Explicit knowledge engineering patterns with macros

Denny Vrandečić

Institute AIFB, University of Karlsruhe
denny@aifb.uni-karlsruhe.de

Abstract. The web ontology language OWL is still a very young language. Experience with the language will build an increasing pool of ontology design patterns and best practises.
In this paper we introduce macros for OWL ontologies. Macros are able to formally specify and capture design patterns for the knowledge engineering task with OWL ontologies. Thus the user is enabled to conveniently reuse them. Macros lead to an enhanced maintainability of the ontology, to a higher level of abstraction in the specification (and thus closer to the human conceptualization), to a faster and less error-prone creation of the ontology (and thus to lower costs) and to automation of several tedious tasks in the ontology life cycle.
We present example macros, discuss further applications, advantages and problems of macros, and discuss the development of an implementation to enable the usage of macros.

1 Introduction

When the W3C defined the web ontology language OWL [21], they tried to meet a list of requirements stated earlier [10]: extensibility, non-monotonicity, compliance to existing standards like XML or RDF, internationalization and others. They also aimed for a powerful and expressive semantic language, yet retaining decidability. Human readability and ease of use of the serialization were obviously secondary goals, as anyone agrees who had the chance of using OWL-ontologies in RDF/XML-Syntax.

Also one of the goals was to provide a small number of orthogonal features that are easy to combine. Although it is perfectly possible to state certain facts, it sometimes may be very tedious: imagine the following simple sentence: *"One either hates or loves Star Wars"*. The naïve understanding of this sentence has to be expressed by a surprisingly big number of steps:

- We create two classes, *StarWarsHater* and *StarWarsLover*
- We make them both subclasses of *Person*
- We make the two classes disjoint of each other
- We create an anonymous class description that is the union of the two classes
- *Person* is made a sublass of this anonymous class description

We see, that it is perfectly possible to state the simple meaning intended with the given sentence, but it is surely more complicated than it needs to be, prone to errors or omissions, and hard to understand when reading the ontology, as it is far away from our initial intuition of the sentence. Why shouldn't we be able just to state that *Person* has a *Partition* in *StarWarsHater* and *StarWarsLover*?

OWL DL is in a position similar to the one programming languages have been decades ago: the user base is growing rapidly, and the language captures the whole powerful semantics of the description logic $\mathcal{SHOIN}(\mathbf{D})$, just like programming languages were able to capture all possible algorithms. But soon programmers recognized numerous tedious tasks which were repeated over and over again.To ease these tasks, they introduced macros, easy shortcuts for a more complex combination of terms. They were used by the human programmer and expanded automatically in a preprocessing step before sent to the compiler. Thus the level of abstraction on the user's side raised without changing the language, boosting his productivity without forcing him or others to update their tools – just by introducing a local preprocessing step.

In this paper we will suggest to follow the same step that was done in programming languages in OWL as well. Introducing macros will provide several benefits. We will explore how to introduce macros and some of the advantages and disadvantages of applying them. In the next section we will first introduce the most powerful application of macros, by aiming for a higher level of abstraction. Then we will present some obvious and well-known patterns which easily fit into the given framework. In section 4 we show how macros can also be used for several other tasks in the ontology life cycle, tasks that are more specific for macros applied in ontologies. We will discuss several possibilities of implementing the system in section 5 and explore their advantages and problems. Finally we review some related work in section 6, before we close with a short outlook and concluding remarks.

2 Captured knowledge engineering patterns

The literature on describing and capturing patterns for knowledge or ontological engineering is growing increasingly faster in the last few years (see, e.g., [22, 23, 6]). Those patterns often describe how language primitives can be combined in order to achieve a specific, well-defined and formalized goal, like partitions (see subsection 3.1) or similar.

Unlike patterns in software engineering [5], the patterns we speak of in ontological engineering are still extremely simple. Most of them may be implemented as simple macros, A macro, as used within this paper, is defined as a symbol, possibly with parameters, that is replaced in a preprocessing step (called macro expansion) according to a defined set of rules. In OWL ontologies the symbol is an URI, the parameters are bound to the symbol with RDF statements, and the resulting replacement are a set of RDF statements created out of the macro according to its definition. In the next section we will describe a few example patterns. Here we will analyse some generic properties of macros.

