Embedding Knowledge Patterns into OWL

Luigi Iannone, Alan Rector, and Robert Stevens

School of Computer Science,
University of Manchester,
Manchester,
M13 9PL UK
{iannone,rector,robert.stevens}@cs.manchester.ac.uk

Abstract. We describe the design and use of the Ontology Pre-Processor Language (OPPL) as a means of embedding the use of Knowledge Patterns in OWL ontologies. We illustrate the specification of patterns in OPPL and discuss the advantages of its adoption by Ontology Engineers with respect to ontology generation, transformation, and maintainability. The consequence of the declarative specification of patterns will be their unambiguous description inside an ontology in OWL. Thus, OPPL enables an ontology engineer to work at the level of the pattern, rather than of the raw OWL axioms. Moreover, patterns can be analysed rigorously, so that the repercussions of their reuse can be better understood by ontology engineers and tools implementers. Thus the delivery of patterns with OPPL can provide a means of addressing the opacity and sustainability of OWL ontologies.

1 Introduction

The current focus of discussion about ontology engineering is on topics such as: the *sustainability* of the knowledge maintenance process; the *opacity* of the design of the underlying reusable knowledge models; and the lack of tool support in these and other areas. To cite one recent work:

> Today, one of the most challenging and neglected areas of ontology design is reusability. The possible reasons include at least: size and complexity of the major reusable ontologies, opacity of design rationales in most ontologies, lack of criteria in the way existing knowledge resources (e.g. thesauri, database schemata, lexica) can be reengineered, and brittleness of tools that should assist ontology designers. [1]

The evidence that knowledge models expressed in OWL can be, and indeed are, opaque to their users comes (indirectly) from the interest sparked by work on entailment justifications. Some models are so hard to interpret that researchers have recently focused on automatically generating explanations that are better suited for users' interpretation and understanding [2]. Whereas justifications seek to make the inferences from the raw OWL more comprehensible, patterns seek to raise the level of abstraction at which the ontology is formulated. This is

L. Aroyo et al. (Eds.): ESWC 2009, LNCS 5554, pp. 218–232, 2009.
© Springer-Verlag Berlin Heidelberg 2009

achieved by encapsulating a particular *pattern* of axioms that captures a particular modelling issue and enables the desired inferences to be made. Understanding may still be required, but the solution is ready-made and its reuse provides consistency of style.

Knowledge Patterns, introduced in [3], have been succinctly described, in [4], as:

> [...] namely representations which capture recurring structure within and across ontologies

This notion, coupled with the concept from Software Engineering of Design Patterns, led to the introduction of Ontology Design Patterns (ODPs) [5] that are collections of best practices in modeling knowledge. ODP have been further categorised into *Logic* and *Content* patterns [1]. Current work in ODPs aims to collect reusable fragments of patterns of knowledge representation that can be accessed, documented, validated, and, finally, deployed by knowledge engineers. Portals[1] have recently been launched to host centralised collections of ODPs to facilitate their uptake. Although this could be beneficial for the creation of patterns validated by the community, we argue that the aspects more strictly related to the pragmatics of the reuse are still largely unaddressed. We observe that, apart from the documentation, there is no way, to the best of our knowledge, to specify that an ontology uses a pattern or the way a pattern should be used. This means that if ontology engineers use a pattern in an ontology they might have to document the details of its application or, in case of reuse, the ontology's maintainers will have to examine the knowledge model to understand the pattern instantiation. We argue that what is needed is a language and a framework that allows the explicit definition of patterns and the ability to refer to them in OWL ontologies. Such a language would have the following advantages:

- Declarative specification of patterns provides the possibility to describe unambiguously what it means, in OWL, to use a pattern inside an ontology and therefore
  - to produce pattern instantiations automatically;
  - to add pattern support to ontology engineering tools.
- Patterns can be analysed formally, and the repercussions of their reuse can be better understood by ontology engineers.
- Specifying those patterns that are to be reused in an ontology represents a viable means (besides natural language documentation) of expressing the ways such ontology should evolve according to the intentions of its original modelers.

Note that the word *pattern* is more often used as a synonym for Ontology Design Pattern than in its general accepted meaning of *knowledge pattern*. From what we said above, an ODP is a *knowledge pattern*, but the converse does not necessarily hold in general. The main contribution of this work is to propose a

[1] E.g.: `http://ontologydesignpatterns.org`

language to specify knowledge patterns in OWL, therefore, in the following, the occurrence of the term *pattern* will not refer to an ODP, unless it is explicitly indicated.

