The Reasonable Ontology Templates Framework

Martin Georg Skjæveland
Department of Informatics, University of Oslo, Norway

Leif Harald Karlsen
Department of Informatics, University of Oslo, Norway

Abstract

Reasonable Ontology Templates (OTTR) is a templating language for representing and instantiating patterns. It is based on simple and generic, but powerful, mechanisms such as recursive macro expansion, term substitution and type systems, and is designed particularly for building and maintaining RDF knowledge graphs and OWL ontologies.

In this resource paper, we present the formal specifications that define the OTTR framework. This includes the fundamentals of the OTTR language and the adaptions to make it fit with standard semantic web languages, and two serialization formats developed for semantic web practitioners. We also present the OTTR framework's support for documenting, publishing and managing template libraries, and for tools for practical bulk instantiation of templates from tabular data and queryable data sources. The functionality of the OTTR framework is available for use through Lutra, an open-source reference implementation, and other independent implementations. We report on the use and impact of OTTR by presenting selected industrial use cases. Finally, we reflect on some design considerations of the language and framework and present ideas for future work.

2012 ACM Subject Classification Computing methodologies → Ontology engineering; Information systems → Data management systems; Computing methodologies → Modeling methodologies

Keywords and phrases Ontology engineering, Ontology design patterns, Template mechanism, Macros

Digital Object Identifier 10.4230/TGDK.2.2.5

Category Resource Paper

Supplementary Material The homepage for the project is `ottr.xyz`. Versioned releases of Lutra and OTTR specifications are published at Zenodo. The source code for Lutra is published on GitLab under the GNU Lesser General Public License v2.1 license. Maven artefacts for Lutra are published at Sonatype.
InteractiveResource (Website): `https://ottr.xyz/`
Software: `https://gitlab.com/ottr`
Software: `https://zenodo.org/communities/ottr/`
Software (Maven artefacts): `https://central.sonatype.com/namespace/xyz.ottr.lutra`

Funding We acknowledge financial support from The Research Council of Norway through SIRIUS, Centre for Scalable Data Access (237898).
Martin Georg Skjæveland: The author acknowledges funding from the EU projects RE4DY (101058384) and Tec4MaaSEs (101138517).

Acknowledgements We wish to thank all past members of the OTTR project: Chris Kindermann, Daniel Lupp, Evgenij Thorstensen, Henrik Forssell, Laura Slaughter, Oliver Stahl; its associated master students: Erik Snilsberg, Lars Ivar Bull Larssen, Magnus Wiik Eckhoff, Marlen Jarholt, Preben Zahl, Shanshan Qu; part-time programmers: Fariha Hossain, Humza Ahmad, Vinicius Graciolli, Yiyao Chen; and SHS for their contributions to the OTTR framework. We are also grateful to the users of the OTTR framework for all their feedback and support.

Received 2024-07-01 **Accepted** 2024-11-07 **Published** 2024-12-18

Editors Aidan Hogan, Ian Horrocks, Andreas Hotho, Lalana Kagal, and Uli Sattler

Special Issue Resources for Graph Data and Knowledge

1 Introduction

Abstraction is a fundamental concept in computer science, particularly in software engineering and information modelling. In these disciplines, abstraction entails identifying and describing the relevant entities and structures for the problem at hand at a suitable level of detail. Done correctly, abstraction helps to hide unnecessary detail and presents the essence of the content that is described to the effect that the content is clearly conveyed and easily understood, and can hence be more efficiently processed by operations acting on the representation.

Figure 1 displays a simplified comparison of different abstraction levels with two code snippets that both write “Hello world!” to screen. The first snippet is written in x86 Linux assembly language[1] and the second is written in the high-level programming language Python. There is a striking difference between the two snippets: the Python code is succinct, easy to read and write and understand, while the assembly code is far more verbose as it must orchestrate a series of low-level resources and steps, such as memory locations and sizes, file descriptors and interrupt handlers, in order to solve the task at hand.[2] For most users, the high level of abstraction provided by the Python snippet is appropriate when all one wants is to write messages to screen – all other details are hidden. This code is safe and robust for this use case; its succinctness makes it difficult to use the code in the wrong way. The assembly code is arguably incomprehensible for most users and appears inefficient to use and manage if the task is just to write messages to screen. However, for some expert users or use cases the level of detail and control offered by assembly languages to interact more directly with hardware is exactly what is needed.

Most modern programming languages, like Python, offer mechanisms for different kinds of user-defined abstractions, such as functions, classes, interfaces, and modules, and it is common to package and distribute a set of such abstractions in an application programming interface (API). A well-designed API offers a suitable abstraction level using terminology that is familiar and natural for its intended users and hides the details of underlying lower-level APIs or systems. Understanding, managing and designing APIs is a central part of modern software engineering.

The Resource Description Framework (RDF) [27] and the Web Ontology Language (OWL) [40] are the standard languages for representing knowledge graphs and ontologies. A challenge for the wider adoption of semantic web languages, and ontology languages in particular, is its inherent complexity, a steep learning curve and the lack of developer-, and end user-friendly ways to interact with their artefacts. Thus, interfaces and simplifications for eliciting the content of ontologies are identified as opportunities for future research [56]. In this regard, it is worth noting that these languages offer very limited options for user-defined abstractions and provide no means to represent modelling patterns or templates that can be instantiated in a precise and deterministic manner, and that can hide details that appear unnecessary and complex to users.

As a case in point, consider the Protégé Pizza Ontology Tutorial[3] which models the domain of pizzas for the purpose of demonstrating and teaching features of OWL and Protégé [37]. This OWL ontology contains 22 types of pizza that are modelled following the same pattern; the description logic axioms that represent the Margherita pizza are listed in Figure 2 (the numbers that follow in parentheses refer to axioms in the figure): every pizza is represented as a subclass of `NamedPizza` (1), some pizzas have a country of origin (2), and toppings are expressed by stating that they are both required (3, 4) and permissible (5) for the pizza. Figure 2 also contains two different standard serialization formats for OWL: Manchester syntax [17] and RDF Turtle [41, 2].

[1] The code snippet is taken from `https://gist.github.com/pablocorbalann/f9d39a80e30b8d8230a9760048d0e575`.
[2] The interested reader can find more information about the assembly code in the following article: `https://pablocorbalann.medium.com/`.
[3] `http://protege.stanford.edu/ontologies/pizza/pizza.owl`

x86 Linux assembly language:

```
section        .text                ; declare the .text section
global         _start               ; has to be declared for the linker (ld)
_start:                             ; entry point for _start
    mov edx, len                    ; "invoke" the len of the message
    mov ecx, msg                    ; "invoke" the message itself

    mov ebx, 1                      ; set the file descriptor (fd) to stdout

    mov eax, 4                      ; system call for "write"
    int 0x80                        ; call the kernel

    mov eax, 1                      ; system call for "exit"
    int 0x80                        ; call the kernel

section        .data                ; here you declare the data
    msg        db "Hello world!"    ; the actual message to use
    len        equ $ -msg           ; get the size of the message
```

Python:

```
print("Hello world!")
```

Figure 1 “Hello world!” printed to screen in x86 Linux assembly language, and in Python.

The Manchester syntax lies close to the description logics representation, while the RDF Turtle serialization is more verbose as all statements are on the form of triples, and must use blank nodes and resources such as `owl:Restriction` in order to represent the same information.

The case demonstrates two points: The first point is that RDF and OWL, the standard knowledge representation languages for the web, appear as expert languages that operate on a too low level of abstraction for the task of representing ordinary compound modelling patterns, such as pizzas, in a succinct and readable manner. The representations arguably expose too many details in the form of logical constructs and language peculiarities in order to be easy to read and understand for non-experts. The second point is that the lack of abstraction mechanisms for RDF and OWL forces all statements to be on the form of RDF triples and OWL axioms and limited to the constructs defined by these standards, such as `some`/`owl:someValuesFrom`. This makes the representations repetitive and verbose. In the code samples this is shown with the repetition of the existentially quantified axiom and the fact that, e.g., a “macro” symbol [58] that allows to express the required and permissible pizza toppings in a single statement is not possible to declare. For the full Pizza Ontology, repetition is also visible with the 22 pizzas using the same pizza modelling patterns by replicating all the axiom schemata. As a result, the representation of only the pizzas, according to the pattern in Figure 2, comprises in total 198 OWL axioms and 1106 RDF triples.

The overall effect is that the current standard representation formats for knowledge graphs and ontology will often appear too far removed from most users’ understanding and conceptualization of the domain, and is therefore difficult to understand and use. Also, the fact that there is no explicit representation of a pattern and its instances makes it difficult to identify that any pattern is followed, which again makes it difficult to ensure consistent modelling. Furthermore, it complicates consistent and efficient updates of the pattern instances as they are spread across multiple sets of OWL axioms or RDF triples. The lack of established representation for consistently reusable

Description Logic:

$$\mathsf{Margherita} \sqsubseteq \mathsf{NamedPizza} \tag{1}$$
$$\mathsf{Margherita} \sqsubseteq \exists\, \mathsf{hasCountryOfOrigin}.\{\mathsf{Italy}\} \tag{2}$$
$$\mathsf{Margherita} \sqsubseteq \exists\, \mathsf{hasTopping}.\mathsf{Mozzarella} \tag{3}$$
$$\mathsf{Margherita} \sqsubseteq \exists\, \mathsf{hasTopping}.\mathsf{Tomato} \tag{4}$$
$$\mathsf{Margherita} \sqsubseteq \forall\, \mathsf{hasTopping}.(\mathsf{Mozzarella} \sqcup \mathsf{Tomato}) \tag{5}$$

Manchester OWL:

```
Class: Margherita
  SubClassOf:
    NamedPizza,
    hasCountryOfOrigin some { Italy },
    hasTopping some Mozzarella,
    hasTopping some Tomato,
    hasTopping only (Mozzarella or Tomato)
```

RDF Turtle:

```
ex:Margherita
  rdfs:subClassOf  p:NamedPizza ,
    [ a  owl:Restriction ;
      owl:onProperty  p:hasCountryOfOrigin ;
      owl:hasValue  ex:Italy ] ,
    [ a  owl:Restriction ;
      owl:onProperty  p:hasTopping ;
      owl:allValuesFrom  [ a  owl:Class ;
                           owl:unionOf  ( ex:Mozzarella ex:Tomato ) ] ] ,
    [ a  owl:Restriction ;
      owl:onProperty  p:hasTopping ;
      owl:someValuesFrom  ex:Tomato ] ,
    [ a  owl:Restriction ;
      owl:onProperty  p:hasTopping ;
      owl:someValuesFrom  ex:Mozzarella ] .
```

Figure 2 Margherita pizza represented as description logic axioms, in OWL Manchester syntax, and in RDF Turtle.

modelling patterns is also evident in today's documentation of vocabularies and ontologies and ontology design patterns [11, 4]. Here, current practice is usually limited to at most textual descriptions, illustrative and informal diagrams, and samples of OWL files that describe and illustrate how to use the resource. These offer little tangible practical help in building knowledge graphs and ontologies at scale. Following best practice descriptions requires considerable manual effort and the result is prone to errors due to the tolerant nature of RDF and RDFS vocabularies unless some constraint language like SHACL [28] is used.

The *Reasonable Ontology Templates (OTTR)* framework [50, 51, 52] is created to fill these gaps. OTTR is a macro-like [58] templating mechanism with which modelling patterns can be represented and instantiated by nested and parameterized templates. Using the OTTR framework, the pizza pattern used in Figure 2 can be represented by an OTTR Template `o-p:NamedPizza` (presented in detail in Section 2), and instances of the template can be used to express replicas of

```
o-p:NamedPizza(ex:Margherita, ex:Italy, (ex:Mozzarella, ex:Tomato)) .
```

Figure 3 Margherita pizza represented as an OTTR template instance.

the pattern; the Margherita pizza in Figure 2 can be represented succinctly and precisely with the OTTR template instance found in Figure 3 that specifies the arguments to parameterized template. Templates can be documented and shared as template libraries targeted for different users at different abstraction levels, and be efficiently instantiated using the OTTR framework's bulk instantiation tools.

Introducing the use of succinctly represented patterns and pattern instances to knowledge graph engineering allows interaction with RDF and OWL knowledge bases at a higher level of abstraction than that of RDF triples and OWL axioms. This brings with it many favourable properties such as adherence to the do-not-repeat-yourself (DRY) principle, encapsulation of complexity, separation of concerns, and better support for different user groups. Templates are also useful for documenting typical modelling use cases, such as vocabulary uses and ontology design patterns. Representing modelling patterns as identifiable templates, allows them to be shared online in a precise and actionable manner, and leads arguably to more modelling uniformity and increased efficiency and quality of knowledge base modelling tasks.

While OTTR at its core is a generic templating language, it is one of few practical pattern-based frameworks that is specifically designed for the construction of knowledge graphs and ontologies to be serialized in RDF, and with demonstrated use in the construction of large-scale ontologies and knowledge graphs [50, 49, 6, 55]. As such, the OTTR framework is an advance of the state of the art of ontology engineering [23, 56] and ontology design pattern [11, 4] tools and methodologies.

