Margherita subClassOf hasTopping some ?topping
6 BEGIN
7 ADD !MargheritaCopy subClassOf NamedPizza,
8 ADD !MargheritaCopy subClassOf hasTopping some ?topping,
9 ADD !MargheritaCopy subClassOf hasTopping only ?allToppings
10 END;
```

4.1.3 Variable scope (`[Variable Free OWL Expression]`)

Individual variables can be constrained with arbitrary expressions. For example, in the following script we make sure that the variable `?x`, apart from being an OWL class, it is a subClassOf `Pizza` (This script can be tried against the Pizza ontology):

[14] http://www.co-ode.org/ontologies/pizza/

```
1 ?x:CLASS[subClassOf Pizza]
2 SELECT ?x subClassOf hasTopping some PepperTopping
3 BEGIN
4 REMOVE ?x subClassOf hasTopping some PepperTopping
5 END;
```

4.2 Working in asserted mode (ASSERTED)

OPPL can be used without an automated reasoner, if the inference is not needed. For example, we might be interested in querying for direct subclasses, using the annotation values for querying, *etc.* This can be useful since it improves the performance of the plug-in. In order to deactivate the inference the `ASSERTED` keyword should be used in the selection section. The following script detects the following pattern in an ontology: a class is asserted to be a subclass of the parent class, and also equivalent to the parent class. Since such structure is redundant, the script removes the `subClassOf` axiom (The script can be used against the Sequence Ontology, SO[15]). By using the asserted mode the execution of the script is much faster:

```
1 ?child:CLASS, ?parent:CLASS, ?filler:CLASS, ?prop:OBJECTPROPERTY
2 SELECT
3 ASSERTED ?child equivalentTo ?parent and (?prop some ?filler),
4 ASSERTED ?child SubClassOf ?parent
5 BEGIN
6 REMOVE ?child SubClassOf ?parent
7 END;
```

4.3 Using constraints (WHERE)

The selected entities can be further constrained using the `WHERE` keyword followed by a constraint method. There are four constraint method: `!=`, `MATCH`, `FAIL`, and `IN`.

4.3.1 Different entities (!=)

This constraint makes sure that the selected entities have different URIs. For example, we can make sure that the subclasses of a given class we want to select are all different. The following example script, which can be used against CL, selects all the direct (`ASSERTED`) subclasses of CL_0000255, then gets all the retrieved entities that are different to each other (`?child != ?other_child`) and makes them pairwise disjoint.

```
1 ?child:CLASS, ?other_child:CLASS
2 SELECT
3 ASSERTED ?child SubClassOf CL_0000255,
4 ASSERTED ?other_child SubClassOf CL_0000255
5 WHERE ?child != ?other_child
6 BEGIN
7 ADD ?child DisjointWith ?other_child
8 END;
```

[15] http://www.berkeleybop.org/ontologies/owl/SO

4.3.2 String matching (MATCH)

The MATCH keyword works as in the SELECT section.

4.3.3 Negation as failure (FAIL)

OWL works with the Open World Assumption (OWA). Thus, for an OWL compliant automated reasoner the fact that an assertion has not been made does not mean that the assertion is false. In other words, OWL does not work with Negation As Failure: for example, if we assert in an ontology that a citizen has British nationality, OWL does not assume that the same citizen has not also Spanish nationality: the citizen may have double nationality. For OWL, it remains unknown till we explicitly assert that the citizen can only have one nationality.

The following script for the Pizza ontology makes use of the FAIL keyword to find all the classes that are not unsatisfiable and removes the `subClassOf hasTopping some PepperTopping` axiom from them:

```
1 x:CLASS[subClassOf Pizza]
2 SELECT
3 ?x subClassOf hasTopping some PepperTopping
4 WHERE FAIL ?x subClassOf Nothing
5 BEGIN
6 REMOVE ?x subClassOf hasTopping some PepperTopping
7 END;
```

4.3.4 Variable values (IN)

The IN keyword constrains the values of a variable to a set of possible ones. For example the following script (Executable with the Pizza ontology) selects the instances of the class Country that are the country of origin of satisfiable classes, and also that are one of England, Italy, France, or Germany and makes them instances of the new class EuropeanCountry:

```
1 ?x: INDIVIDUAL, ?y:CLASS
2 SELECT
3 ?x instanceOf Country,
4 ?y subClassOf hasCountryOfOrigin value ?x
5 WHERE
6 ?x IN {England, Italy, France, Germany},
7 FAIL ?y subClassOf Nothing
8 BEGIN
9 ADD ?x instanceOf !EuropeanCountry
10 END;
```

4.4 OPPL patterns

An OWL pattern is an abstract OWL structure. For example, the closure pattern (An intersection of an existential restriction and an universal restriction: `?p some ?x and ?p only ?y`) can be parameterised with many different entities. The OPPL patterns plug-in[16] (Figure 9) can be used to generate such patterns and to execute them, *i.e.* to fill such abstract structures with actual entities from the target ontology (To bind entities to ?p, ?x and ?y). The OPPL patterns plug-in can be found in `Window >> Views >> Ontology views >> Patterns`.

[16] http://oppl2.sourceforge.net/patterns_documentation.html

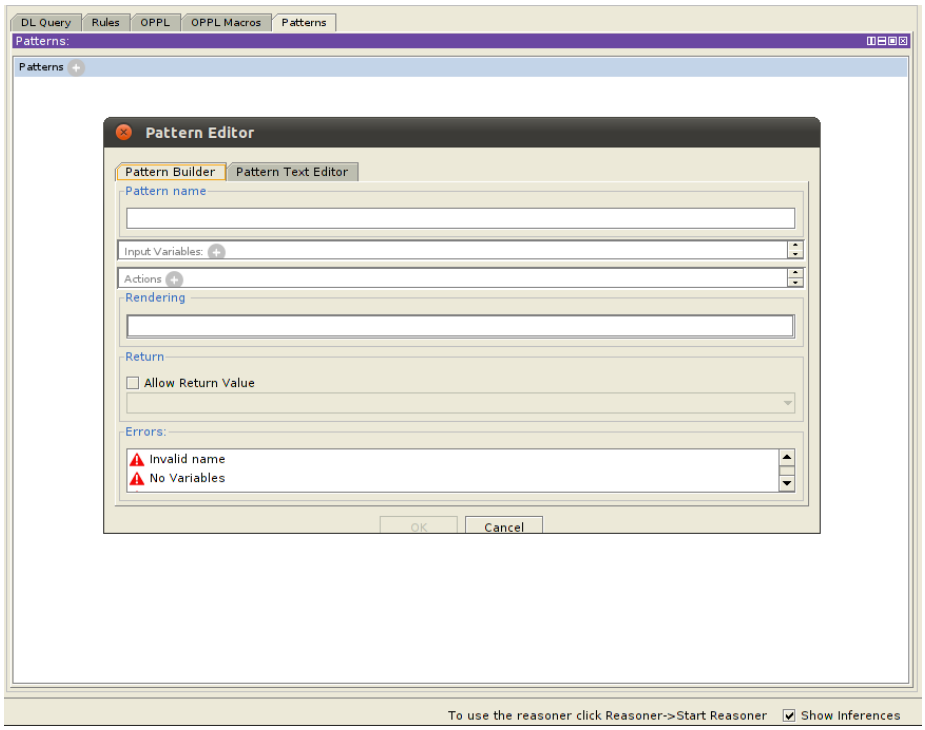

Figure 9: OPPL patterns tab.

4.5 The OPPL API

The OPPL API can be used to write programs that exploit OPPL's functionality. For example the following excerpt shows the source code of a program that detects certain patterns on ontologies defined by the OPPL scripts passed to it by the user (`OPPL_script_source` is an OPPL script that has been extracted from a plain text file):

