Assessing the Safety of Knowledge Patterns in OWL Ontologies

Luigi Iannone, Ignazio Palmisano, Alan L. Rector, and Robert Stevens

University of Manchester
Kilburn Building
Oxford Road M13 9PL
Manchester, UK
lastname@cs.manchester.ac.uk

Abstract. The availability of a concrete language for embedding knowledge patterns inside OWL ontologies makes it possible to analyze their impact on the semantics when applied to the ontologies themselves. Starting from recent results available in the literature, this work proposes a sufficient condition for identifying *safe* patterns encoded in OPPL. The resulting framework can be used to implement OWL ontology engineering tools that help knowledge engineers to understand the level of extensibility of their models as well as pattern users to determine what are the safe ways of utilizing a pattern in their ontologies.

1 Introduction

OWL ontologies may be built for the most disparate purposes. Yet, as soon as they are made public, the problem of their extension arises. The presence of the `owl:import` primitive in the language testifies to the inclination of OWL to encourage ontology re-use. The addition of any axiom to an ontology does, however, modify its semantics. The extent of the impact of these alterations can be evaluated, but , at the moment, not anticipated or formally analyzed when an ontology is either being developed or recently released. The knowledge engineers, at the current state of the art, have only annotations for documenting what are the possible extensions of their ontology, and there is no formal language for their encoding.

The effect of this limitation is better explained using an analogy. Let us suppose for a moment that the Java programming language did not provide the `final` primitive to specify that a certain Java class/method cannot be sub-classed/overridden. API developers could not prevent their users from extending portions of the API object model that are meant to be fixed and non extensible. The consequence would be that all the assumptions about some pieces of code behaving in a fixed and pre-determined way would no longer hold, thus making the re-use of third party code more unpredictable and unreliable. One could argue that knowledge re-use is meant to be more flexible than the code counterpart, that is why we observe that nothing as strict as the `final` primitive is needed. What we claim, though, is that it would be useful if the knowledge engineer, whilst developing their ontologies, could specify what and how to extend it in order to remain compliant with its original meta-model. The users could then decide whether to follow such guidelines or not, but natural language annotations are too

L. Aroyo et al. (Eds.): ESWC 2010, Part I, LNCS 6088, pp. 137–151, 2010.
© Springer-Verlag Berlin Heidelberg 2010

ambiguous for their specification. Secondly, but equally importantly, a formal language could also help the original ontology creators in understanding which ones, amongst many envisaged extensions are actually harmless with respect to the original semantics of the ontology itself.

We recently [1] proposed the adoption of OPPL 2, a declarative manipulation language for ontologies, as a language for embedding knowledge patterns into OWL ontologies. In this paper, we now propose a formal framework for evaluating the impact that the usage of such patterns (written in OPPL) could have on the semantics of an ontology. We aim to provide a methodology for deciding which patterns are actually safe to use in combination with an ontology i.e.: they do not disrupt its original semantics.

The remainder of this paper is structured as follows: Sect. 2 recaps the main features of our concrete language for embedding patterns into an OWL Ontology; Sect. 3 illustrates our proposal for the evaluation of a pattern's impact on an ontology; Sect. 4 briefly surveys the state of the art on the subject; Sect. 5 Discusses the implications of our approach and outlines possible future directions for the investigation in this area.

2 OPPL Patterns

One of the main criticisms of the current way of re-using ontologies is that OWL artefacts are often *opaque* because of their size and their complexity [2]. The rationales, especially those behind the major ontologies, designed to be re-used, are often difficult to grasp. Consider, for instance, the following axiom taken from the Dolce Ultra-Light ontology[1] (in Manchester OWL Syntax):

```
InformationRealization equivalentTo
              PhysicalObject and realizes some InformationObject
              or (Event and hasParticipant some PhysicalObject)
```

It is an implementation of the *Information Realization* pattern that "[...] represents the relations between information objects like poems, songs, formulas, etc., and their physical realizations like printed books, registered tracks, physical files, etc. [2]". There is no distinction between entities or constructs that are fixed parts of the pattern and those which are variable and can be modified in order to re-use the same pattern in different situations. As an example, if we restrict ourselves to the first disjunct:

```
InformationRealization equivalentTo
              PhysicalObject and realizes some InformationObject
```

the pattern describes a relationship between three entities: a sub-class of *InformationRealization*, a sub-property of *realizes*, and a sub-class of *InformationObject*. The fixed aspects of the pattern is the conjunction between being a *PhysicalObject* with, as a filler for a sub-property of *realizes*, a sub-class of *InformationObject*.

Although this appears in the natural language description of the pattern, an automatic tool cannot adequately derive this kind of information when dealing with the OWL exemplar usage of this pattern. This is why in [1] the adoption of OPPL as a concrete language for embedding knowledge patterns into OWL ontologies was proposed. In [3], OPPL was initially motivated by the need of ontology developers to transform one

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

ontology to an axiomatically richer form. The aim of its pattern sub-language is, instead, to encapsulate recurring knowledge structures (set of parameterized operations on OWL axioms) expressed in OWL. For a example-based description of the OPPL pattern sub-language we refer the reader to [1]; the complete OPPL grammar is available online[2].

Here we limit ourselves to report the basic structure of an OPPL pattern: *Variables* and *Actions*. Variables, just as in full OPPL, have one of the following types: CLASS, OBJECTPROPERTY, DATAPROPERTY, INDIVIDUAL, CONSTANT. They can be *input* variables, whose values should be provided by the user, or *generated* variables, whose value depends on other variables.