The OTTR framework has been presented in a series of papers [50, 51, 52, 34]. These papers have presented and characterized the OTTR language at a conceptual level and demonstrated different uses of the framework. The OTTR framework has since then gradually matured to a stable state with multiple different independent implementations and applications by prominent ontology development projects. The purpose of this resource paper is to give a complete and self-contained presentation of the resources that now comprise the OTTR framework: specifications, core template library, reference implementation, and project infrastructure. The paper gives emphasis to the specifications of the formal syntax and semantics of the OTTR language and its implementation for semantic web, which is given in Section 3 and Section 4, and the formal specification of the mapping languages for instantiating templates, which is presented in Section 6. These introduce an abstract and formal model and vocabulary for characterizing the OTTR language that form the basis of the reference implementation. These specifications have not been published before in this rigorous form and are necessary to fully understand the OTTR framework. Section 5 gives an overview of the motivation and support for developing and maintaining template libraries. We also give an updated overview of the OTTR framework's impact, including publicly available template libraries, implementations of the OTTR framework in Section 7, and a selection of industrial and academic uses in Section 8. Section 9 presents related work and Section 10 presents lessons learned and ideas for future development of the OTTR framework collected throughout the project from experience and interaction with its users. Section 11 concludes the paper. First, Section 2 presents an overview of the OTTR framework to tie all the resources together and gives examples to establish intuitions for the following more technical sections.

2 Overview

The OTTR framework is formally described by a series of specifications that define:

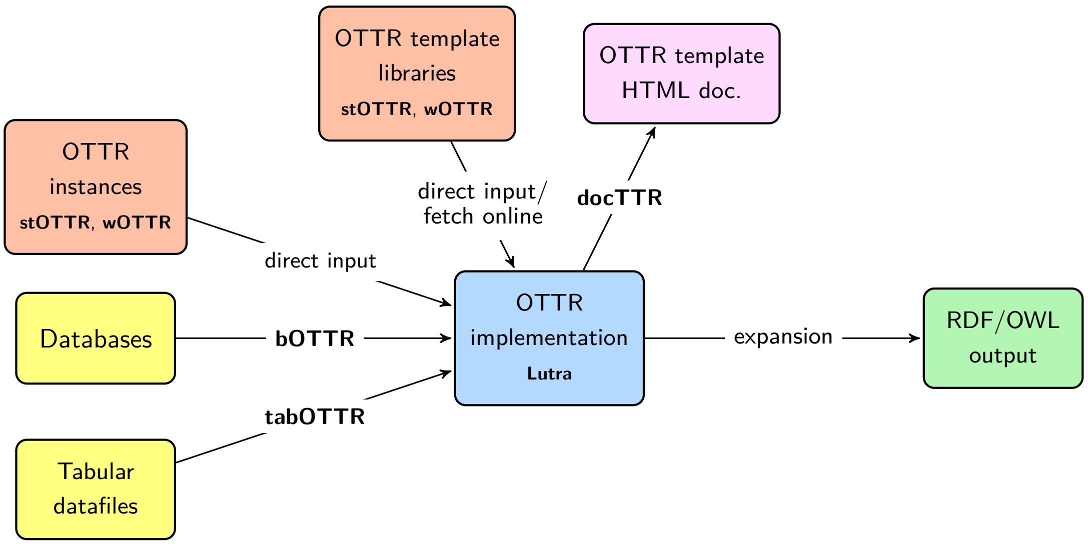


Figure 4 High-level OTTR framework architecture.

- an abstract language for characterizing templates and template instances and the process of expanding template instances,
- serialization formats for representing templates and instances (stOTTR and wOTTR),
- a mapping language for consuming data from queryable databases as template instances (bOTTR), and
- a mapping language for annotating and consuming data from tabular datafiles as template instances (tabOTTR).

Additionally, the framework consists of:

- a template library of basic templates called the core template library that mostly contains templates that represent basic modelling patterns over the vocabularies RDF, RDFS and OWL,
- a tool-supported best practice description of how to document and publish template libraries (docTTR), and
- a reference implementation that supports all the specifications of the framework (Lutra).

Figure 4 shows a high-level architecture diagram of the OTTR framework.

The primary uses of the OTTR framework are to represent and document useful modelling patterns in a precise and actionable manner as (shared) OTTR template libraries, and to use such libraries to expand OTTR template instances to RDF data. The consumed instances can be described either directly using one of the OTTR serialization formats or by way of mappings that extract or identify instances in tabular data sources such as database query results or tabular datafiles. The OTTR template language has different features to guarantee the correctness of the output, and verifying the input according to these correctness measures is a core feature of the framework. These features also help to reveal the intended and correct instantiations of templates and play an important role in the documentation of templates. The following sections present an overview of the OTTR language, the concept behind template libraries and bulk instantiation of templates.

`ottr:Triple` base template:

```
ottr:Triple [
  ottr:IRI ?subject, ! ottr:IRI ?predicate, rdfs:Resource ?object ] ::
BASE .
```

`o-owl-ax:SubClassOf` template:

```
o-owl-ax:SubClassOf[
  owl:Class ?subclass, owl:Class ?superclass ] ::
{
  ottr:Triple(?subclass, rdfs:subClassOf, ?superclass)
} .
```

`o-p:NamedPizza` template:

```
o-p:NamedPizza[
  owl:Class ?pizza, ? owl:NamedIndividual ?country, NEList<owl:Class> ?toppings ] ::
{
  o-owl-ax:SubClassOf(?pizza, pz:NamedPizza),
  o-owl-ax:SubObjectHasValue(?pizza, pz:hasCountryOfOrigin, ?country),
  cross | o-owl-ax:SubObjectSomeValuesFrom(?pizza, pz:hasTopping, ++?toppings),
  o-owl-ax:SubObjectAllValuesFrom(?pizza, pz:hasTopping, _:toppingUnion),
  o-owl-re:ObjectUnionOf(_:toppingUnion, ?toppings)
} .
```

`o-p:NamedPizza` instances:

```
o-p:NamedPizza(ex:Margherita, ex:Italy, (ex:Mozzarella, ex:Tomato)) .

o-p:NamedPizza(ex:PlainHam, none, (ex:Mozzarella, ex:Tomato, ex:Ham)) .

o-p:NamedPizza(ex:Hawaiian, ex:Canada,
    (ex:Mozzarella, ex:Tomato, ex:Pineapple, ex:Ham)) .
```

Figure 5 OTTR templates and instances representing different pizzas.

2.1 Language

The OTTR language and its features will be introduced in an incremental and example-driven approach that builds on the example established in the introduction. The complete specification of the OTTR language is found in Section 3.

2.1.1 Templates, base templates and instances

A *template* has a *signature* that assigns an IRI to the template and lists its *parameters* that specify its permissive *instances*. An *instance* refers to a template's IRI and lists *arguments* that must match the parameters of the referenced template. The template *body* contains instances of other templates and specifies hence how its instances can be *expanded* into instances of templates at a lower level of abstraction; this hierarchy of templates is required to be non-cyclic. At the lowest level of abstraction in the hierarchy of templates are *base templates* that specify how instances should be interpreted into a different representation language, such as RDF. Base templates do not have a body; the translation of base template instances to the underlying representation language is handled by an OTTR implementation that must follow a textual specification of how base templates must be interpreted.

`o-owl-ax:SubClassOf` and `ottr:Triple` instances:

```
1 o-owl-ax:SubClassOf(ex:A, ex:B) .
2
3 ottr:Triple(ex:A, rdfs:subclassOf, ex:B) .
```

Expansion result:
⟨ `ex:A`, `rdfs:subClassOf`, `ex:B` ⟩

Figure 6 `o-owl-ax:SubClassOf` and `ottr:Triple` instances, and their expansion result.

▶ **Example 1.** Figure 5 contains three templates, the base template `ottr:Triple`, and the (regular) templates `o-owl-ax:SubClassOf` and `o-p:NamedPizza`; and instances of the `o-p:NamedPizza` template. All examples in this section are serialized using the stOTTR format. The example templates are formatted so that their signatures are contained in the two first lines of each of the code listings. The remaining lines contain the template body. Instead of a body, the `ottr:Triple` base template is marked with the token `BASE`.

Template instances are *expanded* by recursively replacing an instance with its referenced template's body's instances where the parameters are appropriately substituted by the instance's arguments, akin to unfolding macros. This process terminates with a set of base template instances that can be translated to the underlying representation language as per the specification. A template can hence be understood to represent a mapping from its signature instance format to a set of statements over an underlying language represented by base templates, via a nested non-cyclic template structure.

▶ **Example 2.** The signature of the `ottr:Triple` template in Figure 5 specifies three parameters: `?subject`, `?predicate` and `?object`. (The example also includes parameter types and modifiers which will be explained shortly.) The body of the `o-owl-ax:SubClassOf` template contains one instance of the `ottr:Triple` template where the parameters of the `o-owl-ax:SubClassOf` template are used as parameters. Figure 6 demonstrates the expansion of instances; the example instance of the `o-owl-ax:SubClassOf` instance in line 1 is expanded in one step to the `ottr:Triple` instance in line 3, which represents the RDF triple as shown in the figure.

▶ **Example 3.** The `o-p:NamedPizza` template in Figure 5 is a faithful representation of the pizza modelling pattern used in the Pizza Ontology. The body of the `o-p:NamedPizza` template contains instances of the `o-owl-ax:SubClassOf` template and other templates that represent common OWL axioms and constructs. The first template instance in Figure 5 expressing a Margherita pizza expands in multiple steps to an RDF graph that is equivalent to the RDF graph found in Figure 2 on page 4.

2.1.2 Parameter types and non-blank flags

Parameter types are used to check that templates are correctly instantiated and specified; the arguments' types must be *compatible* with the types of the parameters where the arguments are used, and this must also hold when parameters are used as arguments in template bodies. The OTTR language also contains *parameter modifiers*, where *non-blank* is one such parameter modifier that forbids RDF blank nodes as arguments. OTTR implementations must emit errors when instances and template violate these parameter type specifications.

▶ **Example 4.** The signature of the `ottr:Triple` template assigns types to its parameters; the `?subject` and `?predicate` parameters have the type `ottr:IRI`, and the `?object` has the type `rdfs:Resource`. These parameter types guarantee that no `ottr:Triple` instance can, for example, have a literal in subject position, which would be a violation of the RDF specification [27], since the type assigned to literals is specified by the type system as incompatible with the parameter type `ottr:IRI`. The following `ottr:Triple` instance contains two type errors: the literal values `"A"` and `"B"` are arguments to parameters with the type `ottr:IRI`.

```
ottr:Triple("A", "B", "C") .
```

▶ **Example 5.** In the body of the `o-owl-ax:SubClassOf` template, the types of the parameters `?subclass` (`owl:Class`) and `?superclass` (also `owl:Class`) must be compatible with the types of the first (`ottr:IRI`) and third parameter (`rdfs:Resource`) of the `ottr:Triple` template, respectively – which they are. Furthermore, the parameter types of `o-owl-ax:SubClassOf` template force for example the parameter type of `o-p:NamedPizza`'s `?pizza` parameter to have a type that is compatible with `owl:Class`, since `?pizza` is passed on as an argument to a parameter with this type.

▶ **Example 6.** The `ottr:Triple` signature specifies, using an exclamation mark `!`, the `?predicate` parameter to be non-blank. This ensures that no RDF triple constructed using this template will end up with a blank node in predicate position, which would be a violation of the RDF specification [27]. The following `ottr:Triple` instance violates the non-blank modifier.

```
ottr:Triple(ex:A, _:blank, "C") .
```

2.1.3 Optional parameters and none values

Parameters may be specified as being *optional*, whereas parameters that are not optional are called mandatory. Whether a parameter is optional or not has consequences for the treatment of *none values*, which in OTTR is represented by the reserved token `none` and is used to indicate a missing value. In the expansion of instances, a none value given as an argument to a mandatory parameter is simply ignored and will not contribute to the end result of the expansion – the instance is simply removed. A none value given to an optional argument, on the other hand, will be passed on to body template instances just like other arguments.

▶ **Example 7.** The second argument of the `o-p:NamedPizza` template is marked as optional, using a question mark `?`. This means that instances of the template do not need to specify a country of origin. The `ex:PlainHam` example instance demonstrates this. Here, the none value will be passed on as an argument to the third parameter of the `o-owl-ax:SubObjectHasValue` template. This parameter is mandatory, hence there will be no OWL axiom in the expansion result that expresses the country of origin of the `ex:PlainHam` pizza, however, the other axioms will remain. If the `?country` parameter of the `o-p:NamedPizza` had not been marked as optional, then the `ex:PlainHam` instance would have been simply removed in the first expansion step.

2.1.4 Default values

Parameters may be given a *default value*. This default value is used whenever a none value is given as an argument to the parameter.

▶ **Example 8.** Figure 7 demonstrates the use of a default valued parameter using an alternative signature to the `o-p:NamedPizza` template that assigns `ex:Italy` as the default value to the second argument. The `ex:PlainHam` example instance in Figure 5 would under this signature get `ex:Italy` as its country of origin.

```
o-p:NamedPizza[
  owl:Class ?pizza,
  owl:NamedIndividual ?country = ex:Italy,
  NEList<owl:Class> ?toppings
] .
```

Figure 7 `o-p:NamedPizza` with default valued parameter.

2.1.5 Expansion modes and list values

Template instances can be marked with an *expansion mode* which is only applicable to instances that have arguments that are lists. An expansion mode applied to an instance with one list argument specifies that the selected instance will be instantiated multiple times, one per element in the marked argument list. There are different expansion modes that behave differently when multiple lists are marked in an instance.