```
1 ParserFactory parserFactory = new ParserFactory(manager,
      OWL_ontology, reasoner);
2 Logger logger = Logger.getLogger(Application.class.getName());
3 ErrorListener errorListener = (ErrorListener)new
      LoggerErrorListener(logger);
4 OPPLParser opplparser = parserFactory.build(errorListener);
5 OPPLScript OPPLscript = opplparser.parse(OPPL_script_source);
6 RuntimeExceptionHandler exceptionhandler = new
      QuickFailRuntimeExceptionHandler();
7 ChangeExtractor extractor = new ChangeExtractor(exceptionhandler,
      true);
8 extractor.visit(OPPLscript);
9 ConstraintSystem cs = OPPLscript.getConstraintSystem();
10 Set<BindingNode> nodes =  cs.getLeaves();
11 System.out.println("[OPPL: result entities amount] " + nodes.size()
      );
12 Iterator NodesIterator = nodes.iterator();
13 while(NodesIterator.hasNext()){
14   System.out.println("[OPPL: result entity] " + NodesIterator.next
       ());
15 }
```

4.6 Populous

Populous is a tool for populating ontologies with the data stored in spreadsheets[17]. OPPL can be used as part of populous (Populous includes an OPPL wizard) to add rich axioms as part of of the populating process.

[17] http://www.e-lico.eu/?q=populous

References

[1] M Horridge, N Drummond, J Goodwin, A Rector, R Stevens, and H Wang. The Manchester OWL syntax. In *OWLed*, 2006.

[2] Ondřej Šváb Zamazal, Vojtěch Svátek, and Luigi Iannone. Pattern-based ontology transformation service exploiting oppl and owl-api. In *Proceedings of the 17th international conference on Knowledge engineering and management by the masses*, EKAW'10, pages 105–119, Berlin, Heidelberg, 2010. Springer-Verlag.

A OPPL grammar

The latest version of the grammar can be checked in the OPPL web page[18].

A.1 Statements

```
1 OPPL Statement ::= ( <VariableDeclaration> )? ( <Query> )? ( <
      Actions> )? ";"
2 VariableDeclaration ::= <VariableDefinition> ( "," <
      VariableDefinition> )*
3 Actions ::= "BEGIN" Action ( "," Action )+ "END"
4 VariableDefinition ::= <InputVariableDefinition> | <
      GeneratedVariableDefinition>
5 InputVariableDefinition ::= <VARIABLE_NAME> ":" <variableType> (<
      VariableTypeScope>)? | VARIABLE_NAME <variableType> " = " "
      MATCH" <RegularExpression>
6 GeneratedVariableDefinition ::= <VARIABLE_NAME> ":" <variableType>
      "=" <opplFunction>
7 opplFunction ::= <create> | <creatInteserctions> | <
      createDisjunction>| Any Manchester Syntax with variables
      expression compatible with the generated variable.
8 create ::="create("<value>")"
9 createIntersection ::="createIntersection("<variablevalues>")"
10 createDisjunction ::="createDisjunction("< variablevalues>")"
11 variablealues ::=<VARIABLE_NAME>".VALUES"
12 value ::= a string constant | <generatedValue>
13 generatedValue ::=<variableAttribute> (<aggregator> <
      variableAttribute>)*
14 aggregator ::="+"
15 variableAttribute ::=<VARIABLE_NAME>"."<attributeName>
16 attributeName ::="RENDERING" | "GROUP(" <DIGIT>+")" | "IRI"
17 VariableTypeScope ::= "[" <direction> <VariableFreeOWLExpression>"]
      "
18 direction ::= "subClassOf" | "superClassOf" | subPropertyOf | "
      superPropertyOf" | "instanceOf"
19 /*Direction production is not context free as it depends on which
      variable type the variable
20 is being applied to. The scope, therefore, is not context free
      either*/
21 Constraint ::= <VARIABLE_NAME> "!=" <OWLExpression>
22 | <VARIABLE_NAME> "MATCH" <RegularExpression>
23 | <VARIABLE_NAME> "IN" "{" <OWLExpression> ("," <OWLExpression>)* "
      }"
24 "FAIL" <axiom>
25 variableType ::= "CLASS" | "OBJECTPROPERTY | "DATAPROPERTY" | "
      INDIVIDUAL" | "CONSTANT"
26 Query ::= "SELECT" ("ASSERTED")? <Axiom> ( ", ("ASSERTED")?" <Axiom
      > )* ( "WHERE" <Constraint> ("," <Constraint> )* )?
27 Action ::= "ADD" | "REMOVE" <Axiom>
28 IDENTIFIER ::= <LETTER> (<LETTER>|<DIGIT>)*
29 VARIABLE_NAME ::= "?"<IDENTIFIER>
30 LETTER ::= ["_","a"-"z","A"-"Z", ,"\u00e0"-"\u00f9"]
31 DIGIT ::= ["0"-"9"]
32 OWLExpression ::= An OWL entity in Manchester OWL Syntax (possibly
      containing variables)
33 VariableFreeOWLExpression ::= An OWL entity in Manchester OWL
      Syntax (without variables)
34 RegularExpression ::= "("" A regular expression for string matching
      (applies to the entity rendering) "")"
```

[18] http://oppl2.sourceforge.net/grammar.html

A.2 Manchester OWL Syntax axioms