All variables, except for those of the CONSTANT kind, can be scoped; this means that the set of allowed values for a variable can be restricted:

(a) `superClassOf` and `subClassOf` for CLASS variables; the values must be super-classes (sub-classes) of a class expression: `?x:CLASS[subClassOf` A`]`.
(b) `superPropertyOf` and `subPropertyOf` for DATAPROPERTY and OBJECTPROPERTY variables; the values must be super-properties (sub-properties) of a property expression.
(c) `instanceOf` for INDIVIDUAL variables; the values must be instances of a class expression: `?x:INDIVIDUAL[instanceOf` A `and hasP some` B`]`.

Let us consider the *partWhole* pattern, which describes a whole as having some parts and constrains all the possible parts to be among those in the values of the `?part` variable; this pattern will be referenced in the remainder of the paper.

```
?whole:CLASS, ?part:CLASS, ?allParts:CLASS = createUnion(?part.VALUES)
BEGIN
    ADD ?whole subClassOf has_direct_part some ?part,
    ADD ?whole subClassOf has_direct_part only ?allParts
END;
```

It is the simplest OPPL version of the recurring knowledge pattern that binds an object to all its parts and imposes that such an object cannot have other parts than those specified in the values of the `?part` variable. The pattern presents two input variables (`?whole` and `?part`) and a generated one (`?allParts`) that is the union of all the values assigned to `?part`. It can be instantiated to populate one's ontology, with the only assumption that such an ontology contains the `has_direct_part` object property. One can, for example, invoke the pattern as follows:

```
partWhole(Molecule,Atom)
partWhole(Atom, {Proton, Neutron, Electron})
```

The results of the instantiation are[3]:

```
Molecule subClassOf has_direct_part some Atom
Molecule subClassOf has_direct_part only Atom
Atom subClassOf has_direct_part some Proton
Atom subClassOf has_direct_part some Neutron
Atom subClassOf has_direct_part some Electron
Atom subClassOf has_direct_part only (Proton or Electron or Neutron)
```

To the best of our knowledge, there is no alternative concrete language to express knowledge patterns and use them inside tools. An attempt to build a framework for

[2] `http://www.cs.man.ac.uk/~iannonel/oppl/documentation.html`

[3] This is only an illustrative example, not a correct ontology for Physics (i.e., Hydrogen atoms need not have neutrons).

automatically extracting patterns from an ontology and apply them to another one is described in [2]. It does not, however, attempt to store the extracted pattern in any form, hence it cannot be re-used. Moreover, it relies on SPARQL [4] for implementing the extraction mechanism, which, in the authors' own words, allows only for partial extractions that often require manual refinements (see Sect. 3.1 in [2]).

Besides the ability to store and re-use knowledge patterns using the same vocabulary of the embedding ontology, having a concrete language (such as OPPL) opens other directions of investigation concerning the repercussions of the use of a particular pattern in an ontology. In particular, as we shall see in the next section, one can determine whether the instantiations of such patterns in an ontology have consequences on its semantics. This makes it possible both for the pattern authors and their users to anticipate the circumstances under which the usage of a pattern in combination with an ontology will cause changes in the model, and prevent undesirable alterations.

3 Evaluating the Impact of Using a Pattern

A concrete language for encoding patterns inside ontologies provides a means for ontology engineers to specify how their ontologies could be extended; e.g., the *partWhole* pattern, when instantiated, will produce sets of axioms; this set can then be either added to the ontology itself or to an extension. Sometimes, however, adding an axiom to an ontology can affect its semantics in ways that are not in line with the principle of re-using and extending an ontology. Let us illustrate this by means of an example, by using *partWhole* in combination with two ontologies $\mathcal{O}_1$ and $\mathcal{O}_2$:

$\mathcal{O}_1$	$\mathcal{O}_2$
	`owl:import` $\mathcal{O}_1$
$Molecule \sqsubseteq \top$, $Atom \sqsubseteq \top$	$Proton \sqsubseteq \top$, $Electron \sqsubseteq \top$, $Neutron \sqsubseteq \top$
$has_part^* \sqsubseteq has_part$	$Atom \sqcap Proton \sqsubseteq \bot$, $Atom \sqcap Neutron \sqsubseteq \bot$, $Atom \sqcap Electron \sqsubseteq \bot$
	$Proton \sqcap Electron \sqsubseteq \bot$, $Proton \sqcap Neutron \sqsubseteq \bot$, $Neutron \sqcap Electron \sqsubseteq \bot$

Now let us instantiate `partWhole`, and add the resulting axioms, listed in the previous section, to $\mathcal{O}_2$. One non obvious outcome of the whole process is that now a reasoner can derive that *Molecule* and *Atom* are disjoint[5]. Irrespective of whether this may or may not make sense in the particular domain, we have just proved that the instantiation of a pattern, which was aimed at extending the initial ontology, has some side-effects that act on the semantics of entities defined in our imported ontology ($\mathcal{O}_1$). This means that there are axioms, using only symbols from $\mathcal{O}_1$, which do not hold in $\mathcal{O}_1$ but hold in $\mathcal{O}_2$, after the above pattern instantiations; e.g., $\models Atom \sqcap Molecule \sqsubseteq \bot$. This means that $\mathcal{O}_1$ is being extended without preserving its original semantics.