▶ **Example 9.** The `o-p:NamedPizza` template makes use of expansion modes, indicated with the token `cross` and by marking the list-typed parameter `?toppings` with `++`:

```
  cross | o-owl-ax:SubObjectSomeValuesFrom(?pizza, pz:hasTopping, ++?toppings),
```

The effects of the expansion mode are that one instance of the `o-owl-ax:SubObjectSomeValues-From` template will be created for each element in the `?toppings` list, e.g.,

```
cross | o-owl-ax:SubObjectSomeValuesFrom(ex:Margherita, pz:hasTopping,
  ++(ex:Mozzarella, ex:Tomato)) .
```

will expand in one step to:

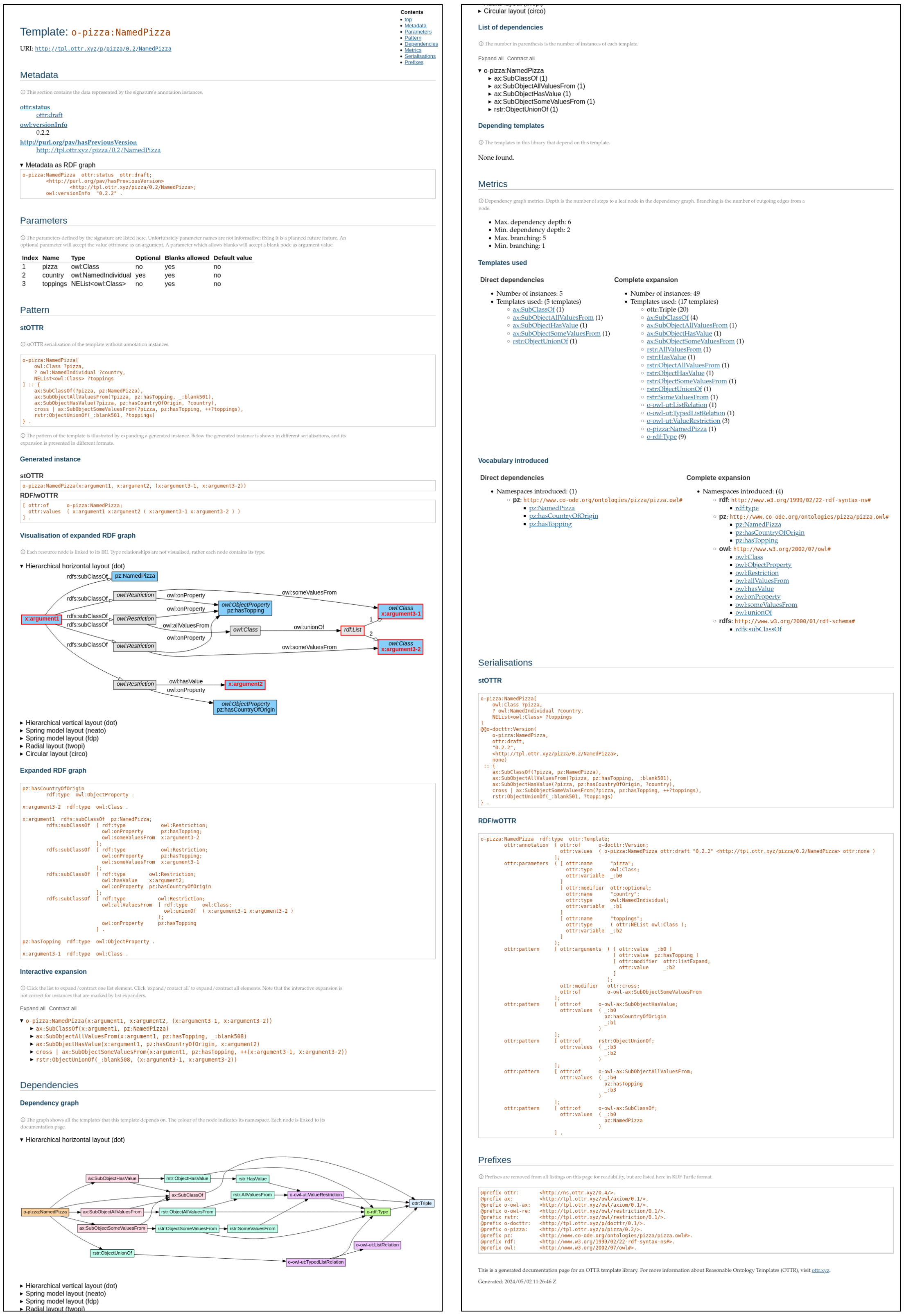
```
o-owl-ax:SubObjectSomeValuesFrom(ex:Margherita, pz:hasTopping, ex:Mozzarella) .
o-owl-ax:SubObjectSomeValuesFrom(ex:Margherita, pz:hasTopping, ex:Tomato) .
```

2.2 Template Libraries

A *template library* is a collection of templates developed and curated for a particular purpose, such as representing patterns for a given vocabulary, domain, or project. The ability to share and reuse templates for common modelling patterns is central to the OTTR framework and will be further elaborated in Section 5. By following best practices and principles similar to linked open data [19] and ontology publication, templates and template libraries are expected to be published and interconnected in a distributed and decentralized manner, promoting their reuse and community-driven curation. Our intention is that template libraries should be developed alongside the development of vocabularies and ontologies which are intended for reuse, in order to promote and simplify correct and consistent typical use of the vocabulary or ontology. Given that a template's signature is clearly documented and understood, there is no need to understand how the template is implemented in order to correctly instantiate the template. Templates at different abstraction levels, and templates and their instances, target different users and use cases, and can hence be created and managed separately and by different users.

▶ **Example 10.** The `o-p:NamedPizza` template is published at its IRI, `https://tpl.ottr.xyz/p/pizza/0.2/NamedPizza`, using content negotiation [44] to serve different presentations of the template,[4] including an HTML documentation page generated by the OTTR's docTTR tool which

[4] `https://tpl.ottr.xyz/p/pizza/0.2/NamedPizza.html`,



Template: `o-pizza:NamedPizza`

URI: http://tpl.ottr.xyz/p/pizza/0.2/NamedPizza

Metadata

ⓘ This section contains the data represented by the signature's annotation instances.

ottr:status
  ottr:draft
owl:versionInfo
  0.2.2
http://purl.org/pav/hasPreviousVersion
  http://tpl.ottr.xyz/pizza/0.2/NamedPizza

▾ Metadata as RDF graph

```
o-pizza:NamedPizza  ottr:status  ottr:draft;
        <http://purl.org/pav/hasPreviousVersion>
                <http://tpl.ottr.xyz/pizza/0.2/NamedPizza>;
        owl:versionInfo  "0.2.2" .
```

Parameters

ⓘ The parameters defined by the signature are listed here. Unfortunately parameter names are not informative; fixing it is a planned future feature. An optional parameter will accept the value ottr:none as an argument. A parameter which allows blanks will accept a blank node as argument value.

Index	Name	Type	Optional	Blanks allowed	Default value
1	pizza	owl:Class	no	yes	no
2	country	owl:NamedIndividual	yes	yes	no
3	toppings	NEList<owl:Class>	no	yes	no

Pattern

stOTTR

ⓘ stOTTR serialisation of the template without annotation instances.

```
o-pizza:NamedPizza[
    owl:Class ?pizza,
    ? owl:NamedIndividual ?country,
    NEList<owl:Class> ?toppings
] :: {
    ax:SubClassOf(?pizza, pz:NamedPizza),
    ax:SubObjectAllValuesFrom(?pizza, pz:hasTopping, _:blank501),
    ax:SubObjectHasValue(?pizza, pz:hasCountryOfOrigin, ?country),
    cross | ax:SubObjectSomeValuesFrom(?pizza, pz:hasTopping, ++?toppings),
    rstr:ObjectUnionOf(_:blank501, ?toppings)
} .
```

ⓘ The pattern of the template is illustrated by expanding a generated instance. Below the generated instance is shown in different serialisations, and its expansion is presented in different formats.

Generated instance

stOTTR

```
o-pizza:NamedPizza(x:argument1, x:argument2, (x:argument3-1, x:argument3-2))
```

RDF/wOTTR

```
[ ottr:of      o-pizza:NamedPizza;
  ottr:values  ( x:argument1 x:argument2 ( x:argument3-1 x:argument3-2 ) )
] .
```

Visualisation of expanded RDF graph

ⓘ Each resource node is linked to its IRI. Type relationships are not visualised, rather each node contains its type.

▾ Hierarchical horizontal layout (dot)

▸ Hierarchical vertical layout (dot)
▸ Spring model layout (neato)
▸ Spring model layout (fdp)
▸ Radial layout (twopi)
▸ Circular layout (circo)

Expanded RDF graph

```
pz:hasCountryOfOrigin
        rdf:type  owl:ObjectProperty .

x:argument3-2  rdf:type  owl:Class .

x:argument1  rdfs:subClassOf  pz:NamedPizza;
        rdfs:subClassOf  [ rdf:type            owl:Restriction;
                           owl:onProperty      pz:hasTopping;
                           owl:someValuesFrom  x:argument3-2
                         ];
        rdfs:subClassOf  [ rdf:type            owl:Restriction;
                           owl:onProperty      pz:hasTopping;
                           owl:someValuesFrom  x:argument3-1
                         ];
        rdfs:subClassOf  [ rdf:type        owl:Restriction;
                           owl:hasValue    x:argument2;
                           owl:onProperty  pz:hasCountryOfOrigin
                         ];
        rdfs:subClassOf  [ rdf:type           owl:Restriction;
                           owl:allValuesFrom  [ rdf:type     owl:Class;
                                                owl:unionOf  ( x:argument3-1 x:argument3-2 )
                                              ];
                           owl:onProperty     pz:hasTopping
                         ] .

pz:hasTopping  rdf:type  owl:ObjectProperty .

x:argument3-1  rdf:type  owl:Class .
```

Interactive expansion

ⓘ Click the list to expand/contract one list element. Click 'expand/contact all' to expand/contract all elements. Note that the interactive expansion is not correct for instances that are marked by list expanders.

Expand all Contract all

```
▾ o-pizza:NamedPizza(x:argument1, x:argument2, (x:argument3-1, x:argument3-2))
   ▸ ax:SubClassOf(x:argument1, pz:NamedPizza)
   ▸ ax:SubObjectAllValuesFrom(x:argument1, pz:hasTopping, _:blank508)
   ▸ ax:SubObjectHasValue(x:argument1, pz:hasCountryOfOrigin, x:argument2)
   ▸ cross | ax:SubObjectSomeValuesFrom(x:argument1, pz:hasTopping, ++(x:argument3-1, x:argument3-2))
   ▸ rstr:ObjectUnionOf(_:blank508, (x:argument3-1, x:argument3-2))
```

Dependencies

Dependency graph

ⓘ The graph shows all the templates that this template depends on. The colour of the node indicates its namespace. Each node is linked to its documentation page.

▾ Hierarchical horizontal layout (dot)

▸ Hierarchical vertical layout (dot)
▸ Spring model layout (neato)
▸ Spring model layout (fdp)
▸ Radial layout (twopi)
▸ Circular layout (circo)

List of dependencies

ⓘ The number in parenthesis is the number of instances of each template.

Expand all Contract all

▾ o-pizza:NamedPizza
  ▸ ax:SubClassOf (1)
  ▸ ax:SubObjectAllValuesFrom (1)
  ▸ ax:SubObjectHasValue (1)
  ▸ ax:SubObjectSomeValuesFrom (1)
  ▸ rstr:ObjectUnionOf (1)

Depending templates

ⓘ The templates in this library that depend on this template.

None found.

Metrics

ⓘ Dependency graph metrics. Depth is the number of steps to a leaf node in the dependency graph. Branching is the number of outgoing edges from a node.