```
1  Axiom <SubClassAxiom
2  | <EquivalentClassAxiom>
3  | <DisjointClassAxiom>
4  | <FunctionalObjectPropertyAxiom>
5  | <SymmetricObjectPropertyAxiom>
6  | <ReflexiveObjectPropertyAxiom>
7  | <TransitiveObjectPropertyAxiom>
8  | <AntiSymmetricObjectPropertyAxiom>
9  | <IrreflexiveObjectPropertyAxiom>
10 | <SubObjectPropertyAxiom>
11 | <EquivalentObjectPropertyAxiom>
12 | <DisjointPropertyAxiom>
13 | <InversePropertyAxiom>
14 | <InverseFunctionalPropertyAxiom>
15 | <FunctionalDataPropertyAxiom>
16 | <ObjectPropertyRangeAxiom>
17 | <ObjectPropertyDomainAxiom>
18 | <SubDataPropertyAxiom>
19 | <EquivalentDataPropertyAxiom>
20 | <DisjointPropertyAxiom>
21 | <DataPropertyDomainAxiom>
22 | <DataPropertyRangeAxiom>
23 | <ClassAssertionAxiom>
24 | <ObjectPropertyAssertionAxiom>
25 | <DataPropertyAssertionAxiom>
26 | <NegativeObjectPropertyAssertionAxiom>
27 | <NegativeDataPropertyAssertionAxiom>
28 | <SameAsAxiom>
29 | <DifferentFromAxiom> | <EntityAnnotatonAxiom>
30 SubClassAxiom ::= <ClassDescription> "SubClassOf" <ClassDescription
       >
31 EquivalentClassAxiom ::= <ClassDescription> "EquivalentTo" (<
       ClassDescription>)+
32 DisjointClassAxiom ::= <ClassDescription> "DisjointWith" (<
       ClassDescription>)+
33 FunctionalObjectPropertyAxiom ::= "Functional" <ObjectProperty>
34 SymmetricObjectPropertyAxiom ::= "Symmetric" <ObjectProperty>
35 ReflexiveObjectPropertyAxiom ::= "Reflexive" <ObjectProperty>
36 TransitiveObjectPropertyAxiom ::= "Transitive" <ObjectProperty>
37 AntiSymmetricObjectPropertyAxiom ::= "AntiSymmetric" <
       ObjectProperty>
38 IrreflexiveObjectPropertyAxiom ::= "Irreflexive" <ObjectProperty>
39 SubObjectPropertyAxiom ::= <ObjectProperty> "SubPropertyOf" <
       ObjectProperty>
40 EquivalentObjectPropertyAxiom ::= <ObjectProperty> "EquivalentTo"
       (<ObjectProperty>)+
41 DisjointPropertyAxiom ::= <ObjectProperty> "DisjointWith" (<
       ObjectProperty>)+
42 InversePropertyAxiom ::= <ObjectProperty> "InverseOf" "("<
       ObjectProperty>")"
43 InverseFunctionalPropertyAxiom ::= <ObjectProperty> "
       InverseFunctional" "("<ObjectProperty>")"
44 FunctionalDataPropertyAxiom ::= "Functional" <DataProperty>
45 ObjectPropertyRangeAxiom ::= <ObjectProperty> "Range" <
       ClassDescription>
46 ObjectPropertyDomainAxiom ::= <ObjectProperty> "Domain" <
       ClassDescription>
47 SubDataPropertyAxiom ::= <DataProperty> "SubPropertyOf" <
       DataProperty>
```