2.1 Closing the gap

Ontologies are specifications of conceptualizations [9]. In order to specify the conceptualizations we have to use the means given to us by the chosen ontology language, in our case the web ontology language OWL. The means are the language primitives like *unionOf*, *subPropertyOf* and others [21], and their possible combinations.

Human ontological patterns usually don't map one to one with the primitives of an ontology language. This is because the choice of human ontological patterns is not driven by requirements like decidability of all combinations of patterns or strict formal semantics. But they are in our mind and we are able to share them with other people. This becomes most obvious when teaching students description logics or OWL and looking at the common mistakes students make. Tutorials like [13] and experience reports [20, 4] point us to such mistakes. They are not common because students are equally stupid around the globe, but because there is a mismatch between human knowledge patterns and the offered ontological primitives.

An example of such a mismatch is the use of the universal quantifier. A student who is new to OWL and DL could make the following statement:

$$Horse \equiv \forall hasParent.Horse$$

It is obvious for the student that everything whose parents are horses must be a horse as well. What the student did not think of – and, as said, this is a common mistake – is, that the very paper you are reading would classify as a horse as well, as all its parents are horses – since it has no parents at all. The student, seeing the error, wants to correct it naïvely. He may turn the statement to

$$Horse \equiv \exists hasParent.Horse$$

this time forgetting about mules, who would incorrectly be classified as horses, because one of their parents is a horse (and the other a donkey). What the student actually wanted to say was

$$Horse \equiv \exists hasParent.\top \sqcup \forall hasParent.Horse$$

Now horses must have at least one parent, and all of them have to be horses as well.

The last statement specifies formally the students conceptualization. But as it is no primitive of the ontology language, the mapping of the students conceptualization to the specification is lost. When maintaining the ontology later, another student (or even the same one) has to understand the indented conceptualization captured in this statement and treat it accordingly.

It would be much easier if we just added a new symbol. Imagine the following statement:

$$Horse \equiv \nabla hasParent.Horse$$

Furthermore, we define $\nabla R.C$ colloquially as "every R must be a C" and specifying its semantics formally as $\exists R.\top \sqcup \forall R.C$. The colloquial definition of ∇ is the

same as the formal definition of $\forall$ – the difference is, that the colloquial definition implicitly includes the $\exists$-part, because, as we have seen from the aforementioned experience reports, humans tend to do it anyway. A system including both, the ∇ macro as well as the $\forall$ constructor would need to provide an alternative, colloquial definition of $\forall$ like "if there is an R, it must be a C").

The new symbol, ∇, is just a macro: it does not add to the expressive power of the language, nor does it change any of its computational properties. It's just syntactic sugar. But it still allows the user to work on a level closer to his own conceptualization than before.

2.2 Advantages of macros

If a knowledge engineering pattern appears repeatedly in an ontology, using a macro will greatly enhance the readability of the ontology. The ontology will not become only shorter and more crisp: the biggest advantage is that the macros guarantee locality, as they are only one symbol, usually with parameters. Imagine a pattern consisting of several ontological primitives like the one described in the previous section. As the pattern has several syntactically independent parts, they may appear in the written down ontology in arbitrary order (especially when exported from a graphical ontology engineering tool). Even in an ontology with a mere hundred axioms, the human reader will have a hard time to figure out the applied knowledge engineering patterns without the help of a dedicated tool searching for such patterns. But when using macros the pattern is represented by a single axiom and does not litter the ontology with one sole concept specified by several syntactical fragments.

Using macros also reduces the number of possible errors. As we don't have to make a manual translation of the pattern we want to apply to the language primitives, and as we reduce the number of axioms that need to be specified, we make the ontology more robust. As these manual translations are tedious as well as demanding with regards to the skills of the user – we need users trained well enough with the patterns in order to apply them – using macros actually make it also more enjoyable to build ontologies, as the user does not have to bother with several low level problems. This step compares to the advent of high level languages in software engineering. The maintainability of the ontology is enhanced as well, due to the smaller number of axioms, locality of the macros and explicit complex semantics of the macros.