The remainder of this paper is organised as follows:

- we specify our proposal for such a language;
- we propose a framework for embedding it into OWL ontologies;
- we present some example of patterns coming from various ontologies and try to give an idea of the extent of the impact of pattern introduction;
- we discuss briefly some motivating applications;
- finally, we discuss some promising research directions that the introduction of this or a similar language could enable.

2 OPPL for Patterns

In [6] we introduced the new version of the Ontology Pre-Processing Language initially proposed in [7] and applied in [8] as a declarative manipulation language for ontologies. In [8] OPPL was initially motivated by the need of ontology developers to transform one ontology to an axiomatically richer form. Here the aim is to design a language to encapsulate recurring knowledge structures (set of parametrized operations on OWL axioms) expressed in OWL. The OWL axiom is the basic unit upon which OPPL acts. Each OPPL construct and its effects can be simply expressed in terms of *additions* and *removals* of axioms. This means that:

a. The semantics underneath patterns in OPPL are OWL-DL semantics, which is both familiar to both OWL users and well-understood by implementers and language designers;
b. The effects of using a pattern can be immediately computed/visualized without needing any intermediate translation.

The grammar for the whole of OPPL can be found at `http://www.cs.man.ac.uk/~iannonel/oppl/documentation.html`. Here, we limit ourselves to illustrating, by means of examples, the most relevant features of the language when used for pattern specification. (Examples can be found in Figures 1 and 2).

2.1 Language Overview

The pattern sub-language of OPPL has two main components, namely:

1. Variables;
2. Actions.

Variables, just as in full OPPL, can have the following types:

1. CLASS;
2. OBJECTPROPERTY;

3. DATAPROPERTY;
4. INDIVIDUAL;
5. CONSTANT.

they cover all the logical entity types in an OWL ontology[2].

Actions can either ADD or REMOVE an axiom that, in its turn can be any axiom in OWL-DL built upon a combination of variables and entities from an ontology. We will introduce the OPPL syntax with an example pattern using the widely known Wine Ontology from W3C OWL Guide[3]. Suppose a knowledge engineer wants to provide a pattern to build new sub-classes of `wine:Wine`, so that the ontology could be enriched with wine classes that do not appear in the initial model. He could specify the OPPL Pattern shown in Figure 1[4].

Variables
```
?region:CLASS,
?winery:CLASS[subClassOf Winery],
?grape:CLASS,
?body:INDIVIDUAL,
?color:INDIVIDUAL,
?flavor:INDIVIDUAL,
?sugar:INDIVIDUAL
```
Actions
```
BEGIN
ADD $thisClass subClassOf Wine,
ADD $thisClass subClassOf  locatedIn some ?region,
ADD $thisClass subClassOf hasMaker some ?winery,
ADD $thisClass subClassOf madeFromGrape some ?grape,
ADD $thisClass subClassOf hasBody value ?body,
ADD $thisClass subClassOf hasColor value ?color,
ADD $thisClass subClassOf hasFlavor value ?flavor,
ADD $thisClass subClassOf hasSugar value ?sugar
END;
```
Rendering
```
Wine with located in ?region made by ?winery from ?grape grape
with ?body body, ?color, color , ?flavor flavor, and ?sugar sugar
```

Fig. 1. Hypothetical pattern for the creation of a kind of Wine in the W3C Wine ontology

[2] With entity here we denote: named classes, object and data properties, individuals, and constants. Note that annotations are not considered as they are currently not part of the logical structure of OWL and their specification is changing rapidly in the revision from OWL 1.0 to OWL 2.0.

[3] `http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine`

[4] Notice that OPPL relies on Manchester OWL Syntax [9], from which its own syntax has been derived by adding variable support. Hence, all the axioms in the example ontologies in this paper will also be reported using their Manchester OWL Syntax rendering, with the exception of Figure 5 that uses RDF/XML-ABBREV as axiom annotations are not yet supported in Manchester Syntax.

The pattern above defines the new wine in terms of 7 variables that correspond to the features specified, in the starting ontology, by the class `wine:Wine` whose description we report below for completeness:

```
Wine subClassOf PotableLiquid
Wine subClassOf locatedIn some Region
Wine subClassOf hasMaker only Winery
Wine subClassOf madeFromGrape min 1 Thing
Wine subClassOf hasBody exactly 1 Thing
Wine subClassOf hasColor exactly 1 Thing
Wine subClassOf hasFlavor exactly 1 Thing
Wine subClassOf hasMaker exactly 1 Thing
Wine subClassOf hasSugar exactly 1 Thing
```

After the variable definition block, the OPPL pattern specifies the operations on the ontology that its use (henceforth *instantiation*) requires to be executed. In our case they consist of adding 8 axioms. This pattern is peculiar because it is supposed to be applied to a class description. It means that the ontology engineer, in order to use this pattern, will have to pick a class already present in the ontology to which the pattern will be applied. The way to refer to such a class inside a pattern is to use the reserved variable name `$thisClass`. An equivalent pattern that is not supposed to be applied to a class can be obtained by adding another variable to the one in Figure 1, and editing the pattern as reported in Figure 2.