```
48 EquivalentDataPropertyAxiom ::= <DataProperty> "EquivalentTo" (<
       DataProperty>)+
49 DisjointPropertyAxiom ::= <DataProperty> "DisjointWith" (<
       DataProperty>)+
50 DataPropertyDomainAxiom ::= <DataProperty> "Domain" <
       ClassDescription>
51 DataPropertyRangeAxiom ::= <DataProperty> "Range" <DataRange>
52 ClassAssertionAxiom ::=  <Individual> "InstanceOf" | "Types" <
       ClassDescription>
53 ObjectPropertyAssertionAxiom ::= <Individual> <ObjectProperty> <
       Individual>
54 DataPropertyAssertionAxiom ::= <Individual> <DataProperty> <
       Constant>
55 NegativeObjectPropertyAssertionAxiom ::= "not" <Individual> <
       ObjectProperty> <Individual>
56 NegativeDataPropertyAssertionAxiom ::= "not" <Individual> <
       DataProperty> <Constant>
57 SameAsAxiom ::= <Individual> "sameAs" (<Individual>)+
58 DifferentFromAxiom ::= <Individual> "differentFrom" (<Individual>)+
59 EntityAnnotationAxiom ::= <IRI> <AnnotationProperty> <Constant>
```

A.3 Manchester OWL Syntax with variables entities

```
 1 ClassDescription ::= <ClassIntersection>
 2 ClassIntersection ::= <ClassUnion> ("and" <ClassUnion>)*
 3 ClassUnion ::= <NonN-aryDescription> ("or " <NonN-aryDescription>)*
 4 NonN-aryDescription ::= <PrimitiveClass> | <ObjectRestriction> | <
       DataRestriction> | "not" <ClassDescription> | "oneOf {" <
       Individual> (, <Individual>)* "}"
 5 DataRestriction ::= <DataProperty> "some" <DataRange> | <
       DataProperty> "only" <DataRange> | <DataProperty> "value" <
       Constant> | <DataProperty> "min" <NonNegativeInteger> (<
       DataRange>)? | <DataProperty> "exactly" <NonNegativeInteger> (<
       DataRange>)? | <DataProperty> "max" <NonNegativeInteger> (<
       DataRange>)?
 6 ObjectRestriction ::= <ObjectProperty> "some" <ClassDescription> |
      <ObjectProperty> "only" <ClassDescription> | <ObjectProperty> "
       value" <Individual> | <ObjectProperty> "min" <
       NonNegativeInteger> (<ClassDescription>)? | <ObjectProperty> "
       exactly" <NonNegativeInteger> (<ClassDescription>)? | <
       ObjectProperty> "max" <NonNegativeInteger> (<ClassDescription>)
       ?
 7 PrimitiveClass ::=<ClassName> | <VariableName>
 8 ObjectProperty ::=<ObjectPropertyName> | <VariableName>
 9 DataProperty ::=<DataPropertyName> | <VariableName>
10 AnnotationProperty ::=<AnnotationPropertyName> | <VariableName>
11 Individual ::=<IndividualName> | <VariableName>
12 Constant ::=<ConstantLiteral> | <VariableName>
13 ClassName ::= <LETTER> (<LETTER>|<DIGIT>)*
14 ObjectPropertyName ::= <LETTER> (<LETTER>|<DIGIT>)*
15 DataPropertyName ::= <LETTER> (<LETTER>|<DIGIT>)*
16 IndividualName ::= <LETTER> (<LETTER>|<DIGIT>)*
17 ConstantLiteral ::= "See the OWL specification"
18 DataRange ::= See Manchester OWL Syntax references above
19 NonNegativeInteger ::= Any integer greater than or equal to zero
20 ClassName ::= <LETTER> (<LETTER>|<DIGIT>)*
21 IRI ::= ["<"a valid IRI as defined in IETF RFC 3987">" | <
       VariableName>".IRI"
22 VariableName ::= "?" <LETTER> (<LETTER>|<DIGIT>)*
23 LETTER ::= ["_","a"-"z","A"-"Z","\u00e0"-"\u00f9"]
24 DIGIT ::= ["0"-"9"]
```