Practical Ontology Pattern Instantiation, Discovery, and Maintenance with Reasonable Ontology Templates

Martin G. Skjæveland(✉), Daniel P. Lupp, Leif Harald Karlsen, and Henrik Forssell

Department of Informatics, University of Oslo, Oslo, Norway
{martige,danielup,leifhka,jonf}@ifi.uio.no

Abstract. Reasonable Ontology Templates (OTTR) is a language for representing ontology modelling patterns in the form of parameterised ontologies. Ontology templates are simple and powerful abstractions useful for constructing, interacting with, and maintaining ontologies. With ontology templates, modelling patterns can be uniquely identified and encapsulated, broken down into convenient and manageable pieces, instantiated, and used as queries. Formal relations defined over templates support sophisticated maintenance tasks for sets of templates, such as revealing redundancies and suggesting new templates for representing implicit patterns. Ontology templates are designed for practical use; an OWL vocabulary, convenient serialisation formats for the semantic web and for terse specification of template definitions and bulk instances are available, including an open source implementation for using templates. Our approach is successfully tested on a real-world large-scale ontology in the engineering domain.

1 Introduction

Constructing sustainable large-scale ontologies of high quality is hard. Part of the problem is the lack of established tool-supported best-practices for ontology construction and maintenance. From a high-level perspective [12], an ontology is built through three iterative phases:

1. Understanding the target domain, e.g., the domain of pizzas
2. Identifying relevant abstractions over the domain, e.g., "Margherita is a particular Italian pizza with only mozzarella and tomato"
3. Formulating the abstractions in a formal language like description logics; here an adapted excerpt taken from the well-known Pizza ontology tutorial:[1]

$$\mathsf{Margherita} \sqsubseteq \mathsf{NamedPizza} \sqcap \exists\, \mathsf{hasCountryOfOrigin}.\{\mathsf{Italy}\} \quad (1)$$

$$\mathsf{Margherita} \sqsubseteq \exists\, \mathsf{hasTopping}.\mathsf{Mozzarella} \sqcap \exists\, \mathsf{hasTopping}.\mathsf{Tomato} \quad (2)$$

$$\mathsf{Margherita} \sqsubseteq \forall\, \mathsf{hasTopping}.(\mathsf{Mozzarella} \sqcup \mathsf{Tomato}) \quad (3)$$

[1] https://protege.stanford.edu/ontologies/pizza/pizza.owl.

© Springer Nature Switzerland AG 2018
D. Vrandečić et al. (Eds.): ISWC 2018, LNCS 11136, pp. 477–494, 2018.
https://doi.org/10.1007/978-3-030-00671-6_28

This paper concerns the third task and targets particularly the large gap that exists between how domain knowledge facts are naturally expressed, e.g., in natural language, and how the same information must be recorded in OWL. The cause of the gap is the fact that OWL at its core supports only unary and binary predicates (classes and properties), and offers no real mechanism for user-defined abstractions with which recurring modelling patterns can be captured, encapsulated, and instantiated. The effect is that every single modelled statement no longer remains a coherent unit but must be broken down into the small building blocks of OWL. And as there is no trace from the original domain statement to the ontology axioms, the resulting ontology is hard to comprehend and difficult and error-prone to manage and maintain.

As a case in point, the Pizza ontology contains 22 different types of pizzas, all of which follow the same pattern of axioms as the encoding of the Margherita pizza seen above. For both the user of the ontology and the ontology engineer this information is opaque. The axioms that make out the instances of the pattern are all kept in a single set of OWL axioms or RDF triples in the same ontology document. Since the pizza pattern is not represented as a pattern anywhere, tasks that are important for the efficient use and management of the ontology, such as finding pattern instances and verifying consistent use of the pattern, i.e., understanding the ontology and updating the pattern, may require considerable repetitive and laborious effort.

In this paper we present *Reasonable Ontology Templates* (OTTR), a language for representing ontology modelling patterns as parameterised ontologies, implemented using a recursive non-cyclic macro mechanism for RDF. A pattern is instantiated using the macro's succinct interface. Instances may be *expanded* by recursively replacing instances with the pattern they represent, resulting in an ordinary RDF graph. Section 2 presents the fundamentals of the OTTR language and exemplifies its use on the pizza pattern. Ontology templates are designed to be practical and versatile for constructing, using and maintaining ontologies; the practical aspects of using templates are covered in Sect. 3. Section 4 concerns the maintenance of ontology template libraries. It presents methods and tools that exploit the underlying theoretical framework to give sophisticated techniques for maintaining template libraries and ultimately the ontologies built from those templates. We define different relations over templates and show how these can be used to define and identify imperfections in a template library, such as redundancy, and to suggest improvements of the library. We believe ontology templates can be an important instrument for improving the efficiency and quality of ontology construction and maintenance. OTTR templates allow the design of the ontology, represented by a relatively small library of templates, to be clearly separated from the bulk content of the ontology, specified by a large set of template instances. This, we believe, supports better delegation of responsibility in ontology engineering projects, allowing ontology experts to build and manage a library of templates and domain experts to provide content in the form of structurally simple template instances. To support this claim we report in Sect. 5 from successful experiments on the use of ontology templates to build

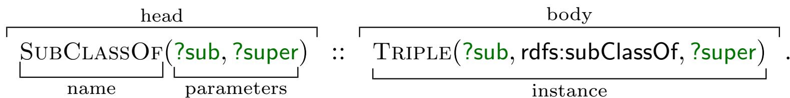
and analyse Aibel's large-scale Material Master Data (MMD) ontology. We compare our work with existing approaches in Sect. 6 and present ideas for future work in Sect. 7.

2 Reasonable Ontology Templates Fundamentals

In this section we develop the fundamentals for OTTR templates as a generic macro mechanism adapted for RDF.