The block after the actions is called *rendering*. Its purpose is merely to provide tools with more user-friendly presentations for patterns. Its content is, in fact, free text in which variable names could be inserted. In a pattern instantiation, such variable names will be replaced with the values assigned to them.

Now suppose our knowledge engineer decides to instantiate the pattern in Figure 2 assigning to its variable the following values:

```
?wine → PrimitivoManduriaDolceNaturale
?region → PugliaRegion
?winery → PuglieseWinery
?grape → Primitivo
?body → Full
?color → Red
?flavor → Strong
?sugar → Sweet
```

The effects of such an instantiation will be the addition of the following axioms:

```
ADD PrimitivoManduriaDolceNaturale subClassOf Wine
ADD PrimitivoManduriaDolceNaturale subClassOf locatedIn some PugliaRegion
ADD PrimitivoManduriaDolceNaturale subClassOf hasMaker some PuglieseWinery
ADD PrimitivoManduriaDolceNaturale subClassOf madeFromGrape some Primitivo
ADD PrimitivoManduriaDolceNaturale subClassOf hasBody value Full
```

```
?wine:CLASS,
?region:CLASS,
?winery:CLASS[subClassOf Winery],
?grape:CLASS,
?body:INDIVIDUAL,
?color:INDIVIDUAL,
?flavor:INDIVIDUAL,
?sugar:INDIVIDUAL
BEGIN
ADD ?wine subClassOf Wine,
ADD ?wine subClassOf  locatedIn some ?region,
ADD ?wine subClassOf hasMaker some ?winery,
ADD ?wine subClassOf madeFromGrape some ?grape,
ADD ?wine subClassOf hasBody value ?body,
ADD ?wine subClassOf hasColor value ?color,
ADD ?wine subClassOf hasFlavor value ?flavor,
ADD ?wine subClassOf hasSugar value ?sugar
END;
?wine Wine with located in ?region made by ?winery from ?grape grape
with ?body body, ?color color , ?flavor flavor, and ?sugar sugar
```

Fig. 2. Pattern for the creation of a kind of Wine in the W3C Wine ontology not applicable to a class

```
ADD PrimitivoManduriaDolceNaturale subClassOf hasColor value Red
ADD PrimitivoManduriaDolceNaturale subClassOf value Strong
ADD PrimitivoManduriaDolceNaturale subClassOf value Sweet
```

And its rendering will be:
`PrimitivoManduriaDolceNaturale` Wine located in `PugliaRegion` made by `PuglieseWinery` from `Primitivo` grape with `Full` body, `Red` color, `Strong` flavor, and `Sweet` sugar.

An OPPL pattern may reference other OPPL patterns. To illustrate this consider another example ontology, this time about food. For the sake of simplicity let us reduce it to the following axioms:

```
Food subClassOf Thing
Meat subClassOf Food
Egg subClassOf Food
Vegetables subClassOf Food
Objectproperty contains
Course subClassOf contains some Food
Menu subClassOf contains some Course
```

Let us suppose that we want to specify sub classes of `Course` and `Menu` on the basis of what they do or do not contain. Let us consider the following pattern (called `freeFromCourse` and shown in Figure 3).

This pattern uses the `RETURN` clause. This clause defines a particular subset of patterns that, when instantiated, not only executes their actions, but

```
?x:CLASS,
?forbiddenContent:CLASS=createUnion(?x.VALUES)
BEGIN
ADD $thisClass equivalentTo Food and contains only (not ?forbiddenContent)
END;
A ?x free stuff;
RETURN $thisClass
```

Fig. 3. Our `FreeFromPattern`

also *returns* a value. The instantiation of the pattern above, when applied to a given class, makes it equivalent to the class of all the individuals whose fillers, for the property `contains`, are all in the complement of the class variable `?forbiddenContent`. This variable is 'generated'–this means that its values will not be provided by the users in the instantiation, but will come from other variables/entities in the pattern. In our case, this variable is the disjunction of all the values assumed by the `CLASS` variable `?x`. Now suppose we instantiate this pattern, assigning `Meat` and `Egg` to `?x`, and apply it to `VegetarianCourse`. The ontology, after the instantiation will include the following axioms:

```
VegetarianCourse equivalentTo contains only (not (Egg or Meat))
```

Although kept simple for the sake of clarity, this example shows the potential of patterns and the advantages of having a language for their expression. The only thing the knowledge engineer had to specify was a single pattern. After that, enriching the ontology with all the plausible combinations of food content one may need does not require an understanding of all of the underlying model. All that needs to be done is the assignment of the desired values to the (in this case) unique input variable.