- Max. dependency depth: 6
- Min. dependency depth: 2
- Max. branching: 5
- Min. branching: 1

Templates used

Direct dependencies

- Number of instances: 5
- Templates used: (5 templates)
  - ax:SubClassOf (1)
  - ax:SubObjectAllValuesFrom (1)
  - ax:SubObjectHasValue (1)
  - ax:SubObjectSomeValuesFrom (1)
  - rstr:ObjectUnionOf (1)

Complete expansion

- Number of instances: 49
- Templates used: (17 templates)
  - ottr:Triple (20)
  - ax:SubClassOf (4)
  - ax:SubObjectAllValuesFrom (1)
  - ax:SubObjectHasValue (1)
  - ax:SubObjectSomeValuesFrom (1)
  - rstr:AllValuesFrom (1)
  - rstr:HasValue (1)
  - rstr:ObjectAllValuesFrom (1)
  - rstr:ObjectHasValue (1)
  - rstr:ObjectSomeValuesFrom (1)
  - rstr:ObjectUnionOf (1)
  - rstr:SomeValuesFrom (1)
  - o-owl-ut:ListRelation (1)
  - o-owl-ut:TypedListRelation (1)
  - o-owl-ut:ValueRestriction (3)
  - o-pizza:NamedPizza (1)
  - o-rdf:Type (9)

Vocabulary introduced

Direct dependencies

- Namespaces introduced: (1)
  - **pz**: `http://www.co-ode.org/ontologies/pizza/pizza.owl#`
    - pz:NamedPizza
    - pz:hasCountryOfOrigin
    - pz:hasTopping

Complete expansion

- Namespaces introduced: (4)
  - **rdf**: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
    - rdf:type
  - **pz**: `http://www.co-ode.org/ontologies/pizza/pizza.owl#`
    - pz:NamedPizza
    - pz:hasCountryOfOrigin
    - pz:hasTopping
  - **owl**: `http://www.w3.org/2002/07/owl#`
    - owl:Class
    - owl:ObjectProperty
    - owl:Restriction
    - owl:allValuesFrom
    - owl:hasValue
    - owl:onProperty
    - owl:someValuesFrom
    - owl:unionOf
  - **rdfs**: `http://www.w3.org/2000/01/rdf-schema#`
    - rdfs:subClassOf

Serialisations

stOTTR

```
o-pizza:NamedPizza[
    owl:Class ?pizza,
    ? owl:NamedIndividual ?country,
    NEList<owl:Class> ?toppings
]
@@o-docttr:Version(
    o-pizza:NamedPizza,
    ottr:draft,
    "0.2.2",
    <http://tpl.ottr.xyz/pizza/0.2/NamedPizza>,
    none)
 :: {
    ax:SubClassOf(?pizza, pz:NamedPizza),
    ax:SubObjectAllValuesFrom(?pizza, pz:hasTopping, _:blank501),
    ax:SubObjectHasValue(?pizza, pz:hasCountryOfOrigin, ?country),
    cross | ax:SubObjectSomeValuesFrom(?pizza, pz:hasTopping, ++?toppings),
    rstr:ObjectUnionOf(_:blank501, ?toppings)
} .
```

RDF/wOTTR

```
o-pizza:NamedPizza  rdf:type  ottr:Template;
        ottr:annotation  [ ottr:of      o-docttr:Version;
                           ottr:values  ( o-pizza:NamedPizza ottr:draft "0.2.2" <http://tpl.ottr.xyz/pizza/0.2/NamedPizza> ottr:none )
                         ];
        ottr:parameters  ( [ ottr:name      "pizza";
                             ottr:type      owl:Class;
                             ottr:variable  _:b0
                           ]
                           [ ottr:modifier  ottr:optional;
                             ottr:name      "country";
                             ottr:type      owl:NamedIndividual;
                             ottr:variable  _:b1
                           ]
                           [ ottr:name      "toppings";
                             ottr:type      ( ottr:NEList owl:Class );
                             ottr:variable  _:b2
                           ]
                         );
        ottr:pattern     [ ottr:arguments  ( [ ottr:value  _:b0 ]
                                             [ ottr:value  pz:hasTopping ]
                                             [ ottr:modifier  ottr:listExpand;
                                               ottr:value     _:b2
                                             ]
                                           );
                           ottr:modifier   ottr:cross;
                           ottr:of         o-owl-ax:SubObjectSomeValuesFrom
                         ];
        ottr:pattern     [ ottr:of      o-owl-ax:SubObjectHasValue;
                           ottr:values  ( _:b0
                                          pz:hasCountryOfOrigin
                                          _:b1
                                        )
                         ];
        ottr:pattern     [ ottr:of      rstr:ObjectUnionOf;
                           ottr:values  ( _:b3
                                          _:b2
                                        )
                         ];
        ottr:pattern     [ ottr:of      o-owl-ax:SubObjectAllValuesFrom;
                           ottr:values  ( _:b0
                                          pz:hasTopping
                                          _:b3
                                        )
                         ];
        ottr:pattern     [ ottr:of      o-owl-ax:SubClassOf;
                           ottr:values  ( _:b0
                                          pz:NamedPizza
                                        )
                         ] .
```

Prefixes

ⓘ Prefixes are removed from all listings on this page for readability, but are listed here in RDF Turtle format.

```
@prefix ottr:       <http://ns.ottr.xyz/0.4/>.
@prefix ax:         <http://tpl.ottr.xyz/owl/axiom/0.1/>.
@prefix o-owl-ax:   <http://tpl.ottr.xyz/owl/axiom/0.1/>.
@prefix o-owl-re:   <http://tpl.ottr.xyz/owl/restriction/0.1/>.
@prefix rstr:       <http://tpl.ottr.xyz/owl/restriction/0.1/>.
@prefix o-docttr:   <http://tpl.ottr.xyz/p/docttr/0.1/>.
@prefix o-pizza:    <http://tpl.ottr.xyz/p/pizza/0.2/>.
@prefix pz:         <http://www.co-ode.org/ontologies/pizza/pizza.owl#>.
@prefix rdf:        <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl:        <http://www.w3.org/2002/07/owl#>.
```

This is a generated documentation page for an OTTR template library. For more information about Reasonable Ontology Templates (OTTR), visit ottr.xyz.

Generated: 2024/05/02 11:26:46 Z

Figure 8 The generated docTTR documentation page for the `o-p:NamedPizza` template.

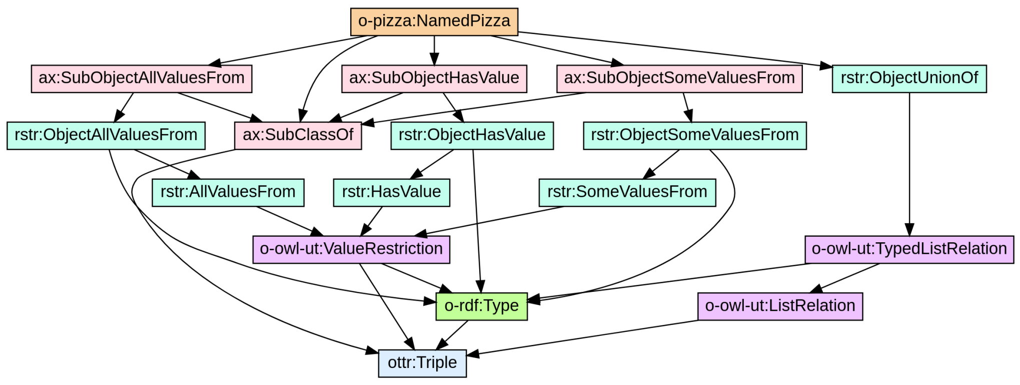


Figure 9 Layered dependencies between templates used by the `o-p:NamedPizza` template.

is shown in Figure 8. The `o-p:NamedPizza` template is an example template in the Core OTTR template library [52], which is available at `https://tpl.ottr.xyz`. The Core OTTR template library contains all the templates used in the examples.

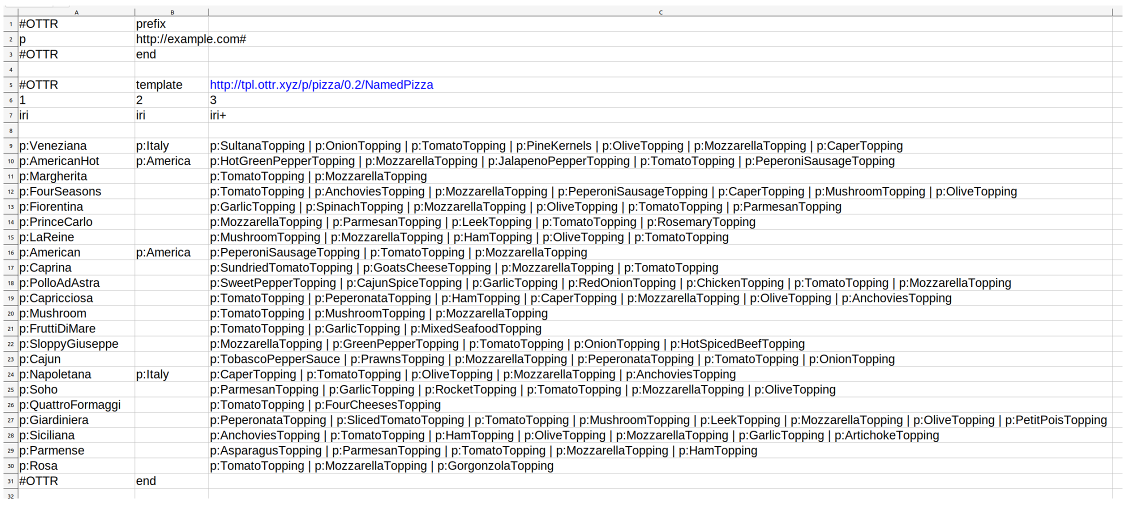
Figure 9 shows the dependency graph with the `o-p:NamedPizza` on top and the base template `ottr:Triple` at the bottom. Observe that the graph is divided into different layers: the "user-facing" `o-p:NamedPizza`, "logical" OWL templates, including `o-owl-ax:SubClassOf`, "utility" templates that represent OWL restrictions and different RDF list patterns, and the low-level base template `ottr:Triple`. Each layer represents different a level of abstractions that hide the complexity of lower levels.

2.3 Template Instantiation

Efficient instantiation of templates is also central to the OTTR framework. For this task, it is natural to consider a template as a mapping from its signature input format to the pattern of its expansion. The OTTR framework provides two specifications, bOTTR and tabOTTR, for selecting and translating data from structured sources into template instances, which in turn can be expanded into a knowledge graph or ontology according to the corresponding template definitions. These are presented in Section 6. The bOTTR specification defines an RDF vocabulary with which mappings from database query results to templates may be specified. The tabOTTR specification describes a simple "markup" language for defining mappings to templates directly in tabular datafiles, such as CSV, TSV or Excel files. These specifications permit the OTTR framework to become a part of a complete data transformation pipeline, where external tools may be used to cleanse and prepare data for template instantiation, and the OTTR framework's mapping specification may be used to collect and integrate data from multiple sources to build knowledge graphs and ontologies at scale.

▶ **Example 11.** Figure 10 demonstrates the use of tabOTTR. It shows a spreadsheet that contains arguments to 22 instances of the `o-p:NamedPizza` template in Figure 5, and that uses tabOTTR processing instructions to describe how the data is to be understood as instances.

`https://tpl.ottr.xyz/p/pizza/0.2/NamedPizza.stottr`,
`https://tpl.ottr.xyz/p/pizza/0.2/NamedPizza.ttl`.



	A	B	C
1	#OTTR	prefix	
2	p	http://example.com#	
3	#OTTR	end	
4			
5	#OTTR	template	http://tpl.ottr.xyz/p/pizza/0.2/NamedPizza
6	1	2	3
7	iri	iri	iri+
8			
9	p:Veneziana	p:Italy	p:SultanaTopping | p:OnionTopping | p:TomatoTopping | p:PineKernels | p:OliveTopping | p:MozzarellaTopping | p:CaperTopping
10	p:AmericanHot	p:America	p:HotGreenPepperTopping | p:MozzarellaTopping | p:JalapenoPepperTopping | p:TomatoTopping | p:PeperoniSausageTopping
11	p:Margherita		p:TomatoTopping | p:MozzarellaTopping
12	p:FourSeasons		p:TomatoTopping | p:AnchoviesTopping | p:MozzarellaTopping | p:PeperoniSausageTopping | p:CaperTopping | p:MushroomTopping | p:OliveTopping
13	p:Fiorentina		p:GarlicTopping | p:SpinachTopping | p:MozzarellaTopping | p:OliveTopping | p:TomatoTopping | p:ParmesanTopping
14	p:PrinceCarlo		p:MozzarellaTopping | p:ParmesanTopping | p:LeekTopping | p:TomatoTopping | p:RosemaryTopping
15	p:LaReine		p:MushroomTopping | p:MozzarellaTopping | p:HamTopping | p:OliveTopping | p:TomatoTopping
16	p:American	p:America	p:PeperoniSausageTopping | p:TomatoTopping | p:MozzarellaTopping
17	p:Caprina		p:SundriedTomatoTopping | p:GoatsCheeseTopping | p:MozzarellaTopping | p:TomatoTopping
18	p:PolloAdAstra		p:SweetPepperTopping | p:CajunSpiceTopping | p:GarlicTopping | p:RedOnionTopping | p:ChickenTopping | p:TomatoTopping | p:MozzarellaTopping
19	p:Capricciosa		p:TomatoTopping | p:PeperonataTopping | p:HamTopping | p:CaperTopping | p:MozzarellaTopping | p:OliveTopping | p:AnchoviesTopping
20	p:Mushroom		p:TomatoTopping | p:MushroomTopping | p:MozzarellaTopping
21	p:FruttiDiMare		p:TomatoTopping | p:GarlicTopping | p:MixedSeafoodTopping
22	p:SloppyGiuseppe		p:MozzarellaTopping | p:GreenPepperTopping | p:TomatoTopping | p:OnionTopping | p:HotSpicedBeefTopping
23	p:Cajun		p:TobascoPepperSauce | p:PrawnsTopping | p:MozzarellaTopping | p:PeperonataTopping | p:TomatoTopping | p:OnionTopping
24	p:Napoletana	p:Italy	p:CaperTopping | p:TomatoTopping | p:OliveTopping | p:MozzarellaTopping | p:AnchoviesTopping
25	p:Soho		p:ParmesanTopping | p:GarlicTopping | p:RocketTopping | p:TomatoTopping | p:MozzarellaTopping | p:OliveTopping
26	p:QuattroFormaggi		p:TomatoTopping | p:FourCheesesTopping
27	p:Giardiniera		p:PeperonataTopping | p:SlicedTomatoTopping | p:TomatoTopping | p:MushroomTopping | p:LeekTopping | p:MozzarellaTopping | p:OliveTopping | p:PetitPoisTopping
28	p:Siciliana		p:AnchoviesTopping | p:TomatoTopping | p:HamTopping | p:OliveTopping | p:MozzarellaTopping | p:GarlicTopping | p:ArtichokeTopping
29	p:Parmense		p:AsparagusTopping | p:ParmesanTopping | p:TomatoTopping | p:MozzarellaTopping | p:HamTopping
30	p:Rosa		p:TomatoTopping | p:MozzarellaTopping | p:GorgonzolaTopping
31	#OTTR	end	
32			

Figure 10 Spreadsheet using tabOTTR to specify 22 instances of the `o-p:NamedPizza` template.