An *OTTR template* T consists of a *head*, head(T), and a *body*, body(T). The body represents a *parameterised* ontology pattern, and the head specifies the template's name and its parameters, param(T). A *template instance* consists of a template name and a list of arguments that matches the template's specified parameters and represents a replica of the template's body pattern where parameters are replaced by the instance's arguments. The template body comprises only template instances, i.e., the template pattern is recursively built up from other templates, under the constraint that cyclic template dependencies are not allowed. There is one special *base template*, the TRIPLE *template*, which takes three arguments. This template has no body but represents a single RDF triple in the obvious way. *Expanding* an instance is the process of recursively replacing instances with the pattern they represent. This process terminates with an expression containing only TRIPLE template instances, hence representing an RDF graph.

Example 1. The SUBCLASSOF template is a simple representation of the rdfs:subClassOf relationship. It has two parameters, ?sub and ?super, and a body containing a single instance of the TRIPLE template.



An example instance of this template is SUBCLASSOF(:Margherita, :NamedPizza); it expands, in one step, to a single TRIPLE instance which represents the (single triple) RDF graph ⟨:Margherita, rdfs:subClassOf, :NamedPizza⟩.

Each template parameter has a *type* and a *cardinality*. (If these are not specified, as in Example 1, default values apply.) The type of the parameter specifies the permissible type of its arguments. The available types are limited to a specified set of classes and datatypes defined in the XSD, RDF, RDFS, and OWL specifications, e.g., xsd:integer, rdf:Property, rdfs:Resource and owl:ObjectProperty. The OWL ontology at ns.ottr.xyz/templates-term-types.owl declares all permissible types and organises them in a hierarchy of *subtypes* and *incompatible* types, e.g., owl:ObjectProperty is a subtype of rdf:Property, and xsd:integer and rdf:Property are incompatible. The most general and default type is rdfs:Resource. This information is used to type check template instantiations; a parameter may not be instantiated by an argument with an incompatible type.

ObjectAllValuesFrom(?X : 1 nonLiteral, ?P : 1 property, ?R : 1 nonLiteral)
:: (?X, rdf:type, owl:Restriction), (?X, owl:onProperty, ?P), (?X, owl:allValuesFrom, ?R) .

SubObjectAllValuesFrom(?X : 1 class, ?P : 1 objectProperty, ?R : 1 class)
:: SubClassOf(?X, _:b1), ObjectAllValuesFrom(_:b1, ?P, ?R) . (4)

ObjectUnionOf(?X : 1 nonLiteral, ?union : + class)
:: (?X, rdf:type, owl:Class), (?X, owl:unionOf, ?union) .

Fig. 1. Basic OWL OTTR templates

The cardinality of a parameter specifies the number of required arguments to the parameter. There are four cardinalities: *mandatory* (written 1), *optional* (?), *multiple* (+), and *optional multiple* ($*$), which is shorthand for ? and + combined. *Mandatory* is the default cardinality. Mandatory parameters require an argument. Optional parameters permit a missing value; *none* designates this value. If *none* is an argument to a mandatory parameter of an instance, the instance is ignored and will not be included in the expansion. A parameter with cardinality *multiple* requires a list as its argument. Instances of templates that accept list arguments may be used together with an *expansion mode*. The mode indicates that the list arguments will in the expansion be used to generate multiple instances of the template. There are two modes: *cross* (written ×) and *zip* (z). The instances to be generated are calculated by temporarily considering all arguments to the instance as lists, where single value arguments become singular lists. In cross mode, one instance per element in the cross product of the temporary lists is generated, while in zip mode, one instance per element in the zip of the lists is generated. List arguments used without an expansion mode behave just like regular arguments. Parameters with cardinality *optional multiple* also accept *none* as a value.

Example 2. Figure 1 contains three examples of OTTR templates that capture basic OWL axioms or restrictions, and exemplify the use of types and cardinalities. The template SubObjectAllValuesFrom represents the pattern ?X $\sqsubseteq$ $\forall$?P.?R and is defined using the SubClassOf and ObjectAllValuesFrom templates. Note that we allow a Triple instance to be written without its template name. The parameters of SubObjectAllValuesFrom are all mandatory, and have respectively the types class, objectProperty and class. The ObjectUnionOf template represents a union of classes. Here the parameter types are nonLiteral and class, where the latter has cardinality *multiple* in order to accept a list of classes. The type of the first parameter, nonLiteral, prevents an argument of type literal.

Example 3. The pizza pattern presented in the introduction is represented as an OTTR template in Fig. 2(a) together with two example instances. The template takes three arguments: the pizza, its optional country of origin, and its list of toppings. The cross expansion mode (×) on the SubObjectSomeValuesFrom

instance causes it to expand to one instance per topping in the list of toppings, e.g., for the first example instance:

- SubObjectSomeValuesFrom(:Margherita, :hasTopping, :Mozzarella) and
- SubObjectSomeValuesFrom(:Margherita, :hasTopping, :Tomato),

creating an existential value restriction axiom for each topping, which results in the set of axioms seen in (2) of the pizza pattern in Sect. 1. By joining SubObjectAllValuesFrom and ObjectUnionOf with a blank node (_:b1), we get the universal restriction to the union of toppings (3). Note that the list of toppings is used both to create a set of existential axioms *and* to create a union class. The optional ?Country parameter behaves so that the SubObjectHasValue instance is not expanded but removed in the case that ?Country is *none*. The first NamedPizza instance in the figure represents exactly the same set of axioms as the listing in Sect. 1.

We conclude this section with the remark that it is in principle possible to choose a "base" other than RDF for OTTR templates, with suitable changes to typing and to which templates are designated as base templates. For instance, we could let templates such as SubClassOf, SubObjectAllValuesFrom, etc. be our base templates, to form a foundation based on OWL. These templates could then be directly translated into corresponding OWL axioms in some serialisation format. (An OTTR template can also be defined as a parameterised Description Logic knowledge base [2].) We have chosen here to base OTTR templates on RDF as this makes a simpler base, and broadens the application areas of OTTR templates, while still supporting OWL.

3 Using Ontology Templates

In this section we present the resources available to enable efficient and practical use of ontology templates: serialisation formats for templates and template instances, tools, formats and specifications that can be generated from templates, and online resources.

Languages. There are currently three serialisation formats for representing templates and template instances: stOTTR, wOTTR, and tabOTTR.

stOTTR[2] is the format used in the examples of Sect. 2 and is developed to offer a compact way of representing templates and instances that is also easy to read and write.