The return clause, however, makes the above pattern suitable to be referenced. This means that other patterns can refer to it in their actions. For instance, let us say we want to specify a pattern for creating `Menu` sub-classes that do not contain a certain food. We can build that pattern upon the pattern already created, using the syntax in Figure 4.

```
?x:CLASS[subClassOf Food]
BEGIN
ADD $thisClass subClassOf Menu,
ADD $thisClass subClassOf contains Course and only ($FreeFromPattern(?x))
END;
A ?x - free Menu
```

Fig. 4. Pattern on Menu reusing the pattern in Figure 3, FreeFromPattern

Let us suppose we want to create a `VegetarianMenu` class and instantiate it using again `Egg` and `Meat` as values for our input variable `?x`.

The following axioms would be added to the ontology on instantiation:

```
VegetarianMenu subClassOf Menu
VegetarianMenu subClassOf contains Course and only
  (contains only not (Egg or Meat))
```

In other words, an inner pattern instantiation retrieves all the class equivalence axioms that have the return variable as member and in place of the reference in the outer pattern[5] it puts the conjunction of all the equivalent classes that appear in such axioms[6].

2.2 Embedding Patterns into OWL Ontologies

This section briefly describes our framework that allows the embedding patterns, specified using OPPL, and their instantiations into ontologies. We will illustrate it using screenshots from a tool we developed as a plug-in for Protégé 4 (`http://www.cs.man.ac.uk/~iannonel/oppl/patterns`). This tool allows the creation, editing and instantiation of such patterns. This therefore facilitates the maintenance of an ontology when using Protégé 4. Trivial as it may appear in the examples above, we provided a way to encapsulate the semantics of being *something which does not contain* `?x`, in the pattern described in Figure 3. Should the way of modeling that change in the ontology, knowledge engineers would need to maintain only the pattern and the framework should update the referencing entities accordingly.

There are three main pieces of information that need to be stored into an ontology in order to enable the use of OPPL patterns, they are:

1. Pattern definitions, i.e.: the OPPL specifications, rendering, and, optionally, return clauses described in the previous section;
2. Pattern instantiations, i.e: the set of values assigned to each variable when using a pattern;
3. Pattern generated effects, i.e.: the reference to the creating pattern for each axiom generated by a pattern instantiation.

All three pieces of information are stored in different annotations. Pattern definitions and pattern instantiations are serialized into ontology annotations, unless the pattern to be instantiated uses the reserved `$thisClass` variable (pattern applicable to classes). If `$thisClass` is used, the pattern instantiation refers to a particular class in the ontology—therefore, it is stored as an annotation for that class.

Finally, every axiom added by a pattern instantiation is annotated with enough information to reference what instantiation caused the assertion of such an axiom. An example of the resulting OWL serialization of the pattern in Figure 3, its instantiation and the corresponding annotated axiom is reported in Figure 5.

[5] Currently, in our language only class variables can be returned.

[6] In our case, there is only one axiom and one equivalent class in the pattern in Figure 3.

```
[...]
    <owl:Ontology rdf:about="">
        <patterns:Free rdf:datatype="&xsd;string"
            >?x:CLASS,?forbiddenContent:CLASS=createUnion(?x.VALUES)
            BEGIN
             ADD $thisClass equivalentTo contains only (not ?forbiddenContent )
 END;
 A ?x free stuff ;
 RETURN $thisClass
  </patterns:Free>
    </owl:Ontology>

[...]
    <!-- http://www.coode.org/patterns/examples/food#VegetarianCourse -->

    <owl:Class rdf:about="#VegetarianCourse">
        <rdfs:subClassOf rdf:resource="#Course"/>
        <patterns:VegetarianCourseFreePatternInstantiation rdf:datatype="&xsd;string">
         $Free(Meat)
        </patterns:VegetarianCourseFreePatternInstantiation>
    </owl:Class>
    <rdf:Description>
        <rdf:type rdf:resource="&owl;Axiom"/>
        <rdf:subject rdf:resource="#VegetarianCourse"/>
        <rdf:predicate rdf:resource="&owl;equivalentClass"/>
        <rdf:object>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#contains"/>
                <owl:allValuesFrom>
                    <owl:Class>
                        <owl:complementOf rdf:resource="#Meat"/>
                    </owl:Class>
                </owl:allValuesFrom>
            </owl:Restriction>
        </rdf:object>
        <patterns:createdBy rdf:datatype="&xsd;string">
        http://www.co-ode.org/patterns#VegetarianCourseFreePatternInstantiation
        </patterns:createdBy>
    </rdf:Description>
```