As we will see in section 6, tools like Protégé [18] already offer the user the ability to apply certain knowledge engineering patterns. But this ability is constrained to the implemented patterns (whereas the macro system is extensible) and the information about the applied patterns is not saved explicitly in the ontology, and thus the higher maintainability and readability is lost (although the advantage of making less errors when creating the ontology is retained).

In section 4 we describe further advanced scenarios where macros will proof useful.

2.3 Disadvantages of macros

Introducing macros to a project also comes with a list of disadvantages and problems: macros introduce a preprocessing step between the human-maintained ontology and the one used by the application. If the application issues messages about the ontology, for example error messages, it will refer to a different ontology than the one the user actually knows (even though they are both semantically equivalent, they will use different constructs). If, in the given example, the reasoner would say, that $\exists hasParent.\top$ is inconsistent, we would not find this construct in our ontology, as it was automatically created out of $\nabla hasParent.Horse$ description.

Maybe reasoners and – even more important – ontology engineering suites will learn to deal with macros (probably starting with a small and generic set of macros instead of enabling them all) and treat them transparently, thus hiding the preprocessing step and the two different ontologies. But this is no condition for the successful usage of macros.

A further disadvantage is the increased number of language elements one has to deal with when working with ontologies. But practise and experience will probably lead to a rather consistent set of generic macros that will find widespread use in the community, some of them probably even more widespread than certain language primitives. But if one does not want to cope with certain macros he always has the possibility to expand them and work on the set of ontology primitives the language provides.

2.4 Project-specific macros

One further advantage of introducing macros instead of implementing new language primitives is that everyone could create new macros as the need arises in a specific project. If an ontology engineer finds that certain ontological patterns appear repeatedly in a project, due to the domain's ontological structure, she may create project- or domain-specific macros. This will bring all the advantages described above: a smaller gap between the conceptualization and its specification, enhanced readability and maintainability, less opportunities for errors and higher speed of development and thus reduction of costs.

Although using project-specific macros will lead to a higher long-term efficiency, it may be problematic for users who only want to dip into an ontology briefly, as they would have to understand the project-specific macros first. Also there is almost no hope for generic tools to introduce appropriate treatment of project-specific macros – but again, we may simply use the macro tool to expand the macros in an ontology and then work with the standard language ontology. We also hope that some ontology tools will, instead of just implementing a set of generic macros, implement the macro framework sketched in this paper and thus enable all users to invent and use their own macros.

3 Example macros

In this section we present a small set of examples. This list is neither intended to be complete nor representative. As mentioned before, the ontological macros were explicitly meant to be extended even for a local project, and don't have to be generally applicable knowledge engineering patterns (although we expect a library of such patterns to emerge from more intensive use of OWL in the near future).

3.1 Partition

A partition of C into $D_1, D_2, D_3, ...$ requires that every instance of C must also be an instance of exactly one of D_1 or D_2 or D_3... This is a very common pattern and may be found in diverse domains like biology, where every instance of a biological order must be an instance of a biological family belonging to this order, or in universities, where we may partition the members of a faculty in professors, students and post-docs, and so on. In DAML+OIL [15], there was the *disjoint union*-language primitive that had the same semantics.

So we define

$$C \underline{\sqcup} D_1, D_2, ..., D_n$$

as

$$C \equiv D_1 \sqcup D_2 \sqcup ... \sqcup D_n$$

$$D_i \sqcap D_j \equiv \bot (\forall i, j | 1 \leq i < j \leq n)$$

3.2 Local Unique Name Assumption

The Non-Unique Name Assumption is one of the pillars of the Semantic Web, and within the open world of the Semantic Web it is a highly needed one. It means, that two names may refer to the same thing. As we don't register all names centrally (and as we cannot register all meanings of names anyway), the only way to deal with several names for one meaning is to accept that we may not assume that two names refer to different things unless stated so explicitly.

Although this is necessary in the context of the Semantic Web, most ontologies today are created locally (but with the Semantic Web in mind): for example, if we write an ontology about all the members of a course, it is obvious (but still often forgotten) that all names we create refer to different individuals, that we actually have some local unique name assumption. Stating this explicitly is a very tedious and error-prone task. If we add a new student, we have to assert he is different from all the other students (see listing 1.1). Deleting a student is even more difficult, as we must assure to delete all assertions claiming the difference from the deleted student. This soon will become a hard to handle mess.