2.4 History

An early predecessor and inspiration to OTTR templates dates back to 2008 [26]. Here, a template mechanism was developed for "lifting" compact data representations, typically tabular data, to rich semantic format according to a complex upper level ontology. A prototype of the template mechanism was implemented using OWL and SWRL rules [18].

The practical and theoretical aspects of OTTR templates were first introduced in 2017, where OTTR templates were defined parameterized knowledge bases using a dedicated OWL ontology [48], and as description logic macros [10].[5]

The formulation of the OTTR language later matured into a dedicated representation of templates and template instances [50, 51]. The OTTR language has since then evolved into a framework and reached the state of stable resource, far beyond a research prototype – with multiple users from different communities and industries, multiple independent implementations initiated, and several publicly available template libraries.

2.5 Resources

All publicly available resources managed by the OTTR team are available from the project web page: `https://ottr.xyz`. The formal specifications and software are hosted in the Git repository at: `https://gitlab.com/ottr/`. Stable releases are also published at Zenodo: `https://zenodo.org/communities/ottr/`.

3 Fundamentals

This section defines the formal templating mechanism that underlies the OTTR framework. The presentation follows three tracks that are given in tandem: (1) definition blocks define the conceptual and formal aspects of the OTTR framework, such as template, template instances, validity of

[5] The name "Reasonable Ontology Templates" comes partly from the fact that in the first version of OTTR, templates were parameterized OWL ontologies that could be directly reasoned over. Also, "reasonable" has a suitable double meaning of "being reasonable" and "being subject to reasoning". The acronym OTTR is inspired by OWL.

templates, and instance expansion; (2) implementation blocks describe how the conceptual model is adapted to semantic web technologies, with the specific purpose of using the OTTR framework to produce RDF graphs; and (3) syntax blocks specify the stOTTR serialization format for OTTR; more details on OTTR serialization format are given in Section 4.

We start by defining terms and types, before we introduce template instances and template objects. We then introduce template expansion, and end by defining libraries and datasets and their properties, such as correctness.

3.1 Terms and Types

OTTR is a language for expressing statements, in particular for ontologies and knowledge graphs represented using OWL and RDF. The basic building blocks for such statements are terms. As different terms may play different roles within these statements, and denote, e.g., relationships, entities, or data values, OTTR introduces a type system over the terms to ensure that terms are used correctly. The type system assigns a type to each term and uses subtype and compatibility relationships between types to check correct and consistent use of terms across ontologies and knowledge graphs. Below we introduce these terms and type systems.

▶ **Definition 12.** *We assume we have a (possibly countably infinite) set $\mathcal{C}$ of constants at least containing the elements nil and none. Furthermore, we assume we have a countably infinite set $\mathcal{V}$ of variables. Let the set of* terms *$\mathcal{E}$ be defined inductively as follows: All elements of $\mathcal{C}$ and $\mathcal{V}$ are terms, and for any finite list of terms $e_1, e_2, \ldots, e_n \in \mathcal{E}$ then $\langle e_1, e_2, \ldots, e_n \rangle \in \mathcal{E}$. Elements of $\mathcal{E}$ of the form nil and $\langle e_1, e_2, \ldots, e_n \rangle$ are called* list terms *or simply* lists. *We let length(l) denote the length of the list l and $l(i)$ denote the ith term (1-indexed) of the list l if $i \leq$ length(l) and none otherwise.*

Note that list terms can be nested arbitrarily, so if e_1, e_2, e_3 are terms, then, e.g., $\langle e_1, \langle e_2, e_3 \rangle, nil \rangle$ is also a term. The special constant *none* denotes a missing value, similar to how `NULL` or `null` is used in SQL and many programming languages. This constant is not intended to be used in the final statements that become part of constructed the knowledge graph, but only within the OTTR framework's definitions.

▶ **Definition 13.** *The set of* basic types *$\mathcal{B}$ is a set that contains at least the elements $\top$ and $\bot$, and that is partially ordered by the* subtype *relation $\leq$ such that for any $t \in \mathcal{B}$ we have $\bot \leq t$ and $t \leq \top$. The inverse relation of subtype is called* supertype.

▶ **Definition 14.** *The set of* types *$\mathcal{T}$ is the smallest set such that*
- $\mathcal{B} \subseteq \mathcal{T}$
- *if $t \in \mathcal{B}$, then $LUB\langle t \rangle \in \mathcal{T}$*
- *if $t \in \mathcal{T}$, then $List\langle t \rangle \in \mathcal{T}$*
- *if $t \in \mathcal{T}$, then $NEList\langle t \rangle \in \mathcal{T}$*

Furthermore, $\leq$ is extended to $\mathcal{T}$ as follows:
- *if $t_1 \leq t_2$ for $t_1, t_2 \in \mathcal{T}$, then $NEList\langle t_1 \rangle \leq NEList\langle t_2 \rangle$ and $List\langle t_1 \rangle \leq List\langle t_2 \rangle$*
- *if $t \in \mathcal{B}$, then $NEList\langle t \rangle \leq List\langle t \rangle$*
- *if $t \in \mathcal{B}$, then $LUB\langle t \rangle \leq t$*

All non-list terms have a unique type given by the typing relation *written $e : t$ for a term e with type t, where none $: \bot$. The typing relation is extended to list-terms as follows:*
- *$nil : List\langle \bot \rangle$*
- *$\langle e_1, e_2, \ldots, e_n \rangle : NEList\langle LUB\langle \top \rangle \rangle$*

We call types of the form $List\langle t\rangle$ and $NEList\langle t\rangle$ (non-empty list) for *list types*. These types have terms that are lists, i.e., ordered collections of terms. Note that we distinguish, at the type level, between empty and non-empty lists. Types of the form $LUB\langle t\rangle$ are called *LUB-types* where LUB is short for *least upper bound*. The motivation for the latter type constructor builds on the following definition.

▶ **Definition 15.** *Let $\rhd$ be the least relation between types such that whenever $t_1 \leq t_2$ then:*

- $t_1 \rhd t_2$
- $LUB\langle t_2\rangle \rhd t_1$

If $t_1 \rhd t_2$ we say that t_1 is compatible with *t_2.*

The intuition behind the compatibility relation between types is to permit the use of terms in places that are compatible with their type. We use the notion of compatible types to check *correct use of terms* and *consistent use of terms*. Correct use means that a term may only be used in places where its type is compatible with the expected type. Consistent use, which is only relevant for LUB-typed terms, means that a term is not used in multiple places that are incompatible.

LUB-types are required when the lexical form of terms is alone not sufficient to determine its type, which is typically when there are more types than different lexical forms. In these cases, one needs to examine the expected types of where the terms are used to establish if the term is used consistently. For the semantic web languages RDF and OWL, this is relevant as the IRIs and blank nodes of RDF may be used to designate different types of entities in OWL that are necessary to keep apart to ensure their correct and consistent use. For instance in OWL, object properties and datatype properties are disjoint types, yet, it is not possible to determine based on the lexical representation alone if an IRI represents an object property or a datatype property. For these terms, we only give an upper bound (LUB) of what types they can have. A term to which we assign the type $LUB\langle t\rangle$ may have a type that is a subtype of t, and may therefore be used any place where any subtype t' of t is expected. However, terms must also be used consistently, so the same term cannot then be used in a place where a type not compatible with t' is expected.

Note how we exploit LUB-types when we assign non-empty list terms the type $List\langle LUB\langle \top\rangle\rangle$. Since all list elements are type-checked, it is unnecessary to give a more specific type to the list itself. Any type violation of a non-empty list is either due to the list itself is given to an argument expecting a non-list, or that a term "inside" the list is of an incompatible type. The type assigned to the list term itself need only account for the first of these two cases. However, to type-check the terms inside lists we need to know how deeply nested a term is inside a list, and how deeply nested a given type is inside a list type. This is captured in the following definitions.

▶ **Definition 16.** *We define $\delta_{\mathcal{E}}$ to be a binary function from pairs of terms to natural numbers as follows:*

- $\delta_{\mathcal{E}}(a,a) = 0$ *for any* $a \in \mathcal{E}$
- *if* $\delta_{\mathcal{E}}(a,b) = n$, *then* $\delta_{\mathcal{E}}(a,\langle e_1,\ldots,b,\ldots,e_n\rangle) = n+1$, *for any* $a,b,e_1,\ldots,e_n \in \mathcal{E}$

If $\delta_{\mathcal{E}}(e_1,e_2) = n$, we say that e_1 occurs at depth *n* in *e_2.*

▶ **Definition 17.** *Let $\delta_{\mathcal{T}}$ be a binary function from pairs of types to natural numbers as follows:*

- $\delta_{\mathcal{T}}(t,t) = 0$, *for any* $t \in \mathcal{T}$
- $\delta_{\mathcal{T}}(t_1,t_2) = n$ *then* $\delta_{\mathcal{T}}(t_1, List\langle t_2\rangle) = n+1$ *and* $\delta_{\mathcal{T}}(t_1, NEList\langle t_2\rangle) = n+1$ *for any pair* $t_1,t_2 \in \mathcal{T}$

If $\delta_{\mathcal{T}}(t_1,t_2) = n$, we say that t_1 occurs at depth *n* in *t_2.*

As an example, in the type $NEList\langle List\langle t\rangle\rangle$, t occurs at depth 2.

The definition is used to relate depths of terms in lists to the type at corresponding depths in nested list types. However, note that for $LUB\langle t\rangle$, t must be a basic type and, e.g., not a list type. It is therefore no n such that t occurs at depth n in $LUB\langle t\rangle$, but, e.g., $LUB\langle t\rangle$ occurs at depth 1 in $NEList\langle LUB\langle P\rangle\rangle$.

We now introduce the implementation of terms and types to be used for creating knowledge graphs and ontologies in RDF and OWL.

▶ **Implementation 18.** All vocabulary terms defined in the OTTR framework use the following namespace, unless otherwise noted:

```
@prefix ottr:    <http://ns.ottr.xyz/0.4/> .
```

▶ **Implementation 19.** Let $\mathcal{E}$ be the set of all valid RDF terms, i.e., IRIs, literals and blank nodes [27]. Variables are designated by blank nodes, so let $\mathcal{V}$ be an infinite set of blank nodes. All IRI terms have type $LUB\langle$`ottr:IRI`$\rangle$, all non-list blank nodes have type $LUB\langle$`rdfs:Resource`$\rangle$, and all literals have a type equal to their specified datatype or `xsd:string` if no datatype is given. The term `rdf:nil` denotes *nil* and has the type *List*$\langle$`rdfs:Resource`$\rangle$. All other RDF lists denote the corresponding list term and have the type *NEList*$\langle LUB\langle$`rdfs:Resource`$\rangle\rangle$.

▶ **Syntax 20.** Terms in stOTTR share the same syntax as terms in Turtle [2], both for IRIs, blank nodes, literals and lists, except that lists are written surrounded with parenthesis with elements *separated by commas*. stOTTR also adopts Turtle's syntax for defining prefixes. Variable terms are written using Turtle's syntax for blank node labels, prefixed by a question mark. We may write `none` for the term *none*, and `()` for the empty list *nil*. stOTTR is space-insensitive.

▶ **Implementation 21.** All basic types are listed in Table 1. All of these, except those prefixed by `ottr:` are IRIs taken from the RDF, RDFS, OWL and XSD standards. `ottr:Bot` denotes $\bot$, whereas `rdfs:Resource` denotes $\top$. The types are presented with a description taken from the respective standards, and possibly given a supertype that follows this description and which forms the basis of determining compatibility between types. The type hierarchy is published at Zenodo: `https://zenodo.org/records/12607216`.

▶ **Syntax 22.** Basic types are denoted by their IRI as defined above, using the syntax for IRIs from Turtle. For complex types, we write `LUB<t>`, `List<t>` and `NEList<t>`, where `t` is a type.

▶ **Example 23.** The term `"3"^^xsd:int` has type `xsd:int`, and since `xsd:int` is a subtype of `xsd:long`, and `xsd:long` is a subtype of `xsd:integer`, we have that `xsd:int` is compatible with `xsd:integer` and can use `"3"^^xsd:int` where a value of type `xsd:integer` is expected.

The term `ex:mary` is an IRI, and therefore has type `LUB<ottr:IRI>`. Since `owl:Named-Individual` is a subtype of `ottr:IRI`, we have that `LUB<ottr:IRI>` is compatible with `owl:NamedIndividual`, and can therefore use the term `ex:mary` where a term of type `owl:NamedIndividual` is expected.