Fig. 5. Pattern specification, instantiation and storage effects in OWL excerpts

Methodologies have recently been proposed to create and reuse knowledge patterns. In particular, in [1] (see Section 3), a *common set of operations* for creating and using Ontology Content Patterns (OCPs) has been described. An ontology content pattern, is, informally, a knowledge pattern that, although domain specific, solves recurrent common modeling problems. In the same paper, the authors illustrate the creation and the usage of patterns by means of two examples extracted from the Dolce Ultra Light ontology (DUL[7]): namely the *Information Realization* and the *Time Indexed Person Role* content patterns. The proposed extraction process produces entries for an online catalogue to be browsed by knowledge engineers. The typical entry of such a catalogue looks like a form with fields specifying:

[7] http://www.loa-cnr.it/ontologies/DUL.owl

- Name;
- Intent;
- Extracted from;
- Examples;
- Diagram;
- Elements.

There seems, however, not to be a concrete representation of the captured pattern so that it could be reused *off the shelf*. We claim that besides the information already provided by catalogues of patterns, encoding the patterns in OPPL could further enhance their usability. Currently the instantiation technique suggested by the authors relies on cloning (partially, or completely) entities from the pattern origin ontology and constructing the resulting axioms using SPARQL [8]. The authors acknowledge that the problem of cloning has not been fully addressed so far and acknowledge it may happen that the ontology engineers have to manually edit SPARQL results. Using OPPL allows the expression of patterns in the same ontological language and therefore avoids the need for custom SPARQL adjustments as the patterns creators can specify which axioms will correspond to a pattern instantiation.

Let us consider, now, the first OCP, *Information Realization*. In a nutshell, the DUL ontology says that:

> [Information Realization] represents the relations between information objects like poems, songs, formulas, etc., and their physical realizations like printed books, registered tracks, physical les, etc.

In Figure 6 we report the OPPL encoding of this pattern. Notice that this pattern when instantiated in an ontology that imports the DUL will assert the opportune sub-class relationships between the variables and the classes/properties in the DUL ontology (i.e: `InformationRealization`, `InformationObject`, and `realizationProperty`) and then the required axiom that relates the information object to its realization. The advantage of this is that it can be directly stored in the ontology, and applied, just assigning values to the three variables. Once validated by the ontology engineer, this pattern can be routinely instantiated to produce as many `InformationRealization` sub-classes as required—all its users have to know is what values they should assign to the variables.

Likewise, we can consider the other example in the paper referenced above: *Time Indexed Person Role* described as:

> a CP that represents time indexing for the relation between persons and roles they play

Looking at the ontology, a *Time Indexed Person Role* is a sub-class of `Situation` whose settings include a sub-class of `Person`, of `Role`, and of `TimeInterval`. A generic pattern for *Time Indexed Person Role* could be the one shown in Figure 7. Notice that there is no particular reason to interpret the

[8] `http://www.w3.org/TR/rdf-sparql-query/`

```
?informationObject:CLASS,
?informationRealization:CLASS,
?realizationProperty:OBJECTPROPERTY
BEGIN
ADD ?informationRealization subClassOf InformationRealization,
ADD ?informationObject subClassOf InformationObject,
ADD ?realizationProperty subPropertyOf realizes,
ADD ?informationRealization  subClassOf PhysicalObject
     and ?realizationProperty some ?InformationObject
END;
Information Realization Pattern:
?informationRealization ?realizationProperty ?informationObject
```