Listing 1.1. Using *differentFrom*

```
<Student rdf:ID="Tony" />
```

```
<Student rdf:ID="Martin">
  <owl:differentFrom rdf:resource="#Tony"/>
</Student>
<Student rdf:ID="Mary">
  <owl:differentFrom rdf:resource="#Tony"/>
  <owl:differentFrom rdf:resource="#Martin"/>
</Student>
```

But OWL already introduces some kind of macro by allowing to use the *AllDifferent* construct. But as we can see in listing 1.2, this also brings the disadvantage of having to handle individuals at two distinct locations in the ontology, the list of *AllDifferent*-Individuals and the individual-declaration itself (if writing by hand you could keep this at the same place by declaring the individuals inside the *AllDifferent*-construct. But tools who read and write the ontology would not have to stick to this style guide and could rewrite the ontology so that there are at two different locations again).

Listing 1.2. Using *AllDifferent*

```
<Student rdf:ID="Tony" />
<Student rdf:ID="Martin" />
<Student rdf:ID="Mary" />
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <Student rdf:about="#Tony" />
    <Student rdf:about="#Martin" />
    <Student rdf:about="#Mary" />
  </owl:distinctMembers>
</owl:AllDifferent>
```

Instead we could use a macro that declares that all individuals in this ontology with a certain namespace have to be different.

Listing 1.3. Using *localUNA*

```
<macro:localUNA rdf:resource="#" />
<Student rdf:ID="Tony" />
<Student rdf:ID="Martin" />
<Student rdf:ID="Mary" />
```

As *Tony*, *Martin* and *Mary* are all of the local base namespace *"#"*, they would all be asserted as mutually different individuals.

Semantically all the listings in this section were equivalent. Using macros – either the predefined one from the OWL-specification or the ones introduced in this paper – leads obviously to the advantages in maintainability and readability described in section 2.

3.3 transitiveOver

Within the SEKT project[1] the upper level ontology PROTON[2] is being developed [24]. Whereas the ontology is mostly within OWL DLP [8], it introduces one extension to the formal semantics of OWL [21] which is the *transitiveOver* relation. *transitiveOver* is a relation between two relations p and q such that if p *transitiveOver* q, a p b and b q c also a p c must hold. An obvious example is the relation *locatedIn* which is *transitiveOver* the relation *subRegionOf*. So if we know that *Ireland subRegionOf Europe* and the *ISWC2005 locatedIn Ireland*, we would also infer that *ISWC2005 locatedIn Europe* is true.

Expressing this in OWL DL is not possible, as this needs a logic capable of property chaining. With macros we have two different options of solving this problem: using materialization or using SWRL.

Materialization means that we use the macro facilities in order to explicitly assert the statements that we would have to infer otherwise. So, if we had an ontology with the above example, the macro expander would delete the actual macro axiom (*locatedIn transitiveOver subRegionOf*) and instead insert the inferred new axiom *ISWC2005 locatedIn Europe*. In subsection 4.2 we discuss the issue of materialization.

Using the Semantic Web Rule Language SWRL [14] instead would circumvent the need for materialization, but requires a reasoner able to deal with the resulting SWRL enhanced ontology, like Hoolet[3] or KAON2[4]. The expansion would have to turn a triple like

Listing 1.4. Declaration of *transitiveOver*

```
<owl:ObjectProperty rdf:resource="locatedIn">
  <macro:transitiveOver rdf:resource="subRegionOf" />
</owl:ObjectProperty>
```

into the following statement (in RDF-syntax).

Listing 1.5. SWRL-description of *transitiveOver*

```
<owlr:Variable rdf:ID="x" />
<owlr:Variable rdf:ID="y" />
<owlr:Variable rdf:ID="z" />
<owlr:Rule>
 <owlr:antecedent rdf:parseType="Collection">
  <owlr:individualPropertyAtom>
   <owlr:propertyPredicate rdf:resource="#locatedIn" />
   <owlr:argument1 rdf:resource="#x" />
   <owlr:argument2 rdf:resource="#y" />
  </owlr:individualPropertyAtom>
```