The following illustrates an interesting feature of OTTR's type system. Some types of the OWL ontology language are defined to be disjoint, such as OWL object properties and datatype properties, and should raise an error in the case that an IRI is assigned multiple such types. Other cases of assigning multiple types to the same IRI can result in what is called *punning*, e.g., stating that `Eagle` is both a `owl:NamedIndividual` and a `owl:Class`, which is permissible in OWL, but may not always be desirable. The type hierarchy presented in Table 1 above does not permit punning, as there is no subtype of, e.g., `owl:NamedIndividual` and `owl:Class` that is different from $\bot$. However, it is easy to extend the type hierarchy with types to allow for punning. Table 2 lists the necessary extensions of types to allow for punning according to the OWL standard. This example is further developed in Example 53 on page 26.

Table 1 The basic types of the OTTR type system.

Type	Supertype	Description
`rdfs:Resource`		All things described by RDF
`ottr:Bot`		Empty type
`ottr:IRI`	`rdfs:Resource`	An IRI (Internationalized Resource Identifier)
`owl:Class`	`ottr:IRI`	OWL Classes (understood as sets of individuals)
`owl:NamedIndividual`	`ottr:IRI`	Individuals in OWL 2
`owl:ObjectProperty`	`ottr:IRI`	Properties connecting pairs of individuals
`owl:DatatypeProperty`	`ottr:IRI`	Properties connecting individuals with literals
`owl:AnnotationProperty`	`ottr:IRI`	Properties used to provide an annotation for an ontology, axiom, or an IRI
`rdfs:Datatype`	`ottr:IRI`	Data values
`rdfs:Literal`	`rdfs:Resource`	Literal values such as strings and integers
`ottr:string`	`rdfs:Literal`	Character strings with or without language tag
`xsd:string`	`ottr:string`	Character strings
`xsd:normalizedString`	`xsd:string`	Whitespace-normalized strings
`xsd:token`	`xsd:normalizedString`	Tokenized strings
`xsd:language`	`xsd:token`	Language tags per [BCP47]
`rdf:langString`	`ottr:string`	Character strings with language tag
`xsd:Name`	`xsd:token`	XML Names
`xsd:NCName`	`xsd:Name`	XML NCNames
`xsd:NMTOKEN`	`xsd:Name`	XML NMTOKENs
`owl:real`	`rdfs:Literal`	All real numbers
`owl:rational`	`owl:real`	All rational numbers
`xsd:decimal`	`owl:rational`	Arbitrary-precision decimal numbers
`xsd:integer`	`xsd:decimal`	Arbitrary-size integer numbers
`xsd:long`	`xsd:integer`	64 bit signed integers
`xsd:int`	`xsd:long`	32 bit signed integers
`xsd:short`	`xsd:int`	16 bit signed integers
`xsd:byte`	`xsd:short`	8 bit signed integers
`xsd:nonNegativeInteger`	`xsd:integer`	Integer numbers ≥ 0
`xsd:positiveInteger`	`xsd:nonNegativeInteger`	Integer numbers > 0
`xsd:unsignedLong`	`xsd:positiveInteger`	64 bit unsigned integer
`xsd:unsignedInt`	`xsd:unsignedLong`	32 bit unsigned integer
`xsd:unsignedShort`	`xsd:unsignedInt`	16 bit unsigned integer
`xsd:unsignedByte`	`xsd:unsignedShort`	8 bit unsigned integer
`xsd:nonPositiveInteger`	`xsd:integer`	Integer numbers ≤ 0
`xsd:negativeInteger`	`xsd:nonPositiveInteger`	Integer numbers < 0
`xsd:double`	`rdfs:Literal`	64-bit floating point numbers incl. +-`Inf`, +-0, `NaN`
`xsd:float`	`rdfs:Literal`	32-bit floating point numbers incl. +-`Inf`, +-0, `NaN`
`xsd:date`	`rdfs:Literal`	Dates (`yyyy-mm-dd`) with or without timezone
`xsd:dateTime`	`rdfs:Literal`	Date and time with or without timezone
`xsd:dateTimeStamp`	`xsd:dateTime`	Date and time with timezone
`xsd:time`	`rdfs:Literal`	Times (`hh:mm:ss.sss...`) with or without timezone
`xsd:gYear`	`rdfs:Literal`	Gregorian calendar year
`xsd:gMonth`	`rdfs:Literal`	Gregorian calendar month
`xsd:gDay`	`rdfs:Literal`	Gregorian calendar day of the month
`xsd:gYearMonth`	`rdfs:Literal`	Gregorian calendar year and month
`xsd:gMonthDay`	`rdfs:Literal`	Gregorian calendar month and day
`xsd:duration`	`rdfs:Literal`	Duration of time
`xsd:yearMonthDuration`	`xsd:duration`	Duration of time (months and years only)
`xsd:dayTimeDuration`	`xsd:duration`	Duration of time (days, hours, minutes, seconds only)
`xsd:hexBinary`	`rdfs:Literal`	Hex-encoded binary data
`xsd:base64Binary`	`rdfs:Literal`	Base64-encoded binary data
`xsd:boolean`	`rdfs:Literal`	`true`, `false`
`xsd:anyURI`	`rdfs:Literal`	Absolute or relative URIs and IRIs
`rdf:HTML`	`rdfs:Literal`	HTML content
`rdf:XMLLiteral`	`rdfs:Literal`	XML content

Table 2 Examples of types that would allow punning.

Type	Supertypes
`:Punned-Class-NamedIndividual`	`owl:Class`, `owl:NamedIndividual`
`:Punned-Class-ObjectProperty`	`owl:Class`, `owl:ObjectProperty`
`:Punned-Class-DatatypeProperty`	`owl:Class`, `owl:DatatypeProperty`
`:Punned-Class-AnnotationProperty`	`owl:Class`, `owl:AnnotationProperty`
`:Punned-Datatype-NamedIndividual`	`rdfs:Datatype`, `owl:NamedIndividual`
`:Punned-Datatype-ObjectProperty`	`rdfs:Datatype`, `owl:ObjectProperty`
`:Punned-Datatype-DatatypeProperty`	`rdfs:Datatype`, `owl:DatatypeProperty`
`:Punned-Datatype-AnnotationProperty`	`rdfs:Datatype`, `owl:AnnotationProperty`
`:Punned-NamedIndividual-ObjectProperty`	`owl:NamedIndividual`, `owl:ObjectProperty`
`:Punned-NamedIndividual-DatatypeProperty`	`owl:NamedIndividual`, `owl:DatatypeProperty`
`:Punned-NamedIndividual-AnnotationProperty`	`owl:NamedIndividual`, `owl:AnnotationProperty`
`:Punned-Class-NamedIndividual-ObjectProperty`	`owl:Class`, `owl:NamedIndividual`, `owl:ObjectProperty`
`:Punned-Class-NamedIndividual-DatatypeProperty`	`owl:Class`, `owl:NamedIndividual`, `owl:DatatypeProperty`
`:Punned-Class-NamedIndividual-AnnotationProperty`	`owl:Class`, `owl:NamedIndividual`, `owl:AnnotationProperty`
`:Punned-Datatype-NamedIndividual-ObjectProperty`	`rdfs:Datatype`, `owl:NamedIndividual`, `owl:ObjectProperty`
`:Punned-Datatype-NamedIndividual-DatatypeProperty`	`rdfs:Datatype`, `owl:NamedIndividual`, `owl:DatatypeProperty`
`:Punned-Datatype-NamedIndividual-AnnotationProperty`	`rdfs:Datatype`, `owl:NamedIndividual`, `owl:AnnotationProperty`

3.2 Template Instances

Statements in OTTR are expressed by *template instances*. Before we can introduce these, we need a couple of utility definitions specific to the use of lists in statements. OTTR has special support for lists in the form of list expanders, which are functions that allow a single statement to expand to multiple statements by replacing a list of terms with the elements of the list, in different ways.

▶ **Definition 24.** *A* list expander *is a function from a list of list terms to a set of lists of terms.*

▶ **Implementation 25.** We define the following list expanders:

$$\begin{aligned}
id(\langle l_1,\dots,l_n\rangle) &= \{\langle l_1,\dots,l_n\rangle\}\\
cross(\langle l_1,\dots,l_n\rangle) &= \{\langle e_1,\dots,e_n\rangle \mid e_i \in l_i\}\\
zipMin(\langle l_1,\dots,l_n\rangle) &= \{\langle l_1(i),\dots,l_n(i)\rangle \mid 1 \le i \le \min_{k\le n} length(l_k)\}\\
zipMax(\langle l_1,\dots,l_n\rangle) &= \{\langle l_1(i),\dots,l_n(i)\rangle \mid 1 \le i \le \max_{k\le n} length(l_k)\}
\end{aligned}$$

That is, *id* is the identity function, *cross* is the cross product of its argument lists, *zipMin* is the convolution restricted to the shortest list, and *zipMax* is the convolution where all lists are made of equal length by padding *none*-terms at the end (remember that $L(i) = none$ if $i > length(L)$).

▶ **Example 26.** This shows the behaviour of the list expanders on the same input.

$$\begin{aligned} cross(\langle 1,2,3\rangle, \langle 4,5\rangle) &= \{\langle 1,4\rangle, \langle 2,4\rangle, \langle 3,4\rangle, \langle 1,5\rangle, \langle 2,5\rangle, \langle 3,5\rangle\} \\ zipMin(\langle 1,2,3\rangle, \langle 4,5\rangle) &= \{\langle 1,4\rangle, \langle 2,5\rangle\} \\ zipMax(\langle 1,2,3\rangle, \langle 4,5\rangle) &= \{\langle 1,4\rangle, \langle 2,5\rangle, \langle 3, none\rangle\} \end{aligned}$$

▶ **Definition 27.** *Let $\mathcal{L}$ be a set of list expanders that contains at least cross, zipMin and zipMax.*

The list expanders defined above will be used in the definition of instance expansion, Section 3.4. With this, we are ready to define the notion of template instance.

▶ **Definition 28.** *A* template instance *(or just* instance*) is a 4-tuple (t, A, E, e) of*

- *a constant term t, called the instance's template name,*
- *a list of terms A called the* instance's arguments,
- *a set of indices E, denoting which arguments to apply a list expander to,*
- *and a list expander e.*

The arity of an instance *is the size of its argument list. A* ground template instance *is a template instance where the value of every argument is a constant.*

A template instance can be viewed as a *call* to a template. A template is a definition of a pattern of statements, and a template instance denotes one instance of the template's pattern.

▶ **Syntax 29.** Instances have the form $t(a_1, \ldots, a_n)$. where t is a template name in the form of an IRI, and each a_i is an argument term. List expanders are written before the template name followed by a | (where the *id*-expander is always omitted), with the argument terms to expand marked with ++. Examples:

Structure	**Syntax**
$(t, \langle a_1, \ldots, a_n\rangle, \emptyset, id)$	`t(a_1, ..., a_n) .`
$(t, \langle a_1, a_2, a_3\rangle, \{1\}, cross)$	`cross | t(++a_1, a_2, a_3) .`
$(t, \langle a_1, a_2, a_3\rangle, \{1,3\}, zipMin)$	`zipMin | t(++a_1, a_2, ++a_3) .`

▶ **Example 30.** The following are examples of written instances:

```
ex:Person(ex:mary, "Mary Smith", "1980-02-03"^^xsd:date) .

ex:Person(ex:peter, "Peter Smith", "1984-10-01"^^xsd:date) .

ex:Person(ex:bob, "Bob Green", none) .

cross | ex:HasFamilyRelation(
  ++(ex:peter, ex:mary),
  ++(ex:carl, ex:nora),
  ex:parentOf) .

zipMax | ex:HasFamilyRelation(
  ++(ex:eric, ex:hannah, ex:bob),
  ex:peter,
  ++(ex:father, ex:mother)) .
```

The corresponding templates to these instances are defined in Example 44 and their corresponding RDF statements can be seen in Example 48. The first three instances each describe a person, where the first argument is the person's IRI, the second argument is the person's name, and the final argument is the person's birthdate. Note that for the final instance, there is no value (i.e., *none*) given for the birthdate, e.g., the birthdate is unknown. The fourth instance uses the *cross* list-expander to create `ex:HasFamilyRelation` instances with the `ex:parentOf` property, for all combinations of elements from the two lists, thus stating that `ex:peter` and `ex:mary` are the parents of `ex:carl` and `ex:nora`. The final instance also instantiates `ex:HasFamilyRelation`, but uses a *zipMax* list expander to pair people with their relation to `ex:peter`, thus making `ex:eric` the `ex:father` of `ex:peter`, whereas `ex:hannah` is his `ex:mother`. For `ex:bob`, there is no given property, i.e., the value is *none*, and its up to the definition of the template, whether this parameter is optional or not, how this is handled.

3.3 Templates

A *template* is a parameterized set of statements – which themselves are template instances. Thus, templates are a recursive structure where a template is defined in terms of other templates. Base templates are the exception, and are used to represent basic statements in a different data representation language.

Before we can define our notion of templates, we need to establish some preliminary definitions.

▶ **Definition 31.** *Let $\mathcal{M}$ be a set of tokens, called modifiers, that contains at least the token optional, denoting an optional value.*

The *optional* modifier is used to control how the *none* term behaves during expansion. This is defined in Section 3.4. However, the intuition is that we specify a parameter as *optional* if *none* is a meaningful argument, and omit *optional* when it is not. All statements with a *none* as arguments to a non-*optional* are discarded. This allows templates to contain subpatterns that are only used when specific values are present (i.e., not *none*).