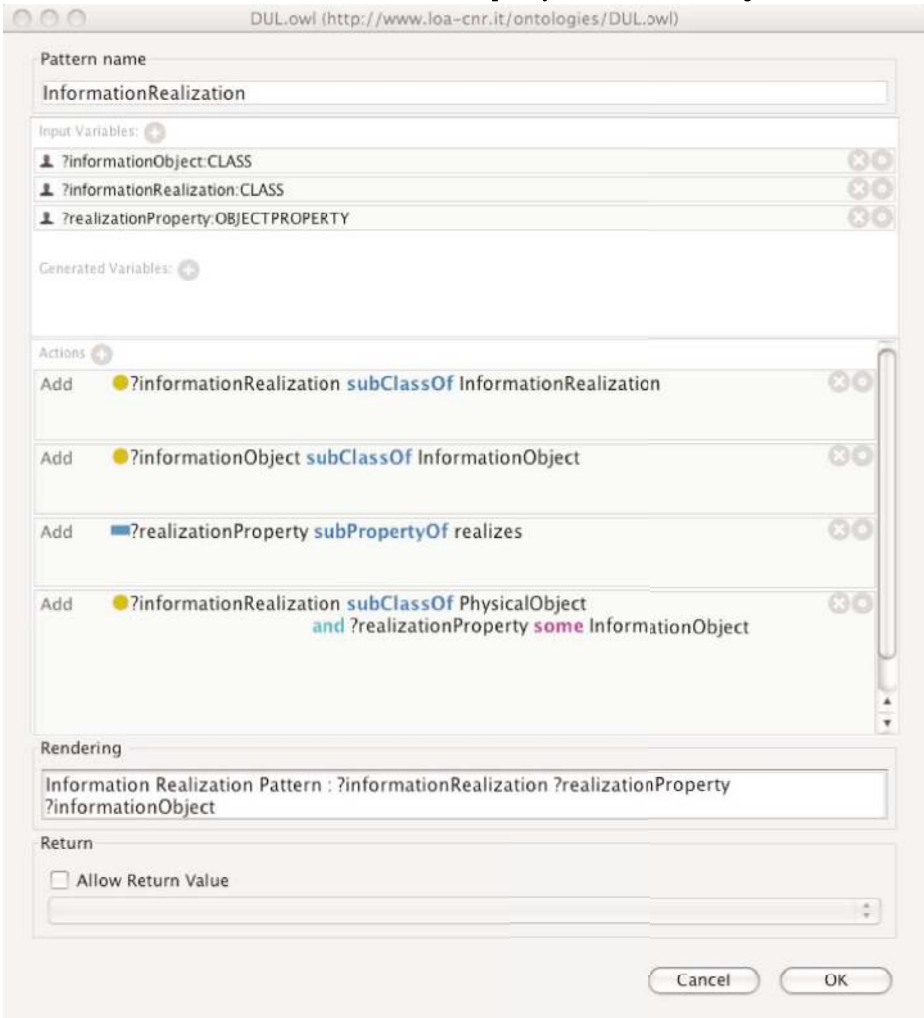

Fig. 6. Information Object Pattern in OPPL

latter pattern as a class, the purpose was only to show an alternative to the plain, class independent implementation of the previous *Information Realization* example. There is, however, another important difference between the two exemplar patterns: their level of generality. In the former both the classes and the property involved have been replaced by variables. In the latter a fixed reference to the property `isSettingFor` appears. The introduction of a `OBJECTPROPERTY` variable that replaces such property would produce a more general pattern. Conversely, the elimination of a variable and its replacement with an entity would result in a more specific version of the *Information Realization Pattern* described above. These two operations correspond to implementing in OPPL what, in [1], is meant by pattern *generalization* and *specialization*.

3 Related Work

The literature about knowledge patterns, even when restricted to the work specifically aimed at Semantic Web formalisms, is vast. Therefore, the survey in this Section is not intended to be exhaustive. Its aim is rather to identify where our contribution stands with respect to the state-of-the-art. Clark et *al.* were the first, to the best of our knowledge, to use the label *knowledge patterns*, defining them as:

> first order theories whose axioms are not part of the target knowledge-base, but can be incorporate via renaming of their non logical symbols (see Section 2 in [3]).

Explicit functions called *morphisms* were required to use patterns in knowledge bases. In the following years, researchers seemed to focus on pattern categorisation (see, for instance, [10] and its references) aiming at creating organised catalogues that engineers could browse and adopt in their knowledge bases. Recently, patterns have also been employed in semi-automatic knowledge acquisition tasks. In [11], they are combined with Case-Based Reasoning in order to guide the extraction of terminological knowledge from text documents. However, as far as we are aware, in the last decade, we can cite only two works that investigate the problem we discussed in this paper: pattern representation languages. Staab *et al.*, in [12], propose a framework based on RDF for creating catalogues of patterns that are (partially - according to the authors themselves) *executable/portable* in any implementation language. Vrandecic, in [13], proposes the creation of scripts (called *macros* in the paper) that, without changing the semantics of the underlying knowledge representation language – OWL-DL in this case – capture sets of axioms and can be reused. Our approach is less ambitious than Staab *et al.*'s as our pattern specification language is targeted to cover OWL-DL ontologies only. Therefore, in Staab *et al.*'s framework, ours could be the component responsible for translating a semantic pattern into OWL-DL.