[1] http://www.sekt-project.com
[2] http://proton.semanticweb.org
[3] http://owl.man.ac.uk/hoolet/
[4] http://kaon2.semanticweb.org

```
  <owlr:individualPropertyAtom>
   <owlr:propertyPredicate rdf:resource="#subRegionOf" />
   <owlr:argument1 rdf:resource="#y" />
   <owlr:argument2 rdf:resource="#z" />
  </owlr:individualPropertyAtom>
 </owlr:antecedent>
 <owlr:consequent rdf:parseType="Collection">
  <owlr:individualPropertyAtom>
   <owlr:propertyPredicate rdf:resource="#locatedIn" />
   <owlr:argument1 rdf:resource="#x" />
   <owlr:argument2 rdf:resource="#z" />
  </owlr:individualPropertyAtom>
 </owlr:consequent>
</owlr:Rule>
```

4 Advanced applications

Macros can also be used for further applications than mere modelling. In this section we will skim over different use cases where macros will proof helpful. We do not expect this list to be complete. Whereas most of the aforementioned advantages were true for macros used in programming languages as well, and thus well-known, the scenarios described here are rather unique for macros in ontologies and not tested yet in comparable settings.

4.1 Language weakening

Right now there exists no efficient DL reasoner able to deal with the whole OWL DL language. Most reasoners have some kind of constraints, like not being able to deal with nominals. Also it may be beneficial to give up on certain costly language features and turn them into semantically inequivalent, weaker counterparts, but therefore gaining efficiency in reasoning, so called approximate reasoning.

For example, a nominal declaration $C = \{o_1, o_2, o_3\}$ could be turned into a simple class membership declaration $C(o_1), C(o_2), C(o_3)$, thus loosing the closed world assertion of the first statement, but turning a $\mathcal{SHOIN}(\mathbf{D})$-Ontology into a semantically close $\mathcal{SHIN}(\mathbf{D})$-Ontology (this is further described in [11]), effectively turning an OWL DL ontology into an OWL DLP ontology [25]. Reasoners, who can handle OWL DLP could now work with the resulting ontology.

4.2 Materialization

As already stated in subsection 3.3, macro expansion could also be used for materialization, that is, the explicit assertion of inferrable facts. This can be good for a variety of reasons: speeding up inferences, substituting the lack of an

appropriate reasoner for the desired expressivity of language or evaluation and maintenance tasks. But materialization has to be used carefully: it can easily lead to quite an explosion in size of the assertional knowledge and cause further problems in the context of the Semantic Web. These issues remain open for discussion.

4.3 Using more expressive languages

OWL is not the most expressive knowledge representation language. Other languages, especially based on larger fragments of First Order Logic than OWL DL, were used for years and are able to capture much more refined specifications than OWL DL allows. RDF [16] was created as a general framework for various KR languages that build on a common syntactic and semantic foundation for the exchange of assertional data. Both RDF Schema [1] and OWL are based on this.

A simple example is extending OWL DL with qualified number restrictions, that is, stating that an object property has a cardinality with regards to a certain concept, for example there could be the role *member* between *band* and *musician* without any cardinality restrictions, but one could still state that the role is constrained to a maximal cardinality of 1 to the concept *drummer*, which means that a *band* may have several *member*s, but only one of them may be a *drummer*.

OWL DL is not able to express this, but DAM+OIL [15] is, with the *maxCardinalityQ*-restriction. So when reusing a DAML+OIL- ontology one does not need to loose the additional information, but may instead declare an appropriate translation to OWL DL, thus being able to use the translated ontology with standard OWL tools and still keep the DAML-ontology as the source ontology.

4.4 Several flavours of an ontology

As seen in the last subsection, with macros we are able to offer several versions of the same ontology, and still keep the maintenance costs low because we only work on one reference version. This version may specify the conceptualization as close as possible, regardless of complexity and size considerations, with many axioms (a so-called heavyweight ontology), and then one may automatically create a lightweight version of it, removing some constructors due to the performance hit they would introduce or due to the available tools' abilities.

We could also decide to expand the same macro in different ways, depending on the task for the expanded ontology. We could use a more heavyweight version of an ontology for the purpose of mapping to a second ontology or as a reference for the original ontology engineering, but create a second one for the on-line-inferencing. Remembering the example in subsection 3.3, we could create different ontologies depending on the tools we are going to plan to use. Whereas Sesame [2] has a native support for the *transitiveOver*-semantics, we would need the SWRL-translation to use it with a SWRL-Reasoner like Hoolet, or a materialization process for applying it with a simple triplestore like Jena[5].