The main shortcoming of *disruptively* extending ontologies in this way can be better illustrated by the following scenario. Let us suppose that we have a software application that uses $\mathcal{O}_1$ for its functionalities. Now imagine we have to add one functionality that requires that we extend $\mathcal{O}_1$. If our extension does not preserve the semantics of $\mathcal{O}_1$, there

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

[5] From: $Molecule \sqsubseteq \forall has_direct_part.Atom$, $Molecule \sqsubseteq \exists has_direct_part.Atom$, $Atom \sqcap Proton \sqsubseteq \bot$, $Atom \sqcap Neutron \sqsubseteq \bot$, $Atom \sqcap Electron \sqsubseteq \bot$ $Atom \sqsubseteq \forall has_direct_part.(Proton \sqcup Electron \sqcup Neutron)$

is no guarantee that all the functionalities in our pre-existing version of the application will keep working after replacing $\mathcal{O}_1$ with it. This means we will have to, at least, re-test, and, in the worst case, re-write, the portion of the application which is now incompatible with the changes produced by our extension. This is one of the reasons why, in the recent literature the notion of *conservative extensions* of an ontology [4] received a lot of attention. Intuitively, a conservative extension of an ontology is another ontology that has the same interpretation for the symbols it shares with the original one. A consequence of this definition is that queries on the shared vocabulary have the same results irrespective of whether they are carried out on the original ontology or on one of its conservative extensions. Producing conservative extensions when re-using ontologies is, therefore, very important to avoid the drawbacks described in the little scenario above. Since in [1] we proposed OPPL patterns as means to produce re-usable ontology, the question becomes whether it is possible to decide if a pattern will produce a conservative extension of the ontology it is applied to. We already know from [5] that determining whether an ontology *safely extends* (is a conservative extension of) another ontology is not decidable for expressive Description Logics such as OWL-DL. Deciding whether a pattern will produce a safe extension presents the further complication of taking variables into account.

To the best of our knowledge, the only workaround for the undecidability mentioned above is the identification of classes of axioms that produce a conservative extension when added to an ontology. Belonging to one of such classes is only a sufficient condition for safety, and, so far, only one class has been fully characterised and depends on the notion of the *locality* of an axiom (see Def. 25 in [5]). Intuitively, an axiom is local w.r.t. a signature when, once reduced to the symbols in the signature, it results in either a tautology or in an axiom that already holds in the ontology under consideration. The idea is that the axiom does not modify the set of valid interpretations for the symbols in the signature w.r.t. the one they have in the original ontology. This means that whenever an axiom which is local w.r.t. a signature is added to an ontology, the resulting ontology is a safe (conservative[6]) extension of the original one. In the remainder of this section we extend the notion of locality to patterns, in order to provide a solution for determining whether a pattern produces safe extensions for an ontology. Let us start by defining a pattern in an abstract way[7].

Definition 3.1 (Variable Names). *Called V_{Class} the set of class variable names, V_{Op} the set of object property variable names, V_{Dp} the set of data property variable names, V_{Ind} the set of individual variable names, and V_{Const} the set of constant variable names, we call $V = \{V_{Class} \cup V_{Op} \cup V_{Dp} \cup V_{Ind} \cup V_{Const}\}$ the set of disjunct sets of variable names.*

Definition 3.2 (Variable Expression). *We can define a variable expression inductively as follows:*

[6] There are different kinds of conservative extensions in literature. Analysing the difference between them is beyond the scope of this paper. Here we use the term conservative extension in its *deductive* meaning, i.e.: those preserving the results of every query when restricted to the symbols in the signature. We refer the reader to [4] and [5] for further particulars.

[7] Without using OPPL proprietary syntax, i.e.: Manchester OWL Syntax.

Called VE_{Class} *the set of* class *variable expressions,* VE_{Op} *the set of object properties variable expressions,* VE_{Dp} *the set of data properties variable expressions,* VE_{Ind} *the set of individual variable expressions,* VE_{Const} *the set of constants,* $VE_{Assertion}$ *the set of* assertion *variable expressions,* $VE = \{VE_{Class} \sqcup VE_{Assertion} \sqcup VE_{Ind} \sqcup VE_{Dp} \sqcup VE_{Op}\}$ *the set of variable expressions,*

- *if* $v \in V$*, then* $v \in VE$*;*
- *if* $v \in V_{Class}$*, then* $\neg v \in VE_{Class}$*;*
- *if* $v \in V_{Op}$ *and e is a (variable) class expression, then* $\exists v.e, \forall v.e \in VE_{Class}$*;*
- *if* $v \in V_{Dp}$ *and e is a data type expression, then* $\exists v.e, \forall v.e \in VE_{Class}$*;*
- *if* $v \in VE_{Class}$ *and* i *is an individual, then* $v(i) \in VE_{Assertion}$*;*
- *if* $v \in VE_{Ind}$ *and e is a (variable) class expression, then* $e(v) \in VE_{Assertion}$*;*
- *if* $v, f \in V_{Ind}$ *and* p *is a (variable) property expression (data or object) then* $p(v, f) \in VE_{Assertion}$*;*
- *if* $v \in V_{Ind}$*, then* $\{v\} \in VE_{Class}$*;*
- *if* $v \in VE_{Class}$ *and* C *is a (variable) class expression, then* $v \sqcap C, v \sqcup C \in VE_{Class}$*;*

Moreover, if $v \in V_{Class}$*,* $v_f \in VE$ *is a* ***generated*** *variable obtained applying a function* $f(v)$ *to* v*, where* f *returns variable expressions; e.g.,* $p \in V_{Op}, f : V_{Class} \times V_{Op} \rightarrow VE = \exists p.v$*; this will be represented as* $v_f = \exists p.v$*.*

Variable expressions can only be assigned to generated variables, while input variables are restricted to named entities and data type constants[8]. All the variable names in this paper will start with a question mark ($?x, ?y, \ldots$). In order to specify the kind of a variable, we will borrow the notation from OPPL— as it is more compact—and write: *?x:CLASS* in place of $?x \in V_{Class}$, or *?p:DATAPROPERTY* in place of $?p \in V_{Dp}$.