With respect to Vrandecic's work, by contrast, our framework seems a step forward, providing a concrete language for encoding patterns, although we differ in the interpretation of in what way a macro (pattern) should be dealt. Vrandecic hypothesizes that patterns should be encoded into the ontology, then interpreted

```
?person:CLASS,
?role:CLASS,
?timeInterval:CLASS
BEGIN
ADD $thisClass subClassOf Situation,
ADD $thisClass subClassOf isSettingFor some ?person,
ADD $thisClass subClassOf isSettingFor some ?role,
ADD $thisClass subClassOf isSettingFor some ?timeInterval,
END;
Situation where ?person play the role ?role during the time interval ?timeInterval
```

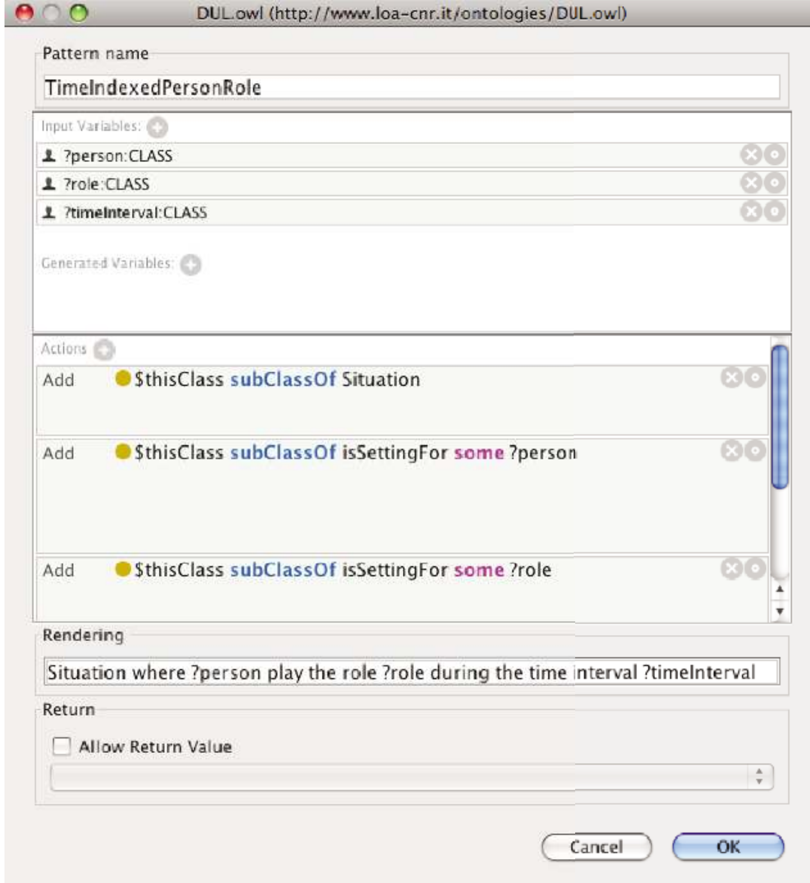

Fig. 7. Time Indexed Person Role Pattern

and resolved using external tools (maybe rule or template based) every time an ontology is opened to be sent to any reasoning service. On the contrary, we believe that patterns should be instantiated and references resolved as soon as they are used in the ontology.

As we explained above, when a pattern is instantiated, our framework will carry out the resulting actions on the ontology, and there will be no need of interpreters after that point.

4 Conclusions and Future Work

In this paper we used OPPL as a means of encoding patterns. OPPL patterns are a set of operations to be executed on the ontology to which they are applied. Such operations are specified in terms of the axioms to be added/removed with variables that have to be instantiated, assigning values to each. Having a language and a concrete framework to encode patterns into ontologies has the added value of enabling an ontology engineer to specify both what could be extended in an ontology and how this could be accomplished. Patterns, besides representing abstractions over recurrent modeling problems, could also function as ontology extension points. Knowledge engineers could systematically create patterns that indicate the way they foresee further axiomatizations in the ontology. this would implement a sort if *interface* - in its software engineering accepted meaning - for ontology expansion and at the same time hiding modeling details.