[5] http://jena.sourceforge.org

5 Implementation

There are several possibilities to implement the proposed system and integrate it into existing methodologies, tool chains or IDEs. In this section we will sketch and discuss these alternatives. We do not discuss user interfaces for creating and using macros. These requirements follow from the rest of the paper:

- easy to use as a preprocessing system. Either as a library that can be accessed from another more complex system or from the command line. The result must be a standard compliant ontology.
- fast to implement. Macros are a basic and easy feature, the implementation should not be complicated.
- easy to extend. It should be possible to create new patterns for the macro system without having to dwell deep into the code or having to learn a lot of background skills.
- be amenable to round-trip development. The system should not only be able to expand macros, but the design should allow for an extension to detect patterns in the ontology and contract them to macros again.

Now we explore the different possibilities to develop the proposed system.

XSLT The standard exchange format for OWL ontologies is RDF/XML, an XML-syntax. The W3C has put a lot of thought into creating a language for the transformation of XML-documents, called XSLT [3]. There are numerous implementations for using XSLT – so why not use it?

XSLT has several problems with regards to OWL. First, RDF/XML-serialization does not lend at all to easy transformation via XSLT, as there are numerous different possibilities to serialize the same semantic content. This problem could be overcome by using the OWL XML presentation syntax [12], which did not yet succeed to become very wide spread though it offers quite some advantages over the RDF/XML-syntax. We have provided a tool that is able to translate from OWL XML presentation syntax to RDF/XML- serialization and back[6].

But even then XSLT has severe drawbacks. Instead of simply declaring the macro, we would have to program the transformations in a functional programming language, which is a precious skill. And the detection of the patterns and translation back to macros would need to be written and maintained separately. Otherwise it would be a very welcome choice.

A dedicated tool The second option is writing a dedicated tool that reads the ontology, discovers the macros defined in the tool and replaces them accordingly to the programmed algorithms. It would be a pragmatic and quick solution, but not very extensible, as new patterns would need to be programmed anew. Having project-specific macros is very hard if the tool is actively developed, as the user

[6] http://km.aifb.uni-karlsruhe.de/owlrdf2owlxml

would need to merge the changes from the main branch of the tool with the project-specific ones. It is impossible to use this tool for roundtrip-engineering without writing new algorithms for detecting the patterns and reducing them to macros.

A dedicated tool with externally declared rules The third option is creating a tool that reads the patterns declared in external files. This certainly raises the biggest effort in the first run, but will pay off later due to the ease of creating new macros. As both, the patterns and the macros are specified in the external files, the system may also be extensible to a roundtrip-engineering tool (although this certainly would require further research). The tool will provide an API so other tools could reuse it as a library and perform the macro expansion transparently.

Using SWRL and Metamodelling Most of the functionality we have seen in this paper translate well to a rule-based system. Why not reuse the work done for rules in the Semantic Web instead of creating new tools?

Macro expansion is different from simple rule application. Expanding a macro is like materializing the head of the rule and removing its body. A system based on SWRL does not work this way: it infers the head, but does not touch the body. Further on, most of the rules we propose in our macros would be Second Order Logic rules, as they often include OWL language construct. Research on metamodelling like [17] explicitly disallow using language constructs in metamodelling.

Still, lessons learned in the syntactic definition of SWRL will surely help a lot in designing a macro specification ontology a macro tool would be based on. This also should ease learning, as it reuses many known ideas from other Semantic Web technologies.

6 Related work

Further ontological patterns may be found in several publications like [7, 22, 23, 6]. Also other ontology languages like DAML+OIL [15] contain some of the patterns described here as language primitives, or else may be used as sources of inspiration for further patterns.

In Protégé [18] the ontology engineer may use wizards to quickly apply some ontological patterns. This is a great help for the engineer, and lowers the number of possible mistakes drastically. There exists for example a wizard for the quick creation of a partition. Other shortcuts within the Protégé IDE (like creation of closures, or – probably best known – the definition of either *necessary* or *necessary and sufficient* conditions in class descriptions, which actually are not OWL primitives but are idioms translated within the IDE) also allow for some of the advantages gained with the help of macros as described herein.