Furthermore, the function *VALUES*, used it in the *partWhole*, always appears in combination with other aggregating functions, such as, for example, $\bigsqcup$ and $\bigsqcap$. Hence the expression: *?x:CLASS, ?z:CLASS=* $\bigsqcap$*?x.VALUES* means that $?z \in VE_{Class}$ is a generated variable containing all the values of $?x$.

Definition 3.3 (Variable axiom)
A variable axiom α *over* V *is an OWL-DL axiom containing at least a variable expression* $ve \in VE$*.*

Therefore, if $v \in VE_{Class}$ and C a (variable or not) class expression according to Def. 3.2, then $v \sqsubseteq C$ and $C \sqsubseteq v$ are variable axioms.

We can now define a pattern as follows:

Definition 3.4 (Pattern). *Called* α_i *an OWL-DL axiom or a variable axiom over* V_p*,* $1 \leq i \leq n, n \geq 1$*, a pattern over* V *is:* $p = (V, \{\alpha_1, \ldots, \alpha_n\})$

Patterns without variable axioms will not be considered in this paper, since their locality can be decided straightforwardly by applying the results in [5] to each plain OWL-DL axiom in the pattern. We focus, instead, on patterns containing at least one variable axiom, in order to find a procedure to determine whether their instantiations produce safe extensions of the ontologies in which they are used in combination.

[8] This limitation keeps the OPPL query sub-language, not used here for patterns, decidable.

In order to define instantiations we need to introduce the notion of binding as follows:

Definition 3.5 (Binding). *A **binding** b over V is a set of assignments from variable names to variable expressions, where the type of variable expression is determined by the type of variable:*
$b = \{v \rightarrow \text{value}, (v, value) \in \{V_{Class} \times VE_{Class}\} \sqcup \{V_{Op} \times VE_{Op}\} \sqcup \{V_{Dp} \times VE_{Dp}\} \sqcup \{V_{Ind} \times VE_{Ind}\} \sqcup \{V_{Const} \times VE_{Const}\}\}$.

*A binding is **complete** over V if it contains an assignment for all the non generated variables in V. $b(v)$ will denote the value assigned to the variable v inside b.*

We can now define instantiations:

Definition 3.6 (Instantiation). *Let $p = (V, \{\alpha_1, \ldots, \alpha_n\})$ a pattern over V according to Def. 3.4, and B a set of complete bindings over V, with $vinV$ and $b \in B$. Then:*

- *An axiom instantiation $\sigma_b(\alpha_i)$ consists of replacing every occurrence of each variable v in α_i with $b(v)$, therefore obtaining an OWL DL axiom;*
- *A pattern instantiation over a single binding is defined as:*
  $\sigma_b(p) = \bigcup_{1 \leq i \leq n} \sigma_b(\alpha_i)$;
- *A pattern instantiation over the whole B corresponds to:*
  $\sigma_B(p) = \bigcup_{b \in B} \sigma_b(p)$.

Given a signature[9] **S** over a generic Description Logic $\mathcal{L}$, a pattern is said to be *local* w.r.t. **S** if all the axioms resulting from all its possible instantiations are local with respect to **S**. More formally:

Definition 3.7 (Local patterns). *Let $p = (V, \{\alpha_1, \ldots, \alpha_n\})$ a pattern over V according to Def. 3.4, and **S** a signature of a Description Logic $\mathcal{L}$.*
*Then p is **local** w.r.t. **S** iff:*
$\forall b$, where b is a complete binding over V,

$$\alpha \in \sigma_b(p) \Rightarrow \alpha \text{ is local with respect to } \mathbf{S}.$$

The problem now becomes to define an algorithm that decides whether a pattern can produce non local axioms in one of its instantiation. The one we propose in this paper uses the semantic locality test τ function detailed in [5] (see Proposition 30)[10], and is reported in Alg. 1. The idea behind it is to produce a set of bindings that are generic enough to encompass all the possible combinations of values assigned to the variables used by the pattern. Then, the algorithm proceeds to apply the classic locality test τ on each of these bindings. It stops when it either finds a binding which produces an instantiation containing an axiom that fails the locality test or when all the bindings have been examined. Such bindings are built considering only the non generated variables since

[9] A *signature* (or *vocabulary*) **S** of a DL is defined in [6] as:" the (disjoint) union of a set C of atomic concepts $(A, B, \ldots)$ representing sets of elements, a set R of atomic roles $(r, s, \ldots)$ representing binary relations between elements, and a set I of individuals $(a, b, c, \ldots)$ representing elements".

[10] Please notice that this reference pre-dates OWL 2 specification and is restricted to class axioms. It can be extended to encompass both property and individual axioms.

the locality of $\mathcal{L}$-constructs containing generated variables only depends on whether the **generating** variables are or not in the signature. Therefore, a generated variable can be seen as a function of one or more non generated one, and does not contribute directly to the locality of an axiom.

Proposition 3.1 (Completeness). *Let $p(V, \{\alpha_1, \ldots, \alpha_n\})$ be a pattern defined over a set of variables as in Def. 3.4. If the pattern is not local, our algorithm terminates returning `false` (i.e.: Alg. 1 is complete w.r.t. locality test failures).*

Proof. *Let us suppose there is some binding b_{out} and some i such that $\sigma_{b_{out}}(\alpha_i)$ fails the locality test τ. If we consider how τ has been recursively defined in Proposition 30 of [5] cited above, the locality depends, ultimately, on whether (some of) the symbols used in the axioms appear or not in the signature **S**. That is to say that, given an axiom α in a Description Logic $\mathcal{L}$ and a signature **S**, the only way to change the locality of α is, to remove (add) symbols used in α from (to) **S**. Now the first part of our algorithm (from line 1 to 10 in Alg. 1), creates a set of bindings B whose elements cover all the possible permutations of variable assignments to values within or outside the signature **S**. This means that, whatever the assignments contained in b_{out}, a binding $b \in B$ exists whose assignments are in the same situation[11] w.r.t. **S** as those in b_{out}. Therefore, the locality test τ will fail on b too when applied to α_i, hence, given the generality of b_{out} we can conclude that our algorithm is complete w.r.t. locality test failures.*

The soundness of our algorithm comes from the fact that it is built upon the locality test τ itself.