However, to enable truly practical use of OTTR for OWL ontology engineering, we have developed a special-purpose RDF/OWL vocabulary, called *wOTTR*, with which OTTR templates and instances can be formulated. This has the benefit that we can leverage the existing stack of W3C languages and tools for developing, publishing, and maintaining templates. The wOTTR format supports

[2] https://gitlab.com/ottr/language/stOTTR/.

writing TRIPLE instances as regular RDF triples. This means that a pattern represented by an RDF graph or RDF/OWL ontology can easily be turned into an OTTR template by simply specifying the name of the template and its parameters with the wOTTR vocabulary. Furthermore, this means that we can make use of existing ontology editors and reasoners to construct and verify the soundness of templates. The wOTTR representation has been developed to closely resemble stOTTR. It uses RDF resources to represent parameters and arguments, and RDF lists (which have a convenient formatting in Turtle syntax) for lists of parameters and arguments. The vocabulary is published at ns.ottr.xyz. A more thorough presentation of the vocabulary is found in [13].

tabOTTR[3] is developed particularly for representing large sets of template instances in tabular formats such as spreadsheets, and is intended for domain expert use.

Generated Queries and Format Specifications. A template may not only be used as a macro, but also, inversely, as a query that retrieves all instances of the pattern and outputs the result in the tabular format of the template head. From a template we can generate queries from both its expanded and unexpanded body. The expanded version allows us to find instances of a pattern in "vanilla" RDF data, while the unexpanded version can be used to collect and transform (in the opposite direction than of expansion) a set of template instances into an instance of a larger template. The latter form is convenient for validating the proper usage of templates within a library, which we present in Sect. 4.

We are also experimenting with generating other specifications from a template, for instance XSD descriptions of template heads, and transformations of these formats, e.g., XSLT transformations. The purpose of supporting other formats is to allow for different data input formats and leverage existing tools for input verification and bulk transformation of instance data to expanded RDF, such as XSD validators and XSLT transformation engines.

Tools and Online Resources. Lutra, our Java implementation of the OTTR template macro expander, is available as open source with an LGPL licence at gitlab.com/ottr. It can read and write templates and instances of the formats described above and expand them into RDF graphs and OWL ontologies, while performing various quality checks such as parameter type checking and checking the resulting output for semantic consistency. Lutra is also deployed as a web application that will parse and display any OTTR template available online. The template may be expanded and converted into all the formats mentioned above, including SPARQL SELECT, CONSTRUCT and UPDATE queries, XSD format, and variants of expansions which include or exclude the head or body.

Also available, at library.ottr.xyz, is a "standard" set of ontology templates for expressing common RDF, RDFS, and OWL patterns as well as other example templates. These templates are conveniently presented in an online library that is linked to the online web application.

[3] https://gitlab.com/ottr/language/tabOTTR/.

```
NamedPizza(
    ?Name : 1 class,
    ?Country : ? individual,
    ?Toppings : + class)
  ::
    SubClassOf(?Name, :NamedPizza),
    SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
    SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
    ObjectUnionOf(_:b1, ?Toppings),
    x | SubObjectSomeValuesFrom(?Name, :hasTopping, ?Toppings) .

NamedPizza(:Margherita, :Italy, ⟨:Tomato, :Mozzarella⟩)
NamedPizza(:Grandiosa, none, ⟨:Tomato, :Jarlsberg, :Ham, :SweetPepper⟩)
```

(a) stOTTR serialisation and instances

```
<http://draft.ottr.xyz/pizza/NamedPizza> a ottr:Template ;
 ottr:hasParameter
  [ ottr:index 1 ; ottr:classVariable :pizza ] ,
  [ ottr:index 2 ; ottr:individualVariable :country;
       ottr:optional true ] ,
  [ ottr:index 3 ; ottr:listVariable (:toppings) ]  .
   ### body:
[] ottr:templateRef t-owl-axiom:SubClassOf ;
    ottr:withValues ( :pizza p:NamedPizza ) .
[] ottr:templateRef t-owl-axiom:SubObjectHasValue ;
    ottr:withValues ( :pizza p:hasCountryOfOrigin :country ) .
[] ottr:templateRef t-owl-axiom:SubObjectAllValuesFrom ;
    ottr:withValues ( :pizza p:hasTopping _:alltoppings ) .
[] ottr:templateRef t-owl-rstr:ObjectUnionOf ;
    ottr:withValues ( _:alltoppings (:toppings) ) .
[] ottr:templateRef t-owl-axiom:SubObjectSomeValuesFrom ;
    ottr:hasArgument [ ottr:index 1; ottr:value :pizza ] ,
        [ ottr:index 2; ottr:value p:hasTopping ] ,
        [ ottr:index 3; ottr:eachValue (:toppings) ] .
```

(b) wOTTR serialisation

```
:pizza  rdfs:subClassOf  p:NamedPizza ,
     [ a                    owl:Restriction ;
       owl:onProperty       p:hasTopping ;
       owl:someValuesFrom   :toppings ] ,
     [ a                   owl:Restriction ;
       owl:onProperty      p:hasTopping
       owl:allValuesFrom    [
           a             owl:Class ;
           owl:unionOf   ( :toppings ) ] ] ,
     [ a                owl:Restriction ;
       owl:onProperty   p:hasCountryOfOrigin ;
       owl:hasValue     :country ] .
p:hasTopping           a  owl:ObjectProperty .
p:hasCountryOfOrigin   a  owl:ObjectProperty .
:toppings              a  owl:Class .
```

(c) Expanded RDF graph

```
SELECT  *
  { ?param1    rdfs:subClassOf       p:NamedPizza ,
      [ owl:onProperty       p:hasTopping ;
        owl:someValuesFrom   param3item ;
        rdf:type             owl:Restriction ] ,
      [ owl:allValuesFrom    [ owl:unionOf  ?param3 ;
                               rdf:type     owl:Class ] ;
        owl:onProperty       p:hasTopping ;
        rdf:type             owl:Restriction ]
    OPTIONAL {
      ?param1   rdfs:subClassOf  [
        owl:hasValue         ?param2 ;
        owl:onProperty       p:hasCountryOfOrigin ;
        rdf:type             owl:Restriction ]
    }
      ?param3   (rdf:rest)*/rdf:first ?param3item
  }
```