But tools like the wizards allow only for the simple initial application of a knowledge engineering pattern. As they are not explicitly included within the

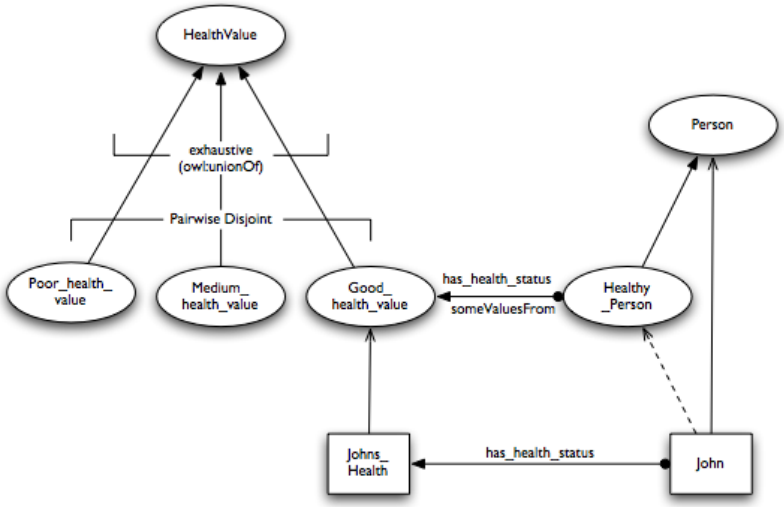

Fig. 1. Figure from [19], depicting a higher level constructor (left side)

ontology the information about applied patterns is lost to the IDE. Changes in parts of the ontology created with the help of wizards do not happen at the same abstraction level as their creation – this means, the ontology engineer must have a deep understanding of the applied wizards in order to manage and maintain the wizards work successfully afterwards. The information about the applied pattern is lost after its application.

Finally, adding new patterns in Protégé is not possible without adding source code, whereas in the approach presented in the paper only a further pattern must be defined and the rest of the work is handled by the system. This is especially helpful for project-specific patterns, where the work needed to adapt a Ontology Engineering tool like Protégé is much more expensive than the expected benefit.

Another example of actually using higher abstractions than the OWL DL constructs is given by the W3C Semantic Web Best Practise group in [19]: not only does the note describe several patterns which could be used in order to create further macros, but they also introduce a graphical notation for taxonomical relationships that carry information such as exhaustive decomposition of classes which is not representable by a single OWL constructor. Interestingly this means that in their notes the W3C introduced a graphical notation for a complex relation even before such a representation for simple ontology primitives. This hints at the relative importance of such constructors. An example is shown in figure 1.

7 Conclusions

In this paper we have described and discussed the introduction of macros to OWL. Macros allow for a higher level of abstraction, enhance maintainability, increase the speed of ontology creation, and ease several tasks in the ontology life cycle. Some example macros and an elaboration of further applications of the macro system highlight the benefit of the suggested system.

Next steps are implementing a macro tool and collecting experiences with its usage, defining the macro specification ontology and create a comprehensive library of interesting, generic macros. Then an integration of the macro system in IDEs like Protégé, SWOOP[7] or OntoStudio[8] has to follow, thus introducing macros to a wider audience. Work has to be done to find out how macros can be visualized and handled within IDEs, and if this is possible for generic macros as described herein. With the experience gained by now, the feasibility of a round-trip engineering tool for macros should become clear.

It is hard to create and manage ontologies on the Semantic Web. We showed that macros help in many of the involved tasks in several different steps of the ontology life cycle. Macros sweeten the daily work of the ontology engineer by providing syntactic sugar for the Semantic Web.

Acknowledgements

Research reported in this paper has been partially financed by the EU in the IST-2003-506826 project SEKT Semantically Enabled Knowledge Technologies (http://www.sekt-project.com).