Proposition 3.2 (Soundness). *Let $p(V, \{\alpha_1, \ldots, \alpha_n\})$ be a pattern defined over a set of variables as in Def. 3.4. If our algorithm terminates returning `false`, the pattern is not local.*

Proof. *Let us suppose our algorithm terminates returning `false`. This can only happen if there is a b inside the set of bindings built in the first part of the algorithm (from line 1 to 10 in Alg. 1) which causes an instantiation of some axiom α_i to fail the locality test. Therefore, according to Def. 3.7, p is not local.*

3.1 Algorithm Trace

Now consider again the *partWhole* pattern. Suppose we want to test its locality before applying it to the ontology $\mathcal{O}_2$, and we choose as our signature the whole set of symbols appearing either in $\mathcal{O}_1$ or in $\mathcal{O}_2$ (variable types are as defined in *partWhole*):

$$\mathbf{S} = \{\mathit{Atom}, \mathit{Molecule}, \mathit{Electron}, \mathit{Proton}, \mathit{Neutron}, \mathit{has_part}, \mathit{has_direct_part}\}$$

$$\alpha_1 = \mathit{?whole} \sqsubseteq \exists \mathit{has_direct_part}.\mathit{?part}$$

$$\alpha_2 = \mathit{?whole} \sqsubseteq \forall \mathit{has_direct_part}.\mathit{?allParts}$$

[11] Please notice that we are **not** asserting they are the **same** assignments, as it is irrelevant for the locality test. What really matters is whether the values are or not in **S**.

Algorithm 1. Locality check for patterns

Require: A pattern $p(V, \{\alpha_1, \ldots, \alpha_n\})$.
A signature **S** over a Description Logic $\mathcal{L}$.
Ensure: The locality of p w.r.t. **S**.

1: $V_{input} \leftarrow \{v \in V, v\ is\ not\ generated\} = \{v_1, v_2 \ldots v_n\}, n = |V_{input}|$
2: **for** $1 \leq j \leq n$ **do**
3: **for** $b \in B$ **do**
4: $\mathbf{S} \leftarrow \mathbf{S} \cup \{v_j^{\mathbf{S}}\}$, $v_j^{\mathbf{S}}$ a symbol name
5: $v_j^{\neg\mathbf{S}}$ a symbol name, $v_j^{\neg\mathbf{S}} \notin \mathbf{S}$
6: $b' \leftarrow b \cup \{v_j \rightarrow v_j^{\mathbf{S}}\}, b'' \leftarrow b \cup \{v_j \rightarrow v_j^{\neg\mathbf{S}}\}$
7: $B \leftarrow (B \setminus b) \cup \{b', b''\}$
8: **end for**
9: **end for**
10: *found* $\leftarrow$ false
11: **for** $\neg$*found* $\wedge B \neq \emptyset$ **do**
12: Select $b \in B$
13: *found* $\leftarrow \neg\tau(\sigma_b(p), \mathbf{S})$
14: $B \leftarrow B \setminus \{b\}$
15: **end for**
16: **return** $\neg$*found*

Our *partWhole* pattern is:

$$partWhole(\{?part, ?whole, ?allParts\}, \{\alpha_1, \alpha_2\})$$

If we run our algorithm, the following results will be generated:

Updated signature	$\mathbf{S} \leftarrow \mathbf{S} \cup \{whole^{\mathbf{S}}, part^{\mathbf{S}}\}$
Set of complete bindings $B = \{b_1, b_2, b_3, b_4\}$	$b_1 = \{?part \rightarrow part^{\mathbf{S}}, ?whole \rightarrow whole^{\mathbf{S}}\}$ $b_2 = \{?part \rightarrow part^{\mathbf{S}}, ?whole \rightarrow whole^{\neg\mathbf{S}}\}$ $b_3 = \{?part \rightarrow part^{\neg\mathbf{S}}, ?whole \rightarrow whole^{\mathbf{S}}\}$ $b_4 = \{?part \rightarrow part^{\neg\mathbf{S}}, ?whole \rightarrow whole^{\neg\mathbf{S}}\}$
Instantiated axiom	$\sigma_{b_1}(\alpha_1) = whole^{\mathbf{S}} \sqsubseteq \exists has_direct_part.part^{\mathbf{S}}$

$\tau(\sigma_{b_1}(\alpha_1)) = \texttt{false}$ because it is neither a tautology nor did it hold previously in $\mathcal{O}_2$; therefore, *partWhole* is not local w.r.t. **S** when applied to $\mathcal{O}_2$.

3.2 Algorithm Complexity

The computational complexity of the proposed algorithm is non polynomial in n, where n is the number of non generated variables. The algorithm, indeed, builds as many bindings as necessary to represent the possible combinations of value assignment to variables with respect to their belonging or not to a signature; therefore, two possible values for each non generated variable, which brings 2^n combinations. The aim of this work is not to propose and efficient version of it, nevertheless we observe that:

– If one changes the strategy for creating the bindings, from the current breadth-first exploration of a tree whose nodes are variable assignments, to a depth-first one, the locality check can be moved and performed as soon as a new complete

binding b is created. This means that if it returns `false` for some b, there is no need to proceed in creating any other bindings. We preferred to write our algorithm without this optimisation for the sake of clarity, but its introduction would make the non polynomial exploration only the worst case.