(d) SPARQL SELECT query

?param1	?param3item	?param2
p:Margherita	p:MozzarellaTopping	
p:Margherita	p:TomatoTopping	
p:Mushroom	p:MozzarellaTopping	
p:Mushroom	p:MushroomTopping	
p:Mushroom	p:TomatoTopping	
p:Napoletana	p:AnchoviesTopping	p:Italy
p:Napoletana	p:CaperTopping	p:Italy
p:Napoletana	p:MozzarellaTopping	p:Italy
p:Napoletana	p:OliveTopping	p:Italy
p:Napoletana	p:TomatoTopping	p:Italy

(e) Excerpt query results

```
#OTTR  prefix
p  http://www.co-ode.org/ontologies/pizza/pizza.owl#
#OTTR  end
#OTTR  template  http://draft.ottr.xyz/pizza/NamedPizza
pizza          country  toppings
1              2        3
iri            iri      iri+
p:Margherita   p:Italy  p:Tomato|p:Mozzarella
p:Grandiosa             p:Tomato|p:Jarlsberg|p:Ham|p:Pepper
#OTTR  end
```

(f) tabOTTR instance serialisation

Fig. 2. NamedPizza template and example instances in different serialisations

Example 4. Figure 2 contains different representations of the NamedPizza template. Figure 2(b) contains the published version of the template, available at its IRI address: http://draft.ottr.xyz/pizza/NamedPizza. Figure 2(c) contains the expansion of the template body. Figure 2(d) displays the generated SPARQL query that retrieves instances of the pizza pattern; an excerpt of the results applying the query to the Pizza ontology is given in Fig. 2(e). Figure 2(f) contains a tabOTTR representation of the two instances seen in Fig. 2(a). We encourage the reader to visit the rendering of the template by the web application at osl.ottr.xyz/info/?tpl=http://draft.ottr.xyz/pizza/NamedPizza and explore the

various presentations and formats displayed. An example-driven walk-through of the features of Lutra can be found at ottr.xyz/event/2018-10-08-iswc/.

4 Maintenance and Optimisation of OTTR Template Libraries

In this section, we present an initial list and analysis of some of the more central relations between OTTR templates, and discuss their use in template library optimisation. We focus in particular on removing redundancy within a library, where we distinguish two different types of redundancy: a lack of reuse of existing templates, as well as recurring patterns not captured by templates within the library. We present an efficient and automated technique for detecting such redundancies within an OTTR template library.

4.1 OTTR Template Relations

Optimisation and maintenance of OTTR template libraries is made possible by its solid formal foundation. OTTR syntax makes it possible to formally define relations between OTTR templates which can tangibly benefit the optimisation of a template library. Naturally, there are any number of ways templates can be "related" to one another, and the "optimal" size and shape of a template library is likely to be highly domain and ontology-specific. As such, we do not aspire to a best-practice approach to optimising a template library. Instead, we illustrate the point by defining a few central template relations and demonstrating their usefulness for library optimisation and maintenance, independently of the heuristics used. Here, we limit ourselves to template relations defined syntactically in terms of instances, and do not consider, e.g., those defined in terms of semantic relationships between full expansions of templates. We consider the following template relations:

directly depends (DD) S *directly depends on* T if S's body has an instance of T.

depends (D) *depends* is the transitive closure of *directly depends*.

dependency-overlaps (DO) S *dependency-overlaps* T if there exists a template upon which both S and T directly depend.

overlaps (O) S *overlaps* T if there exist template instances $i_{\mathrm{S}}, i_{\mathrm{T}}$ in $\mathsf{body}(\mathrm{S})$ and $\mathsf{body}(\mathrm{T})$ and substitutions ρ and η of the parameters of S and T resp. such that $\rho(i_{\mathrm{S}}) = i_{\mathrm{T}}$ and $\eta(i_{\mathrm{T}}) = i_{\mathrm{S}}$.

contains (C) S *contains* T if there exists a substitution ρ of the parameters of T such that $\rho(\mathsf{body}(\mathrm{T})) \subseteq \mathsf{body}(\mathrm{S})$.

equals (E) S *is equal to* T if S contains T and vice versa.

Each of the listed relations is, in a sense, a specialisation of the previous one (except for *DO*, which is a specialisation of *DD* as opposed to *D*). For instance, *DO* imposes no restrictions on the instance arguments, whereas *O* intuitively requires parameters to occur in compatible positions of i_{S} and i_{T}.

NamedPizza(?Name : 1 class, ?Country : ? individual, ?Toppings : + class) (cf. Fig. 2(a)) .

AnnotatedPizza(?Name : 1 class, ?Label : + literal, ?PrefLabel : ? literal, ?Definition : ? literal)
:: SubClassOf(?Name, :Pizza),
× | (?Name, rdfs:label, ?Label), (?Name, skos:prefLabel, ?PrefLabel), (?Name, skos:definition, ?Definition) . (5)

Burger(?Name : 1 class, ?Condiments : + class, ?Label : + literal, ?PrefLabel : ? literal, ?Definition : ? literal)
:: SubClassOf(?Name, :Burger),
× | SubObjectSomeValuesFrom(?Name, :hasCondiment, ?Condiments),
SubClassOf(?Name, _:b2), ObjectAllValuesFrom(_:b2, :hasCondiment, _:b3), (6)
ObjectUnionOf(_:b3, ?Condiments), (7)
× | (?Name, rdfs:label, ?Label), (?Name, skos:prefLabel, ?PrefLabel), (?Name, skos:definition, ?Definition) . (8)

BurgerMeal(?Name : 1 class, ?Sides : + class)
:: SubObjectSomeValuesFrom(?Name, :hasMain, :Burger),
SubObjectAllValuesFrom(?Name, :hasSide, _:b4), ObjectUnionOf(_:b4, ?Sides) . (9)