▶ **Implementation 32.** We extend $\mathcal{M}$ with an additional modifier called *nonBlank*, that specifies that the value is not a blank node. Its behaviour is defined in Implementation 55.

Recall that our implemented terms contain blank nodes, and that blank nodes are, according to the RDF specification [27], not permitted as predicates in RDF triples. Blank nodes can also be undesirable in certain other settings, for example, if concrete values are required to be meaningful for their intended use. The *nonBlank* modifier is introduced to control where blank nodes are permitted and not. That is, a blank node is not allowed as an argument to a parameter marked with *nonBlank*.

▶ **Definition 33.** *A* parameter *is a 4-tuple (v, t, d, M) consisting of:*
- *a variable term v different from none, called the* parameter variable
- *a type t*
- *a (possibly none) constant term d, called the parameter's* default value
- *a (possibly empty) set of* modifiers *M.*

In the above definition, we use *none* to denote that a parameter does not have a default value, and say that a parameter *does not have a default value* if the parameter's default value is *none*.

▶ **Syntax 34.** A parameter is written with modifiers first, where a question mark denotes *optional* and an exclamation mark denotes *nonBlank*, and nothing is written for the empty set of modifiers. Following this comes the parameter's type. We can omit writing the type `rdfs:Resource`. Then follows the parameter's variable. Finally, if the default value is not *none*, the value is written at the end separated from the variable with an equals sign. Examples:

Structure	**Syntax**
$(v, t, none, \emptyset)$	`t ?v`
$(v, t, none, \{optional\})$	`? t ?v`
$(v, t, d, \{optional, nonBlank\})$	`!? t ?v=d`

▶ **Definition 35.** *A* template signature *(or just* signature*) is a triple* (t, P, N) *of*

- *a constant term* t*, called the signature's* template name*,*
- *a list of parameters* P *such that all parameter variables in a signature's list of parameters are different,*
- *and a set of* annotations N*, which is a set of ground template instances; we call these* annotation instances*.*

The arity of a signature *is the size of its parameter list. The* type of a variable *is the type of its parameter within the signature's parameter list.*

As we shall see, a signature is part of the definition of a template. However, a signature is also meaningful on its own, as documentation of how to use a template, similar to function and method signatures in programming languages.

A signature may contain a set of ground instances called annotations. These are meant to be used for documenting the signature (similar to Javadoc in Java or Docstrings in Python), such as who created the template, the version of the template, and a description of the template pattern.

▶ **Syntax 36.** A signature is written starting with the template name, followed by the list of parameters enclosed in square braces. Annotations, if any, are listed after this, separated by commas and prefixed with `@@`. Examples:

Structure	**Syntax**
$(t, P, \emptyset)$	`t[P] .`
$(t, P, \{(i_1, a_1), (i_2, a_2)\})$	`t[P] @@i_1(a_1), @@i_2(a_2) .`

A signature ends with a dot.

▶ **Definition 37.** *A* base template *is a pair* $(S, base)$ *of a* template signature *and the token base.*

A base template denotes a parameterized basic statement that cannot be broken down into a set of smaller (parameterized) statements. Base template instances can either be used directly in an OTTR serialization, or, more commonly, be transformed into statements in a different language and serialization format.

▶ **Syntax 38.** A base template is written similarly to a template, except that the pattern is replaced with the `BASE` keyword, that is:

Structure	**Syntax**
$(S, base)$	`S :: BASE .`

▶ **Implementation 39.** Our implementation contains one base template that denotes an RDF triple:

$$\begin{aligned}&((\texttt{ottr:Triple},\\ &\quad\langle(\texttt{subject}, \texttt{ottr:IRI}, \emptyset),\\ &\quad\;(\texttt{predicate}, \texttt{ottr:IRI}, \{nonBlank\}),\\ &\quad\;(\texttt{object}, \texttt{rdfs:Resource}, \emptyset)\rangle, \emptyset),\\ &\;base)\end{aligned}$$

or, equivalently in stOTTR format:

```
ottr:Triple [
    ottr:IRI ?subject,
    ! ottr:IRI ?predicate,
    rdfs:Resource ?object
] :: BASE .
```

One can imagine implementations supporting other base templates: Base templates for RDF quadruples, OWL expressions, rows in tabular files, or SQL `INSERT` statements.

We are now ready to define the central concept of a template.

▶ **Definition 40.** *A* template *is pair (S, B) of a* template signature *S and a set of template instances B called the template's* pattern*; we call these* pattern instances*.*

A template is the core construct in OTTR, and is the primary means of abstraction. Using templates, we can create complex parameterized statements that are easy to reuse. A template can either be defined in terms of base templates directly, or by instantiating other templates (or a combination). Taking a bottom-up approach, this supports layers of abstractions, each layer creating more complex statements that are closer to the terminology of that of a concrete domain to be modelled. To use a template, all one needs to know is its signature. The signature states the arguments a user must provide, and may also contain annotations that further describe the intended use of the template.

▶ **Syntax 41.** A template is written with the signature first (as described above) except the final dot, followed by `::`, and then the pattern instances separated by comma an enclosed in curly braces, and finally ends with a dot. Examples:

Structure	**Syntax**
$(S, \emptyset)$	`S ::  .`
$(S, \{i_1, i_2\})$	`S :: i_1, i_2 .`

▶ **Definition 42.** *Let $\mathbb{S}$ be the set of all signatures, $\mathbb{B}$ be the set of all base templates, and $\mathbb{T}$ be the set of all templates. Let $\mathbb{O} = \mathbb{S} \cup \mathbb{B} \cup \mathbb{T}$, and let the elements of $\mathbb{O}$ be called* template objects*.*

Furthermore, let σ be a function from sets of template objects to sets of signatures, such that $\sigma(O)$ is the set of all template signatures contained either directly in O, or within a template or base template in O.

▶ **Definition 43.** *We say that a template instance I is the* instance of *a template signature T if T has the same template name as I. For an argument a in instance I of signature T, we say that its* corresponding parameter of *T is the parameter with the same index in the parameter list as the index of a in the argument list of I.*

▶ **Example 44.** The templates used in Example 30 are defined as follows:

```
ex:Person[ owl:NamedIndividual ?p, xsd:string ?name, ? xsd:date ?born ]
    @@ottr:Triple(ex:person, ex:madeBy, ex:leifhka),
    @@ottr:Triple(ex:person, rdfs:label, "Person Template")
:: {
    o-rdf:Type(?p, ex:Person),
    ottr:Triple(?p, ex:hasName, ?name),
    ottr:Triple(?p, ex:born, ?born)
} .
```

```
ex:HasFamilyRelation [
    owl:NamedIndividual ?p1,
    owl:NamedIndividual ?p2,
    ! owl:ObjectProperty ?r=ex:isFamilyRelatedTo
] :: {
    ottr:Triple(?p1, ?r, ?p2)
} .
```

Note that we have given the property parameter a default value, so if *none* is given as argument, the default value `ex:isFamilyRelatedTo` is used. The `?born` parameter is specified as optional, hence a missing birthdate will still create a person with an IRI and name, but no birthdate.

One can now use these templates to register complete families, where input is given as lists of IRIs and names for parents and children per family. We can capture both the creation of the persons and their relations with a single template as follows:

```
ex:NuclearFamily[
    List<owl:NamedIndividual> ?parents,
    List<xsd:string> ?parentNames,
    List<owl:NamedIndividual> ?children,
    List<xsd:string> ?childrenNames
] :: {
    zipMax | ex:Person(++?parents, ++?parentNames, none),
    zipMax | ex:Person(++?children, ++?childrenNames, none),
    cross | ex:HasFamilyRelation(++?parents, ++?children, ex:parentOf)
}
```

As we assume that input does not contain any dates of birth, the `ex:NuclearFamily` template uses a *none* value as argument for the corresponding parameter in the `ex:Person` template.

3.4 Instance Expansion

We have now defined the core constructs in the OTTR framework, and will proceed to define the process of *instance expansion*, which is to iteratively transform instances into ultimately instances of base templates only.

We treat list expansion separately first, as this is technically the most complex part of the expansion process. List expansion is specified using two functions, where the first selects the lists to expand from the instance and expands them using the given list expander function, while the second creates one instance per element in the result of this function application. The full list expansion is the composition of these two functions.

▶ **Definition 45.** *Let $I = (t, A, E, e)$ be an instance of arity m with list expander indices $E = \{i_1, \ldots, i_n\}$ where $i_k < i_{k+1}$. Define the function ϵ_1 from instances to set of argument lists as follows:*

$$\epsilon_1((t, A, \{i_1, \ldots, i_n\}, e)) = e(\langle A(i_1), \ldots, A(i_n)\rangle)$$

Here we use $A(i_k)$ to denote the i_k'th element of A. Furthermore, let

$$\epsilon_2(A, L, E, i) = \begin{cases} L(E'(i)), & \textit{if } i \in E \\ A(i), & \textit{otherwise} \end{cases}$$

where $E'(i)$ is the position of i in E in ascending order. Finally, let

$$\epsilon((t, A, E, e)) = \{t(\epsilon_2(A, L, E, 1), \ldots, \epsilon_2(A, L, E, m)) \mid L \in \epsilon_1(t, A, E, e)\}$$

The function ϵ takes an instance and produces a set of instances by first selecting the argument lists that are to be expanded and applies the list expander function (with ϵ_1), and then creates new instances based on the expanded lists by combining elements of the expansion with the original non-expanded values of the argument instance (with ϵ_2).

▶ **Definition 46.** *The* direct expansion $\delta(I)$ *of a ground instance* $I = (t, A, E, e)$*, where* t *corresponds to the template object* T*, is defined as follows:*

1. *if* $E \neq \emptyset$*, then the direct expansion of* I *is* $\epsilon(I)$*.*
2. *if there is an* i *such that* $a_i = none$ *and its corresponding parameter is not optional and has no default value, then* $\delta(I) = \emptyset$*.*
3. *if* T *is a base template or a signature (and not a template), then* $\delta(I) = \{I\}$*.*
4. *otherwise, let* $T = (S, B)$ *and build the* induced substitution σ of T and I *by considering each argument* a_i *of* I *and its corresponding parameter* P_i *with variable* x_i *in* T*:*
   - *if* a_i *has value none and* p_i *has a default value* d*, then* $\sigma := \sigma \cup \{x_i/d\}$
   - *otherwise,* $\sigma := \sigma \cup \{x_i/a_i\}$

   Then let $\delta(I) = B\sigma$*, that is,* σ *applied to the pattern* B *of* T*.*

In the above definition, the first case is performing the list expansion defined in the previous definition. The second case handles *none* values, where *none* values given to non-optional parameters (without default value) result in an empty expansion, i.e., the instance is discarded, and *none* values given to parameters with a default value are replaced with that default value. The third case states that the expansion of a base template (or a signature, i.e., the case where we do not have the full definition of a template object) is just the base template itself (however, note that step comes after the former two, so these steps apply first). The final case is the replacement of an instance to a template with the pattern the template denotes, where parameter values are substituted with argument values.

This denotes a single step in the expansion, the full expansion of an instance is simply the fix-point of this process.

▶ **Definition 47.** *The* expansion of a set of ground instances $\mathcal{I}$ *is the fix-point of the following function:*

$$\eta(\mathcal{I}) = \bigcup \{\delta(I) \mid I \in \mathcal{I}\}$$

Example 48 gives an example of the expansion process of the previously exemplified instances and templates.

▶ **Example 48.** The example demonstrates the expansion of selected instances from Example 30. For the two first examples, we show the step-wise expansion process.

The following instance:

```
ex:Person(ex:mary, "Mary Smith", "1980-02-03"^^xsd:date) .
```

. . . expands in one step to:

```
o-rdf:Type(ex:mary, ex:Person),
ottr:Triple(ex:mary, ex:hasName, "Mary Smith"),
ottr:Triple(ex:mary, ex:born, "1980-02-03"^^xsd:date)
```

. . . which expands in one step to:

```
ottr:Triple(ex:mary, rdf:type, ex:Person),
ottr:Triple(ex:mary, ex:hasName, "Mary Smith"),
ottr:Triple(ex:mary, ex:born, "1980-02-03"^^xsd:date)
```

. . . which is equivalent to the following RDF graph:

```
ex:mary rdf:type ex:Person ;
        ex:hasName "Mary Smith" ;
        ex:born "1980-02-03"^^xsd:date .
```

The following instance:

```
ex:Person(ex:bob, "Bob Green", none) .
```

. . . expands in one step to:

```
o-rdf:Type(ex:bob, ex:Person),
ottr:Triple(ex:bob, ex:hasName, "Bob Green"),
ottr:Triple(ex:bob, ex:born, none)
```

. . . which expands in one step to:

```
ottr:Triple(ex:bob, rdf:type, ex:Person),
ottr:Triple(ex:bob, ex:hasName, "Bob Green")
```

. . . which is equivalent to the following RDF graph:

```
ex:bob rdf:type ex:Person ;
       ex:hasName "Bob Green" .
```

The following instance:

```
cross | ex:HasFamilyRelation(
  ++(ex:peter, ex:mary),
  ++(ex:carl, ex:nora),
  ex:parentOf) .
```

. . . expands to the following RDF graph:

```
ex:peter ex:parentOf ex:carl, ex:nora .
ex:mary ex:parentOf ex:carl, ex:nora .
```

The following instance:

```
zipMax | ex:HasFamilyRelation(
  ++(ex:eric, ex:hannah, ex:bob),
  ex:peter,
  ++(ex:fatherOf, ex:motherOf)) .
```

. . . expands to the following RDF graph:

```
ex:eric ex:fatherOf ex:peter .
ex:hannah ex:motherOf ex:peter .
ex:bob ex:isFamilyRelatedTo ex:peter .
```

Finally, we define the process of annotation expansion.