– Patterns represent abstractions over recurring knowledge structures. Although an upper limit on the number of variables in a pattern does not exist, we observe that the introduction of a variable in a pattern often requires more than one axiom to go with it for representing the piece of recurring knowledge it is abstracting over. In other words, a variable axiom such as *?x:CLASS* $\sqsubseteq$ *?y:CLASS* has little meaning if there are no other axioms further characterising both *?x* and *?y*. However, such axioms, in their turn, decrease the level of generality of the pattern and its applicability to other ontologies. This means that there is a physiological balance between the number of variables in patterns, its meaningfulness, and its generality; this leads to think that real world patterns will have a limited number of input variables, and therefore acceptable performance for locality checks even in the worst case.

3.3 Determination of Safe Combinations

The example in Sec. 3.1, however, also raises the question of whether it is possible to invoke our *partWhole* pattern in a safe way. If we observe the instantiations our algorithm creates, we notice that, all the $\sigma_b(\alpha_i)$ for each b containing the assignment *?whole* $\rightarrow$ *whole*$^{\neg\mathbf{S}}$, as they become of the form $\bot \sqsubseteq \exists \ldots$ or $\bot \sqsubseteq \forall \ldots$. This means that our *partWhole* pattern is local, whenever it is instantiated assigning to *?whole* a value outside the signature **S**.

We could then think of modifying the second part of our algorithm (from line 11 onwards in Alg. 1), so that combinations of assignments that produce local instantiations are returned, if they exists. These modifications are reported in Alg. 2.

Algorithm 2. Extended Locality check for patterns

Require: A pattern $p(V, \{\alpha_1, \ldots, \alpha_n\})$.
A signature **S** over a Description Logic $\mathcal{L}$.
Ensure: The binding configurations that result in local instantiations for p.

1: identical to line 1–9 in Alg. 1
2: *LocalBindings* $\leftarrow \emptyset$
3: **for** $B \neq \emptyset$ **do**
4: Select $b \in B$
5: **if** $\tau(\sigma_b(p), \mathbf{S})$ **then**
6: *LocalBindings* $\leftarrow$ *LocalBindings* $\cup \{b\}$
7: **end if**
8: $B \leftarrow B \setminus \{b\}$
9: **end for**
10: **return** *LocalBindings*

This algorithm does not stop when it detects the first binding that produces a non local instantiation ($\tau(\sigma_b(p), \mathbf{S}) =$ `false`), but it examines all the bindings. This makes it impossible to use the depth first optimization mentioned above, nevertheless, finer grained ones could be introduced.

We observe that, for example, not every axiom, in a pattern, depends on all the variables. Therefore, we could determine the axioms that use the smallest number of variables, let us call them $\{\alpha_{min_1}, \dots \alpha_{min_k}\}$, compute the bindings restricted to those variables and check the locality instantiating such axioms. The only bindings we will expand to the other variables are those amongst the restricted ones that produced local instantiations on $\{\alpha_{min_1}, \dots \alpha_{min_k}\}$. In this way we can significantly prune the number of bindings to examine with respect to the non polynomial worst case.

3.4 Maximization of Safe Instantiations

Consider now a very small ontology $\mathcal{O}_{Food}$ and a pattern *foodWithout*:

$\mathcal{O}_{Food}$	$Ingredient \sqsubseteq \top$, $Meat \sqsubseteq Ingredient$, $Eggs \sqsubseteq Ingredient$, $Salad \sqsubseteq Ingredient$, $Food \sqsubseteq \top$, $Food \sqsubseteq \forall contains.Ingredient$
foodWithout	$(\{?x{:}CLASS, ?y{:}CLASS, ?forbidden{:}CLASS = \bigsqcup ?y.VALUES\}, \{\alpha\})$ $\alpha = ?x \sqsubseteq Food \sqcap \forall contains.\,(\neg ?forbidden)$
Possible bindings	$b_1 = \{?x{:}CLASS \rightarrow x^{\mathbf{S}}, ?y{:}CLASS \rightarrow y^{\mathbf{S}}\}$ $b_2 = \{?x{:}CLASS \rightarrow x^{\mathbf{S}}, ?y{:}CLASS \rightarrow y^{\neg\mathbf{S}}\}$ $b_3 = \{?x{:}CLASS \rightarrow x^{\neg\mathbf{S}}, ?y{:}CLASS \rightarrow y^{\mathbf{S}}\}$ $b_4 = \{?x{:}CLASS \rightarrow x^{\neg\mathbf{S}}, ?y{:}CLASS \rightarrow y^{\neg\mathbf{S}}\}$

If we ran Alg. 2 it would return b_3 and b_4 as the only binding configurations that produce local instantiations. In particular[12]:

$$\sigma_{b_3}(p)\ x^{\neg\mathbf{S}} \sqsubseteq Food \sqcap \forall contains.(y^{\mathbf{S}}) \quad \rightarrow \quad \bot \sqsubseteq Food \sqcap \forall contains.\,(\neg y^{\mathbf{S}}) \text{ (tautology)}$$
$$\sigma_{b_4}(p)\ x^{\neg\mathbf{S}} \sqsubseteq Food \sqcap \forall contains.(\neg y^{\mathbf{S}}) \quad \rightarrow \quad \bot \sqsubseteq Food \qquad \text{(tautology)}$$

This means that, given a signature **S**, if we instantiate our *foodWithout* pattern using, as values for *?x:CLASS*, entities that do not belong to **S** we will produce instantiations that will not modify the semantics of the entities in **S**. However, if we edited our pattern *foodWithout* by scoping *?x:CLASS* as *?x:CLASS*[`subClassOf` *Food*], we would constrain[13] *?x:CLASS* to have as values only sub-classes of *Food*. This implies $\sigma_{b_2}(p) \rightarrow x^{\mathbf{S}} \sqsubseteq Food$, which holds if we consider that of $x^{\mathbf{S}}$ is a value for *?x:CLASS*, constrained inside the scope declared above to be $\sqsubseteq$ *Food*.