Furthermore, a declarative language for encoding patterns makes it possible to explore its expressivity limitations as well as the effects that certain kinds of patterns could have when instantiated within an ontology. Promising research results on the evaluation of the impact of ontology modifications [14,15] could be reused and applied to assess the consequences of the instantiation of a pattern. Such analysis could be automated and embedded into knowledge engineering tools that will provide more information, at a higher level of abstraction, to their users interested in maintaining an ontology exposing patterns.

Introducing a pattern language opens another interesting direction: pattern matching and induction. Matching is meant here as deciding whether either a given ontology or its part is compliant (*matches*) with a given pattern. Pattern induction, in contrast, is intended as detecting the regularities in an ontology, seeking recurring patterns. The application of Machine Learning techniques to study the complexity of the induction for this language is certainly worth investigating.

Last, but not least, its adoption as a language for patterns will contribute in steering the further developments of the OPPL language itself. OPPL was born as a scripting language to automate bulk modifications to ontologies. The language creators main concern was to maintain its decidability keeping a very parsimonious attitude with respect to the set of possible language constructs. It is likely that patterns will push in the opposite direction, demanding more and more expressivity. Reaching a useful trade-off will be an interesting research topic in itself. We therefore see this new version of OPPL, with its ability to declare knowledge patterns and instantiate them within an ontology, as the starting point for supporting the sustainable development of richly axiomatised ontologies.

References

1. Presutti, V., Gangemi, A.: Content ontology design patterns as practical building blocks for web ontologies. In: Li, Q., Spaccapietra, S., Yu, E., Olivé, A. (eds.) ER 2008. LNCS, vol. 5231, pp. 128–141. Springer, Heidelberg (2008)

2. Horridge, M., Parsia, B., Sattler, U.: Laconic and precise justifications in owl. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T.W., Thirunarayan, K. (eds.) International Semantic Web Conference. LNCS, vol. 5318, pp. 323–338. Springer, Heidelberg (2008)
3. Clark, P., Thompson, J., Porter, B.W.: Knowledge patterns. In: KR, pp. 591–600 (2000)
4. Clark, P.: Knowledge patterns. In: [16], pp. 1–3
5. Gangemi, A.: Ontology design patterns for semantic web content. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 262–276. Springer, Heidelberg (2005)
6. Iannone, L., Egaña, M., Rector, A., Stevens, R.: Augmenting the Expressivity of the Ontology Pre-Processor Language (2008), http://www.webont.org/owled/2008/papers/owled2008eu_submission_16.pdf
7. Egaña, M., Antezana, E., Stevens, R.: Transforming the Axiomisation of Ontologies: The Ontology Pre-Processor Language. In: Proceedigns of OWLED 2008 DC OWL: Experiences and Directions, Washington, DC (2008)
8. Egaña, M., Rector, A.L., Stevens, R., Antezana, E.: Applying ontology design patterns in bio-ontologies. In: [16], pp. 7–16
9. Horridge, M., Drummond, N., Godwin, J., Rector, A., Stevens, R., Wang, H.: The Manchester OWL Syntax. In: Proceedigns of OWLED 2006 OWL: Experiences and Directions, Athens GA, USA (2006)
10. Blomqvist, E., Sandkuhl, K.: Patterns in ontology engineering: Classification of ontology patterns. In: Chen, C.S., Filipe, J., Seruca, I., Cordeiro, J. (eds.) ICEIS (3), pp. 413–416 (2005)
11. Blomqvist, E.: Ontocase - a pattern-based ontology construction approach. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 971–988. Springer, Heidelberg (2007)
12. Staab, S., Erdmann, M., Maedche, A.: Engineering ontologies using semantic patterns. In: IJCAI Workshop on E-business & The Intelligent Web (2001)
13. Vrandecic, D.: Explicit knowledge engineering patterns with macros. In: Proceedings of the Ontology Patterns for the Semantic Web Workshop at the ISWC 2005, Galway, Ireland (November 2005)
14. Grau, B.C., Horrocks, I., Kutz, O., Sattler, U.: Will my ontologies fit together? In: Parsia, B., Sattler, U., Toman, D. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 189, CEUR-WS.org (2006)
15. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Ontology reuse: Better safe than sorry. In: Calvanese, D., Franconi, E., Haarslev, V., Lembo, D., Motik, B., Turhan, A.Y., Tessaris, S. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 250, CEUR-WS.org (2007)
16. Gangemi, A., Euzenat, J. (eds.): EKAW 2008. LNCS, vol. 5268. Springer, Heidelberg (2008)