References

1. D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation 10 February 2004.
2. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proc. of the First International Semantic Web Conference, Sardinia, Italy*, pages 54–68, 2002.
3. J. Clark. XSL Transformations. W3C Recommendation 16 November 1999, 1999. available at http://www.w3.org/TR/xslt.
4. A. Cregan, M. Mochol, D. Vrandečić, and S. Bechhofer. Pushing the limits of owl, rules and protégé. a simple example. In *Proceedings of OWL Experiences and Directions Workshop, Galway, Ireland, November 2005*, 2005.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Mass., 1995.
6. A. Gangemi, C. Catenacci, and M. Battaglia. Inflammation Ontology Design Pattern: an exercise in building a core biomedical ontology with Descriptions and Situations. In D. Pisanelli, editor, *Ontolgies in Medicine*, pages 64–80. IOS Press, 2004.

[7] http://www.mindswap.org/2004/SWOOP/
[8] http://www.ontoprise.de/content/e3/e43/index_eng.html

7. A. Gómez-Pérez, M. Fernández-López, and O. Corcho. *Ontological Engineering*. Advanced Information and Knowlege Processing. Springer, 2003.
8. B. Grossof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proceedings of WWW 2003*, Budapest, Hungary, May 2003.
9. T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
10. J. Heflin. OWL Web Ontology Language Use Cases and Requirements. W3C Recommendation 10 February 2004.
11. P. Hitzler and D. Vrandečić. Resolution-based approximate reasoning for owl dl. In *Proc. of the 4th International Semantic Web Conference, Galway, Ireland*, 2005.
12. M. Hori, J. Euzenat, and P. Patel-Schneider. Owl web ontology language xml presentation syntax. Note, Worldwide web consortium, Cambridge (MA US), 2003.
13. M. Horridge, H. Knublach, A. Rector, R. Stevens, and C. Wroe. A practical guide to building OWL ontologies using the Protégé-OWL plugin and CO-ODE tools, 2004. University of Manchester.
14. I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: a semantic web rule language combining OWL and RuleML, 2003.
15. I. Horrocks, F. van Harmelen, P. Patel-Schneider, T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, D. Fensel, R. Fikes, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, and L. A. Stein. DAML+OIL (March 2001), 2001. Joint Committee, http://www.daml.org/2001/03/daml+oil-index.
16. F. Manola and E. Miller. Resource Description Framework (RDF). primer. W3C Recommendation 10 February 2004.
17. B. Motik. On the properties of metamodeling in OWL. In *Proceedings of the 4th International Semantic Web Conference, Galway, Ireland*, 2005.
18. N. Noy, R. Fergerson, and M. Musen. The knowledge model of Protégé-2000: Combining interoperability and flexibility. In R. Dieng and O. Corby, editors, *Proc. of the 12th EKAW*, volume 1937 of *LNAI*, pages 17–32, Juan-les-Pins, France, 2000. Springer.
19. A. Rector. Representing specified values in OWL: "value partitions" and "value sets". W3C Working Group Note 17 May 2005, 2005. available at http://www.w3.org/TR/swbp-specified-values/.
20. A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In E. Motta, N. R. Shadbolt, and A. Stutt, editors, *Proc. of the 14th EKAW*, volume 3257 of *LNCS*, pages 63–81. Springer, 2004.
21. M. K. Smith, C. Welty, and D. McGuinness. OWL Web Ontology Language Guide, 2004. W3C Recommendation 10 February 2004, available at http://www.w3.org/TR/owl-guide/.
22. S. Staab, M. Erdmann, and A. Maedche. Engineering ontologies using semantic patterns. In A. P. . D. O'Leary, editor, *Proceedings of the IJCAI-01 Workshop on E-Business & the Intelligent Web, Seattle, WA, USA, August 5, 2001*, 2001.
23. V. Svatek. Design Patterns for Semantic Web Ontologies: Motivation and Discussion. In *Witold Abramowicz (ed.), Business Information Systems, Proceedings of BIS 2004, Poznan, Poland*, 2004.
24. I. Terziev, A. Kiryakov, and D. Manov. Base upper-level ontology (bulo) guidance. SEKT deliverable 1.8.1, Ontotext Lab, Sirma AI EAD (Ltd.), 2004.
25. D. Vrandečić, P. Haase, P. Hitzler, Y. Sure, and R. Studer. DLP – An introduction. Technical report, AIFB, Universität Karlsruhe, FEB 2005.