Therefore, by changing the scope of one of the variables, one extra safe binding configuration would be determined, thus limiting the unsafe ones to be those configurations in which a user assigns to both *?x:CLASS* and *?y:CLASS* values that appear in **S**. Unfortunately, there is no simple way, for the time being, to modify any of the algorithms above in order to provide indications on how to scope variables so as to include previously non local binding configurations into the local ones. We limit ourselves to observe that, in order to solve this problem, it is necessary to compute, given an axiom

[12] In applying the τ locality test all the entities outside **S** are transformed into $\bot$

[13] Please see Sect. 2 for details on how to scope variables.

α and an ontology $\mathcal{O}$, with $\not\models_{\mathcal{O}} \alpha$, what are, if any, the possible combinations of axioms $AxiomSet_1, \ldots, AxiomSet_n$ such that, if $\mathcal{O}_i = \mathcal{O} \cup AxiomSet_i$ *then* $\forall i, \models_{\mathcal{O}_i} \alpha$. In particular, in our settings, our α would be the result of the instantiations that fail the locality test, and the sets of axioms we would be interested in are those that can translated into variable scope restrictions like the one on *?x:CLASS* in the *foodWithout* example above. Unfortunately, to the best of our knowledge this problem has not been solved yet in the literature, but it represents an interesting future direction of investigation.

The results discussed in this section, however, make it possible, give a pattern encoded in OPPL, to decide what values assigned to its variables will provide conservative extensions of the ontology it is used in combination with. Given a signature, indeed, our algorithms return the binding configurations leading to local instantiations. The purpose of such binding configurations is only to distinguish between variable values that belong or not to the input signature, which, as we showed above, is all is needed in order to determine the locality of the pattern instantiations. Thus, users know in advance, for example, that the *partWhole* pattern can be used safely w.r.t. **S**={*Molecule*, *Atom*} only when *?whole:CLASS* has values outside **S**, whichever those values are.

3.5 Implementation

Both algorithms described above have been implemented in the PATTERNS plugin[14] for Protégé 4. Some screenshots presenting the *partWhole* pattern and its safety analysis results are reported, in Fig. 1. The *parthWhole* pattern has two input variables, which can therefore be bound to any named class entity in the ontology; the signature **S** used in the depicted safety check is $\mathbf{S} = \{Molecule, Atom\}$.

When both variables are not bound, the possible situations are 2^2, i.e.,

- *?whole*, *?part* $\in$ **S**
- *?whole*, *?part* $\notin$ **S**
- only one of *?whole* or *?part* $\in$ **S**;

These four cases are depicted on the lower left part of Fig. 1; green circles mark the cases in which instantiation is safe, i.e., no changes occur to the semantics of the entities contained in **S**, while red circles mark the unsafe cases.

If one of the variables is bound, the number of cases decreases to 2^1; the lower right part of Fig. 1 represents the case in which *?part* is bound to *Atom*, which belongs to the signature. In this case, it is clear that the safety of the pattern is determined by *?whole*: whenever it is included in the signature, the resulting action could change the semantics of the symbols in **S**.

4 Related Work

Knowledge patterns have been gaining considerable attention as a means of promoting good ontology engineering practice. Our contribution to this aim builds upon previous work on the nature and cataloguing of such patterns. Clark et *al.* were the first, to the

[14] `http://www.cs.man.ac.uk/~iannonel/oppl/patterns/`

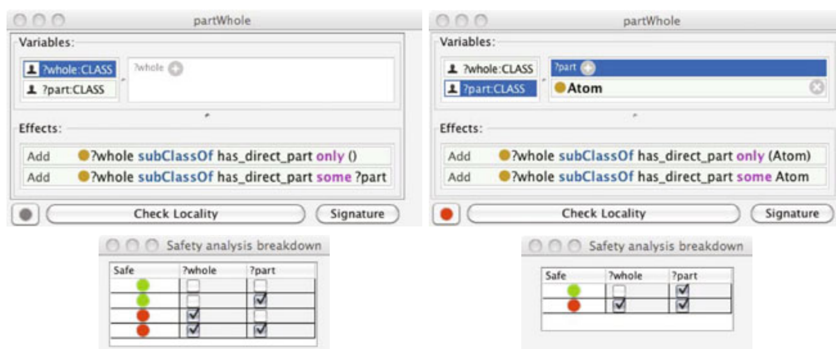

Fig. 1. The *partWhole* pattern instantiation and safety analysis breakdown, without any user assigned variable (left) and with one assigned variable (right). The signature for the safety check is $\mathbf{S} = \{Molecule, Atom\}$.

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 incorporated via renaming of their non logical symbols' (see Section 2 in [7]). In the subsequent years, researchers seemed to focus on pattern categorisation (see, for instance, [8] and its references) aiming to create organised catalogues that engineers could browse and adopt in their knowledge bases. Staab *et al.*, in [9], 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 [10], 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 that can be reused.