(a) OTTR template library with redundancies and lack of re-use

NamedPizza(?Name : 1 class, ?Country : ? individual, ?Toppings : + class)
:: SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
NamedFood(?Name, :NamedPizza, ?Toppings, :hasTopping) . (⋆)

AnnotatedPizza(?Name : 1 class, ?Label : + literal, ?PrefLabel : ? literal, ?Definition : ? literal)
:: SubClassOf(?Name, :Pizza),
Annotation(?Name, ?Label, ?PrefLabel, ?Definition) . (⋆)

Burger(?Name : 1 class, ?Condiments : + class, ?Label : + literal, ?PrefLabel : ? literal, ?Definition : ? literal)
:: NamedFood(?Name, :Burger, ?Condiments, :hasCondiment), (⋆)
Annotation(?Name, ?Label, ?PrefLabel, ?Definition) . (⋆)

BurgerMeal(?Name : 1 class, ?Sides : + class)
:: SubObjectSomeValuesFrom(?Name, :hasMain, :Burger),
SubObjectAllValuesFromUnion(?Name, :hasSide, ?Sides) . (⋆)

New OTTR templates representing previously uncaptured patterns:

NamedFood(?Name : 1 class, ?Category : 1 class, ?Extras : + class, ?hasExtra : 1 objectProperty) (10)
:: SubClassOf(?Name, ?Category),
× | SubObjectSomeValuesFrom(?Name, ?hasExtra, ?Extras),
SubObjectAllValuesFromUnion(?Name, ?hasExtra, ?Extras)

Annotation(?Name : 1 class, ?Label : + literal, ?PrefLabel : ? literal, ?Definition : ? literal) (11)
:: × | (?Name, rdfs:label, ?Label), (?Name, skos:prefLabel, ?PrefLabel), (?Name, skos:definition, ?Definition) .

SubObjectAllValuesFromUnion(?x : 1 class, ?Property : 1 objectProperty.?RangeList : + class) (12)
:: SubObjectAllValuesFrom(?x, ?Property, _:b1), ObjectUnionOf(_:b1, ?RangeList) .

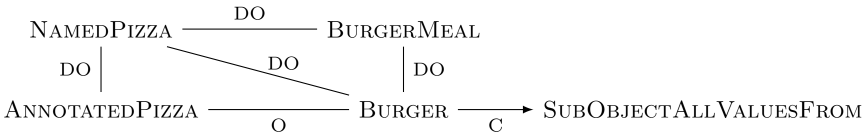
(b) A refactored version of the templates in Fig. 3(a). The refactored templates from Fig. 3(a) are listed first, where a star (⋆) indicates a dependency to a new template, found at the bottom.

Fig. 3. OTTR template library before and after redundancy removal

Example 5. Consider the template library given in Fig. 3(a). All but the BurgerMeal template contain an instance of SubClassOf, hence all pairs of templates except for (AnnotatedPizza, BurgerMeal) have a dependency-overlap. Closer inspection reveals that Burger contains SubObjectAllValuesFrom (4, Fig. 1), due to the instances

– SubClassOf(?Name, _:b2)
– ObjectAllValuesFrom(_:b2, :hasCondiment, _:b3)

in Burger (6). (Numbers refer to numbered lines in the figures.) Finally, AnnotatedPizza and Burger overlap, since they both directly depend on the same Triple templates (5) and (8). These relationships are depicted in the graph below (dependency relationships omitted for the sake of legibility). Directed/undirected edges depict nonsymmetric/symmetric relations, respectively.



We wish to discuss these relations in the context of *redundancy removal* within an OTTR template library. More specifically, we discuss two types of redundancy:

Lack of reuse is a redundancy where a template S has a contains relationship to another template T, instead of a dependency relationship to T. That is, S duplicates the pattern represented by T, rather than instantiating T. This can be removed by replacing the offending portion of body(S) with a suitable instance of T. A first approach to determining such a lack of reuse makes use of the fact that templates can be used as queries: template S contains T iff T as a query over S yields answers.

Uncaptured pattern is a redundancy where a pattern of template instances is used by multiple templates, but this pattern is not represented by a template. In order to find uncaptured patterns one must analyse in what manner multiple templates depend on the same set of templates. If multiple templates *overlap* as defined above, this is a good candidate for an uncaptured pattern. However, an overlap does not necessarily need to occur for an uncaptured pattern to be present: as demonstrated in the following example, a dependency-overlap can describe an uncaptured pattern that is relevant for the template library.

Example 6. Continuing with our previous example of the library in Fig. 3(a), we find that it contains both an instance of lack of reuse and multiple instances of uncaptured patterns. The containment of SubObjectAllValuesFrom in Burger indicates a lack of reuse, and the overlap of Burger and AnnotatedPizza

is an uncaptured pattern which we refactor into the template Annotation (11). By repairing the lack of reuse in Burger (6) with an instance of SubObjectAllValuesFrom, there are two dependency-overlaps that represent uncaptured patterns: the instances (6,7)(9), which are refactored into a new template SubObjectAllValuesFromUnion (12), and the dependency-overlap between Burger and NamedPizza, which is described by the NamedFood template (10). These new templates as well as the updated template definitions for the pre-existing ones are given in Fig. 3(b).

4.2 Efficient Redundancy Detection

Naive methods for improving a template library using the relations as described in the previous section quickly become infeasible for large knowledge bases, as they require expensive testing of unification of all template bodies. We have developed an efficient method for finding lack of reuse and uncaptured patterns, which over-approximates the results of unification. The method uses the notion of a dependency pair, which intuitively captures repeated use of templates without considering parameters: a *dependency pair* $\langle I, T \rangle$ is a pair of a multiset of templates I and a set of templates T, such that T is the set of all templates that directly depend on all templates in I, and have at least as many directly depends relationships to each template in I as they occur in I. The idea is that I represents a pattern used by all the templates in T. In order to also detect patterns containing different Triple instances, we will in this section treat a Triple instance (s, p, o) as a template instance of the form $p(s, o)$ and thus treat p as a template. Note that for a set of dependency pairs generated from a template library, the first element in the pair, i.e., the I, is unique for the set, while the T is generally not unique.