Locality has been often used in literature to support re-use through modularization. In [11], for instance, a methodology for extracting the right portion of axioms from an ontology and safely import it inside another one is proposed. This approach has a point of view that is orthogonal to the one in this paper, as it deals with the problem of someone building an ontology trying to re-use as much as possible from third party sources, and to remain coherent with the semantics of what they are re-using. In [11] as well as in other variations based on safe extensions [12], users individuate ontologies they wish to re-use, select the vocabulary to which they wish to restrict and extract, where possible, a proper sub-set of such ontologies to incorporate them in their own. This, despite guaranteeing that users embed (approximately) the minimal required knowledge from the re-used ontologies into the one they are creating, does not help in determining how to extend the ontology and does not prevent the user to subsequently alter, in a non conservative way, the re-used knowledge. Our work, therefore, can be seen as complementary to the re-use by modularization. If knowledge engineers were able to embed patterns in their ontology, encoded in a concrete language such as OPPL, users could then extract from such ontology the modules that are relevant to their purposes (using the methodologies just cited), but, also, decide, perhaps using the methods illustrated here, how safe it is to use the exposed patterns in the ontology extension they are

creating. Ontology engineers, conversely, when creating ontologies meant to be re-used and extended, besides designing them in a modular fashion, could also attach patterns whose usage can be *certified* to be safe with respect to the vocabulary each module uses, as shown in the previous section.

5 Conclusions and Future Work

The main contribution of this paper is the proposal of a framework to achieve safe extensions when re-using OWL ontologies. Our approach is based on knowledge patterns embedded in OWL artefacts using OPPL, and on the recent results in safely extending ontologies. In particular, it extends the notion of locality tests to OPPL patterns and provides an algorithm for determining whether a pattern is local or not with respect to a signature. In addition we provided an extension of this algorithm to compute, in case of patterns that turn out not to be local in general, the combinations of values, w.r.t. an input signature, that result in local instantiations. When implemented inside a tool, these algorithms can help the knowledge engineers in creating and using patterns anticipating the effects on the semantics of the ontologies in which they use the patterns. Thus, when a user decides to use an OPPL pattern wishing to extend an ontology, the tool can point out, given a signature, what are the allowed combinations of value assignments that will produce conservative extensions of the original ontology. Conversely, ontology engineers can evaluate in advance what are the safe uses of the patterns they are designing for extending their own ontologies and even get an idea on how extensible are such ontologies are while still preserving the semantics of the original vocabulary. An implementation of the algorithm is currently available in the PATTERNS plugin for Protégé.

Future work in this field includes the optimisation of this technique together with the exploitation of OPPL features, such as the scoping of variables, in order to achieve safer versions of non local patterns. Another interesting observation is that this work relies on ontologies exposing the patterns for their re-use and evaluates the impact of instantiating them in new ontologies. This leaves out all the ontologies that have already been created without any patterns. The detection and the induction of patterns from existing ontologies becomes, then, a promising research direction. Discovering the presence of regularities in a model and characterising them in the form of a pattern could support ontology design and re-use in many ways:

- The model, or a portion of it, is expressed in the form of patterns and so becomes clearer to understand thanks to their greater level of abstraction
- Portions of a model which are non compliant to the discovered patterns can be individuated more easily and corrected if the pattern is recognized as valid and normative for the model. that is, a ‘house-style’ can be more consistently applied.
- If the induction process reveals that a pattern, which is unsafe according to the criteria provided in this work, has been used inside an ontology, this can lead to the discovery and hopefully to the correction of modeling errors that will enhance the quality of the model itself.

References

[1] Iannone, L., Rector, A.L., Stevens, R.: Embedding knowledge patterns into owl. In: Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 218–232. Springer, Heidelberg (2009)
[2] 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)
[3] Egaña, M., Rector, A.L., Stevens, R., Antezana, E.: Applying ontology design patterns in bio-ontologies. In: Gangemi, A., Euzenat, J. (eds.) EKAW 2008. LNCS (LNAI), vol. 5268, pp. 7–16. Springer, Heidelberg (2008)
[4] Ghilardi, S., Lutz, C., Wolter, F.: Did i damage my ontology? a case for conservative extensions in description logics. In: Doherty, P., Mylopoulos, J., Welty, C.A. (eds.) KR, pp. 187–197. AAAI Press, Menlo Park (2006)
[5] Cuenca Grau, B., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: Theory and practice. J. of Artificial Intelligence Research (JAIR) 31, 273–318 (2008)
[6] Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Just the right amount: extracting modules from ontologies. In: WWW '07: Proceedings of the 16th international conference on World Wide Web, pp. 717–726. ACM, New York (2007)
[7] Clark, P., Thompson, J., Porter, B.W.: Knowledge patterns. In: KR, pp. 591–600 (2000)
[8] Blomqvist, E., Sandkuhl, K.: Patterns in ontology engineering: Classification of ontology patterns. In: ICEIS, vol. (3), pp. 413–416 (2005)
[9] Staab, S., Erdmann, M., Maedche, A.: Engineering ontologies using semantic patterns. In: IJCAI Workshop on E-business & The Intelligent Web (2001)
[10] 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)
[11] Jiménez-Ruiz, E., Cuenca Grau, B., Sattler, U., Schneider, T., Llavori, R.B.: Safe and economic re-use of ontologies: A logic-based methodology and tool support. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 185–199. Springer, Heidelberg (2008)
[12] Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Extracting modules from ontologies: A logic-based approach. In: Stuckenschmidt, H., Parent, C., Spaccapietra, S. (eds.) Modular Ontologies. LNCS, vol. 5445, pp. 159–186. Springer, Heidelberg (2009)