Example 7. Three examples of dependency pairs from the library in Fig. 3(a) are

1. $\langle\{$SubClassOf, SubClassOf, rdfs:label$\}, \{$Burger$\}\rangle$
2. $\langle\{$SubClassOf, ObjectAllValuesFrom$\}, \{$SubObjectAllValuesFrom, Burger$\}\rangle$
3. $\langle\{$skos:definition, rdfs:label, skos:prefLabel$\}, \{$AnnotatedPizza, Burger$\}\rangle$

The first pair indicates that Burger is the only template that directly depends on two occurrences of SubClassOf and one occurrence of rdfs:label. Note that Burger directly depends on other templates too, and these will give rise to other dependency pairs. However there is no other template than Burger that directly depends on this multiset of templates. The second example shows that SubObjectAllValuesFrom and Burger directly depend on the templates SubClassOf and ObjectAllValuesFrom.

One can compute all dependency pairs by starting with the set of dependency pairs of the form $\langle\{i : n\}, T\rangle$ where all templates in T have at least n instances of i, and then compute all possible *merges*, where a merge between two clusters

$\langle I_1, T_1 \rangle$ and $\langle I_2, T_2 \rangle$ is $\langle I_1 \cup I_2, T_1 \cap T_2 \rangle$. We have implemented this algorithm with optimisations that ensure we compute each dependency pair only once.

The set of dependency pairs for a library contains all potential lack of reuse and uncaptured patterns in a library. However, note that in the dependency pairs where either I or T has only one element, the dependency pair does not represent a commonly used pattern: If I has only one element then it does not represent a redundant pattern. If T has only one element then the pattern occurs only once. If on the other hand both sets contain two or more elements then the dependency pair might represent a useful pattern to be represented as a template, and we call these *candidate pairs*.

For a candidate pair, there are three cases to consider: 1. the set of instances does not form a pattern that can be captured by a template, as the usage of the set of instances does not unify; 2. the pattern is already captured by a template, in which case we have found an instance of lack of reuse; otherwise 3. we have found one or more candidates (one for each non-unifiable usage of the instances of I) for new templates. The two first cases can be identified automatically, but the third needs user interaction to assess. First, a user should verify for each of the new templates that it is a meaningful pattern with respect to the domain; second, if the template is meaningful, a user must give the new template an appropriate name.

To remove the redundancy a candidate pair $\langle I, T \rangle$ represents, we can perform the following procedure for each template $t \in T$ and $T' = T \setminus \{t\}$. First we check for lack of reuse of t: this may only be the case if t's body has the same number of instances as there are templates in I. We verify the lack of reuse by checking if $t' \in T'$ contains t; this is done by verifying that t used as a query over t''s body yields an answer. If there is no lack of reuse, we can represent the instances of I as they are instantiated in t, as the body of a new template where all arguments are made into parameters. Again, we need to verify that the new template is contained in other templates in T' before we can refactor, and before any refactoring is carried out, a user should always assess the results.

Example 8. Applying the method for finding candidates to the library in Fig. 3(a), gives 19 candidate pairs, two of which are the 2nd and 3rd candidate pair of Example 7. The 1st dependency pair of Example 7 is not a candidate pair since the size of one of its elements ({Burger}) is one.

By using the process of removing redundancies as described above, we will find that for the 2nd candidate pair of Example 7 we have a lack of reuse of SubObjectAllValuesFrom in Burger, as discussed in previous examples. The two instances of SubClassOf and ObjectAllValuesFrom in Burger (see Example 5) can be therefore be replaced with the single instance: SubObjectSomeValuesFrom(?Name, :hasCondiment, _:b3).

From the 3rd candidate pair in Example 7 there is no lack of reuse, but we can represent the pattern as the following template:

<name>(?x1, ?x2, ?x3, ?x4)
:: (?x1, rdfs:label, ?x2), (?x1, skos:prefLabel, ?x3), (?x1, skos:definition, ?x4).

The template and parameters should be given suitable names and parameters given a type, as exemplified by the Annotation template (11) found in Fig. 3(b). The procedure of identifying dependency pairs and lack of reuse is implemented and demonstrated in the online walk-through at ottr.xyz/event/2018-10-08-iswc/.

For large knowledge bases, the set of candidate pairs might be very large, as it grows exponentially in the number of template instances in the worst case. This means that manually assessing all candidate pairs is not feasible, and smaller subsets of candidates must be automatically suggested. We have yet to develop proper heuristics for suggesting good candidates, but the cases with the most common patterns (the candidates with largest T-sets), the largest patterns (the candidates with the largest I-sets), or large patterns that occur often could be likely sources for patterns to refactor. The latter of the three can be determined by maximising a weight-function, for instance of the form $f(\langle I, T\rangle) = w_1|I| + w_2|T|$. However, these weights might differ from use-case to use-case. Another approach for reducing the total number of candidates to a manageable size, is to let a user group some or all of the templates according to subdomain, and then only present candidates with instances fully contained in a single group. The idea behind such a restriction is that it seems likely that a pattern is contained within a subdomain. We give an example of these techniques in the following section.

5 Use Case Evaluation

In this section we outline an evaluation of OTTR templates in a real-world setting at the engineering company Aibel, and demonstrate in particular our process of finding and removing redundancies over a large, generated template library.

Aibel is a global engineering, procurement, and construction (EPC) service company based in Norway best known for its contracts for building and maintaining large offshore platforms for the oil and gas industry. When designing an offshore platform, the tasks of matching customer needs with partly overlapping standards and requirements as well as finding suitable products to match design specifications are highly non-trivial and laborious. This is made difficult by the fact that the source data is usually available only as semi-structured documents that require experience and detailed competence to interpret and assess. Aibel has taken significant steps to automate these tasks by leveraging reasoning and queries over their *Material Master Data* (MMD) ontology. It integrates this information in a modular large-scale ontology of ∼200 modules and ∼80,000 classes and allows Aibel to perform requirements analysis and matching with greater detail and precision and less effort than with their legacy systems. Since the MMD ontology is considered by Aibel as a highly valuable resource that gives them a competitive advantage, it is not publicly available.

The MMD ontology is produced from 705 spreadsheets prepared by ontology experts and populated by subject matter experts with limited knowledge of

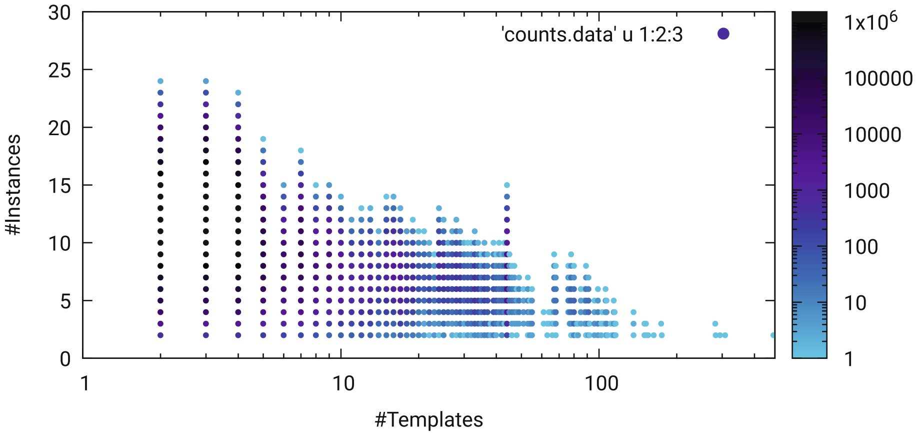


Fig. 4. A scatter plot of the sizes of the two sets for all candidate pairs from Aibel use case. The colour shade denotes the logarithm of the number of candidates at each point.

modelling and semantic technologies. The column headers of the spreadsheets specify how the data is to be converted into an ontology, and the translation is performed by a custom-built pipeline of custom transformations, relational databases, and SPARQL CONSTRUCT transformations. The growing size and complexity of the system, the simple structure of the spreadsheets and lack of common modelling patterns make it hard to keep an overview of the information content of the spreadsheets and enforce consistent modelling across spreadsheets. The absence of overarching patterns also represents a barrier for Aibel's wish to extend the ontology to cover new engineering disciplines, as there are no patterns that are readily available for reuse.

The aim of our evaluation is to test whether OTTR templates and the tools presented in this paper can replace Aibel's current in-house built system and improve the construction and maintenance of the MMD ontology. By exploiting the simple structure of the spreadsheets we automatically generated OTTR templates: one for each spreadsheet (705 templates), one for each unique column header across spreadsheets (476 templates), and one for each axiom pattern used, e.g., existential restriction axiom (4 templates).

To analyse the large template library, we applied the algorithm for finding candidate pairs described in Sect. 4.2, giving a total of 54,795,593 candidate pairs. The scatter plot in Fig. 4 shows the distribution of sizes for the two sets; the largest number of instances and templates for a given candidate is 24 and 474, respectively. The large number of candidates makes it impossible to manually find potential templates, thus we employed the semi-automatic method described in the previous section to suggest possible improvements to the library. In order to demonstrate the process, we selected the candidates that contain a specific template, the template modelling a particular type of *pipe elbows* from

the AMSE B16.9 standard, which is an often-used example from the MMD ontology. This template occurs in a total of 12,273 candidates. To reduce the number of candidates further, we removed candidates with instances of a generic character, such as `rdfs:label`, to end up with candidates with domain-specific templates. By using a weight function, we selected the candidate with the largest set of templates and at least 6 instances. From this candidate, with 33 templates and 7 instances, we obtained a template suggestion that we were able to verify is contained by all of the 33 templates in T, by using the template as a query over the templates in T. We added this new template to the library and refactored it into all the 33 templates using its pattern.

Fixing this single redundancy reduced the total number of candidates by over 1.8 million. This great reduction in candidates comes from the fact that fixing a redundancy represented by a candidate C can also fix the redundancies of candidates having a pattern that is contained in, contains, or overlaps C's pattern. This indicates that, despite a very large number of candidates, small fixes can dramatically reduce the overall redundancy. Furthermore, by automatically refactoring all lack of reuse in the entire library, the number of candidates decreases to under 3 million. The average number of instances per template went from 5.6 down to 2.7 after this refactoring. In addition to the redundancies fixed above, we were also able to detect equal templates (pairs of templates both having a lack of reuse of the other). Out of the 931 templates we analysed, only 564 were unique. Thus, we could remove a total of 367 redundant templates from the library. Note that all of the improvements made above should be reviewed by a user, as discussed in Sect. 4.2, to ensure that the new template hierarchy properly represents the domain.

The use case evaluation indicates that OTTR templates and tools can replace Aibel's custom built approach for transforming spreadsheets into ontologies. Indeed, OTTR greatly exceeds the expressivity of Aibel's spreadsheet structure and provides additional formal structure that can be used to analyse and improve the modelling patterns used to capture domain knowledge. As future evaluation, we want to work with Aibel's domain experts in order to identify promising heuristics for finding the best shared patterns. We believe that these new patterns and user requirements from Aibel may foster new ideas for added expressivity and functionality of OTTR languages and tools. Furthermore, we want to evaluate whether we can replace Aibel's hand-crafted queries with queries generated from templates. This would avoid the additional cost of maintaining a large query library, while benefiting from already existing templates and OTTR's compositional nature and tools for building and analysing the generated queries.

6 Related Work

Modularised ontologies, as well as the use and description of ontology design patterns, have attracted significant interest in recent years, as demonstrated by the multitude of languages and frameworks that have emerged. However, a hurdle for the practical large-scale use of ontology design patterns is the lack of

a tool supported methodology; see [4] for a discussion of some of the challenges facing ontology design patterns. In this section we present selected work related to our approach that we believe represents the current state of the art.

An early account of the features, benefits and possible use-cases for a macro language for OWL can be found in [14].

The practical and theoretical aspects of OTTR templates were first introduced in [13] and [2]. This paper presents a more mature and usable framework, including formalisation and use of template relations, real-world evaluation, added expressivity in the form of optional parameters and expansion modes, and new serialisation formats.

GDOL [9] is an extension of the Distributed Ontology, Modelling, and Specification Language (DOL) that supports a parametrisation mechanism for ontologies. It is a metalanguage for combining theories from a wide range of logics under one formalism while supporting pattern definition, instantiation, and nesting. Thus it provides a broad formalism for defining ontology templates along similar lines as OTTR. To our knowledge, GDOL has yet to investigate issues such as dependencies and relationships between patterns, optional parameters, and pattern-as-query (the latter being listed as future work). A protege plugin for GDOL is in the works and DOL is supported by Ontohub (an online ontology and specification repository) and Hets (parsing and inference backend of DOL).

Ontology templates as defined in [1] are parameterised ontologies in $\mathcal{ALC}$ description logic. Only classes are parameterised, and parameter substitutions are restricted to class names. This is quite similar to our approach, yet it is not adapted to the semantic web, and nested templates and patterns-as-queries are not considered. Furthermore, it appears this project has been abandoned, as the developed software is no longer available.

OPPL [6] was originally developed as a language for manipulating OWL ontologies. Thus it supports functions for adding and removing patterns of OWL axioms to/from an ontology. It relies heavily on its foundations in OWL-DL and as such can only be used in the context of OWL ontologies. Despite this, the syntax of OPPL is distinct from that of RDF, thus requiring separate tools for viewing and editing such patterns, though a Protégé plugin does exist, in addition to a tool called *Populous* [7] which allows OPPL patterns to be instantiated via spreadsheets. By allowing patterns to return a single element (e.g., a class) OPPL supports a rather restricted form of pattern nesting as compared to OTTR.

Tawny OWL [10] introduces a Manchester-like syntax for writing ontology axioms from within the programming language Clojure, and allows abstractions and extensions to be written as normal Clojure code alongside the ontology. Thus the process of constructing an ontology is transformed into a form of programming, where existing tools for program development, such as versioning, testing frameworks, etc. can be used. The main difference from our approach is that Tawny OWL targets programmers and therefore tries to reuse as much of the standards and tools used in normal Clojure development, whereas we aim to reuse semantic technology standards and tools.

OPLa [5] is a proposal for a language to represent the relationships between ontologies, modules, patterns, and their respective parts. They introduce the OPLa ontology which describes these relationships with the help of OWL annotation properties. This approach does not, however, attempt to mitigate issues arising with the *use* of patterns, but focuses more on the description of patterns, than on practical use.

There are other tools and languages such as XDP [3], built on top of WebProtégé as a convenient tool for instantiating ODPs, the M^2 mapping language [11] that allows spreadsheet references to be used in ontology axiom patterns, and RDF shape languages, such as SHACL [8], that may be used to describe and validate patterns. Although these have similarities with OTTR, we consider these more specialised tools and languages, where for example analysis of patterns is beyond their scope.

7 Conclusion and Future Work

This paper presents OTTR, a language with supporting tools for representing, using and analysing ontology modelling patterns. OTTR has a firm theoretical and technological base that allows existing methods, languages and tools to be leveraged to obtain a powerful, yet practical instrument for ontology construction, use and maintenance.

For future work, the natural next step with respect to template library optimisation is to continue and expand the analysis of Sect. 4, both for existing and new template relations. In particular, it is natural to compare templates both syntactically using their full expansion and in terms of their semantic relationship. The latter would allow us, e.g., to answer questions about consistency and whether a given library is capable of describing a certain knowledge pattern. We also want to develop specialised editors for OTTR templates, such as a plugin for Protégé, and extend support for more input formats, such as accessing data from relational databases.

Acknowledgements. We would like to thank Per Øyvind Øverli from Aibel, and Christian M. Hansen from Acando for their help with the evaluation of OTTR. The second and fourth author were supported by Norwegian Research Council grant no. 230525.

References

1. Blasko, M., Kremen, P., Kouba, Z.: Ontology evolution using ontology templates. Open J. Semant. Web (OJSW) **2**, 15–28 (2015)
2. Forssell, H., et al.: Reasonable macros for ontology construction and maintenance. In: Proceedings of the 30th International Workshop on Description Logics (2017)
3. Hammar, K.: Ontology design patterns in WebProtege. In: Proceedings of the ISWC 2015 Posters and Demonstrations Track (2015)

4. Hammar, K., et al.: Collected research questions concerning ontology design patterns. In: Hitzler, P., et al. (eds.) Ontology Engineering with Ontology Design Patterns, pp. 189–198. IOS Press (2016)
5. Hitzler, P., et al.: Towards a simple but useful ontology design pattern representation language. In: Proceedings of the 8th Workshop on Ontology Design and Patterns (2017)
6. Iannone, L., Rector, A., Stevens, R.: Embedding knowledge patterns into OWL. In: Aroyo, L., et al. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 218–232. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02121-3_19
7. Jupp, S.: Populous: a tool for building OWL ontologies from templates. BMC Bioinform. **13**(S–1), S5 (2012)
8. Knublauch, H., Kontokostas, D.: Shapes constraint language (SHACL) (2017). W3C Recommendation
9. Krieg-Brückner, B., Mossakowski, T.: Generic ontologies and generic ontology design patterns. In: Proceedings of the 8th Workshop on Ontology Design and Patterns (2017)
10. Lord, P.: The semantic web takes wing: programming ontologies with Tawny-OWL. In: OWLED (2013)
11. O'Connor, M.J., Halaschek-Wiener, C., Musen, M.A.: M2: a language for mapping spreadsheets to OWL. In: OWLED (2010)
12. Ogden, C.K., Richards, I.A.: The Meaning of Meaning. Harvest Book, San Diego (1946)
13. Skjæveland, M.G., et al.: Pattern-based ontology design and instantiation with reasonable ontology templates. In: Proceedings of the 8th Workshop on Ontology Design and Patterns (2017)
14. Vrandečić, D.: Explicit knowledge engineering patterns with macros. In: Proceedings of the Ontology Patterns for the Semantic Web Workshop at the ISWC 2005 (2005)