▶ **Definition 49.** *The* annotation expansion of a template signature *is the result of replacing the annotation instances of the template signature with their expansion.*

3.5 Template Library and Dataset

In this section, we will define what it means for a set of template objects and instances to be correct, e.g., with respect to the type system and template signature specifications. We start by defining the notions of template library and dataset.

▶ **Definition 50.** *A* template library *is a set of template objects. A* template dataset *is a pair* $(\mathcal{L}, \mathcal{I})$ *of a template library* $\mathcal{L}$ *and a set of ground template instances* $\mathcal{I}$.

▶ **Definition 51.** *For a term* v *occurring in an instance* $(t, \langle a_1, \ldots, a_n \rangle, E, e)$*, we say that* v *has* inferred type p *if* v *is a term in an argument* a_i *at depth* n *and either:*

- $i \notin E$*, with a corresponding parameter with a type having the type* p *at depth* n
- $i \in E$*, with a corresponding parameter with a type having* p *at depth* $n - 1$

The inferred type of a term is the type the term is *used as*. A term may therefore have many inferred types, one for each time the term occurs in any instance.

▶ **Definition 52.** *A term* v *is* consistently typed *in a set of instances if there exists a type* p *unequal to* $\perp$ *such that*

- p *is a subtype of all inferred types of* v*, and*
- *the type of* v *is compatible with* p.

In other words, a term is consistently typed if there is a type one can assign it that is a subtype of all of its inferred types and that is compatible with the actual type of the term. Note that this definition covers both the consistent use of terms and correct typing as discussed above. For example, if a term v is used both as an `xsd:int` and as a `xsd:string`, this is a case of inconsistent use of the term v, as there is no subtype for these inferred types (unequal to $\perp$), which violates the first point. If the term v has type `xsd:int` but is used as a `xsd:string`, then it is a case of incorrect typing and a violation of the second point of the definition.

▶ **Example 53.** Assume the IRI `ex:Eagle` is used both as a `owl:NamedIndividual` and a `owl:Class`. Under the type hierarchy given in Table 1 there exists no subtype of these types, hence `ex:Eagle` *is not* consistently typed. Under the type hierarchy given in Table 2 there exists a subtype of these types, `:Punned-Class-NamedIndividual`, hence `ex:Eagle` *is* consistently typed.

▶ **Definition 54.** *An instance is* modifier correct *if all of its arguments satisfy the corresponding parameter modifiers.*

▶ **Implementation 55.** Any non-blank constant and any variable of a parameter marked with *nonBlank* satisfies the *nonBlank* modifier.

Note that the definition above ensures that the *nonBlank* modifier is propagated upwards in the dependency graph of templates so that any variable used as an argument to a *nonBlank*-parameter must be marked as *nonBlank*.

▶ **Definition 56.** *A set of instances is* consistently typed *(*modifier correct*) if every term occurring in it is consistently typed (modifier correct). A template library is* consistently typed *(*modifier correct*) if the set of all instances occurring in it is consistently typed (*modifier correct*).*

This covers the correct use of terms.

We now define correctness of the interplay between template objects and between templates and instances by way of several properties that combined form the notion of correctness.

▶ **Definition 57.** *A template* T directly depends *on a template object* S *if* T *has a pattern that contains an instance of* S*. A template library is* acyclic *if the directly depends relation is* acyclic.

Acyclicity ensures that instance expansion terminates and is finite. Note that this also disallows recursively defined templates. However, we have no means of manipulating or producing new terms apart from through list expansion. Under the current type system, we are unable to define a template that can apply a list expander to an instance of itself, as this would not be consistently typed. Thus, a recursive call within a template's pattern can only reuse the same arguments it was originally given or have constants as arguments, thus creating an infinite loop in the expansion.

▶ **Definition 58.** *A set of instances $\mathcal{I}$ has* referential integrity *with respect to a template library $\mathcal{L}$ if every instance has a name corresponding to a template signature in $\sigma(L)$, and that the arity of the instance equals the arity of the corresponding template signature.*

A template library has referential integrity *if no two non-signature template objects have the same name and the set of all instances occurring in it has referential integrity with respect to it.*

Referential integrity ensures that all instances refer to a unique template object, and that the number of arguments equals the number of parameters in the corresponding signature.

▶ **Definition 59.** *A template object is* well-founded *if it is a base template or if it is a template that depends only on well-founded templates. A template library is* well-founded *if it contains only well-founded templates.*

Well-foundedness is a property that ensures that all templates are properly defined, that is, there are no templates that depend on a template object that is a signature only. It characterizes the fact that instances can be expanded all the way to instances of base templates only. Note that well-foundedness is not the same as acyclicity. A template that depends on a template object which is a signature is non-well-founded but acyclic, while a template that directly depends on a base template and itself (recursively) is well-founded and cyclic.

▶ **Definition 60.** *A* semi-valid template library *is a template library that is consistently typed, modifier correct, acyclic, and has referential integrity.*

▶ **Definition 61.** *A* valid template library *is a semi-valid template library that is well-founded. A* valid template dataset *is a template dataset where its template library is valid, and its set of instances is consistently typed and has referential integrity with respect to the template library.*

The difference between a semi-valid and a valid library is whether all template objects are properly defined or not.

▶ **Example 62.** Below are examples of violations of correctness of instances and templates as defined above, based on the templates from Example 44.

```
ex:Person(ex:bob, "Bob Green") .
ex:Person(_:bob, ex:bob_green, none) .
ex:Person(_:b, _:b, none) .
ex:HasFamilyRelation(ex:bob, ex:mary, _:someProp) .
```

The errors in the above instances are:
1. The instance has two arguments, but the signature requires three.
2. The IRI `ex:bob_green` is given as an argument to a parameter of the incompatible type `xsd:string`.
3. The blank node `_:b` is used inconsistently; it is used as an argument to two parameters with the types `owl:NamedIndividual` and a `xsd:string` that have no common subtype unequal to $\bot$.
4. The blank node `_:someProp` is used as an argument to a parameter with a nonBlank modifier.

```
ex:ErrTemplate1 [ owl:ObjectProperty ?r ] :: {
  ex:HasFamilyRelation(ex:bob, ex:mary, ?r) } .

ex:ErrTemplate2 [ ottr:IRI ?p ] :: {
  ex:Person(?p, "Mr. P", none) } .

ex:ErrTemplate3 [ owl:NamedIndividual ?p, xsd:string ?n ] :: {
  ex:MakePerson(?p, ?n) } .

ex:ErrTemplate4 [ ottr:IRI ?p ] :: {
  ex:ErrTemplate4(?p) } .
```

The errors in the above templates are:

- In `ex:ErrTemplate1`, the parameter `?r` has no nonBlank modifier and is used as argument to a template parameter with a nonBlank modifier.
- In `ex:ErrTemplate2`, the parameter `?p` has the type `ottr:IRI` and is used as an argument to a template parameter with the incompatible type `owl:NamedIndividual`.
- `ex:ErrTemplate3` depends on an undefined template `ex:MakePerson`.
- `ex:ErrTemplate4` has a cyclic definition.

4 Serialization Formats

The OTTR framework offers two serialization formats for representing templates and instances, a special-purpose format called stOTTR, and an RDF-based format specified using the wOTTR vocabulary.

4.1 stOTTR: Terse OTTR Syntax

The stOTTR serialization format is designed to be a terse and easy to read and write syntax for representing OTTR templates and instances following the abstract model and syntax as defined in Section 3. The stOTTR grammar takes the Turtle RDF grammar [2] as starting point and expands this to support expressing templates and instances. Formally, stOTTR is specified in Antlr[6] Extended Backus-Naur form (EBNF) that extends relevant parts of the formal Turtle grammar which is used for the representation of terms, i.e., IRIs, blank nodes and literals. The stOTTR grammar specification is developed in GitLab,[7] and published at `ottr.xyz`[8] and Zenodo.[9] Figure 5 on page 7 demonstrates the stOTTR format on the `o-p:NamedPizza` template.

4.2 wOTTR: RDF Vocabulary

wOTTR is an RDF vocabulary for expressing OTTR templates and instances in an RDF format. The motivation for an RDF-based serialization format for OTTR is to support a development and management environment for OTTR based only on semantic web standards, using, e.g., triple stores, SPARQL, OWL, and rule languages to manipulate and manage templates and their instances. The vocabulary is designed to result in a compact and readable representation of templates and instances in Turtle format exploiting in particular Turtle's compact RDF list

[6] `https://www.antlr.org/`
[7] `https://gitlab.com/ottr/spec/stOTTR`
[8] `https://spec.ottr.xyz/stOTTR/0.1.4/`
[9] `https://zenodo.org/records/12568905`

```
1  o-p:NamedPizza rdf:type ottr:Template ;
2    ottr:parameters
3      ( [ ottr:type owl:Class ; ottr:variable _:pizza ]
4        [ ottr:modifier ottr:optional ; ottr:type owl:NamedIndividual ;
5          ottr:variable _:country ]
6        [ ottr:type ( ottr:NEList owl:Class ) ; ottr:variable _:toppings ] ) ;
7    ottr:pattern
8      [ ottr:of o-owl-ax:SubObjectHasValue ;
9        ottr:values ( _:pizza pz:hasCountryOfOrigin _:country ) ] ,
10     [ ottr:of o-owl-ax:SubObjectSomeValuesFrom ;
11       ottr:modifier ottr:cross ;
12       ottr:arguments
13         ( [ ottr:value _:pizza ]
14           [ ottr:value pz:hasTopping ]
15           [ ottr:modifier ottr:listExpand ; ottr:value _:toppings ] ) ] ,
16     [ ottr:of o-owl-ax:SubClassOf ;
17       ottr:values ( _:pizza pz:NamedPizza ) ] ,
18     [ ottr:of o-owl-re:ObjectUnionOf ;
19       ottr:values ( _:b0 _:toppings ) ] ,
20     [ ottr:of o-owl-ax:SubObjectAllValuesFrom ;
21       ottr:values ( _:pizza pz:hasTopping _:b0 ) ] .
```

Figure 11 The `o-p:NamedPizza` template in wOTTR syntax.

representation for expressing parameter lists, argument lists and complex type specifications. The entities defined in the wOTTR vocabulary lie close to the formal vocabulary established in Section 3; the classes, properties and named individuals of the vocabulary are listed in Table 3, Table 4, and Table 5, respectively. The wOTTR vocabulary is developed in GitLab,[10] and published at `ottr.xyz`[11] and Zenodo.[12]

Although the mapping from the wOTTR vocabulary to the formally defined concepts of OTTR should be immediate, there are some design choices and peculiarities that are due to the wish for a compact and readable representation, and the constraints of the RDF and OWL standards. We will illustrate these by using the `o-p:NamedPizza` template represented in the wOTTR vocabulary, which is listed in Figure 11.

Variables As RDF does not include variables we have chosen to use blank nodes for representing a template's parameters. The variables of a template are specified as a list of parameters using the predicate `ottr:variable`, see, e.g., line 3 in Figure 11. Care must then be taken to not use the same blank nodes as constants.

Lists The wOTTR language makes frequent use of RDF lists as a means to represent an ordering of resources. We do this since RDF lists have a succinct serialization in RDF Turtle and as it is syntactically similar to ordinary function calls and predicates. In Figure 11, lists are used for parameter lists (starting on line 3), instance argument lists (e.g., on line 13), instance argument value lists (e.g., on line 9), and complex type specifications; line 6 specifies the type `NEList<owl:Class>`.

Annotation properties All properties of the wOTTR vocabulary are annotation properties. This is to indicate that the vocabulary is not intended to be used for reasoning over templates and instances. The use of RDF lists, as explained above, also places the vocabulary outside the OWL 2 DL fragment, as RDF lists are used in the serialization of OWL.

[10] `https://gitlab.com/ottr/spec/wOTTR`
[11] `https://spec.ottr.xyz/wOTTR/0.4.5/`
[12] `https://zenodo.org/records/12581215`

Table 3 wOTTR vocabulary classes and their definition. The `ottr:` prefix is omitted.

Class	Definition
`:Signature`	A **signature** specifies the permissible input for instances. It does this through its list of parameters. The IRI of the signature is a unique name that its instances must reference.
`:Template`	A **template** is a signature that additionally specifies a pattern. The pattern, which is a set of instances, determines the result of the direct expansion (1-step expansion) of an instance of the template.
`:BaseTemplate`	A **base template** is a signature with no pattern. The expansion of an instance of a base template is the instance itself.
`:Parameter`	A **parameter** specifies the variable terms or resources of a pattern and the permissible values for the corresponding instance arguments.
`:ParameterModifier`	A **parameter modifier** is a flag or marker that is set on a parameter to alter the permissible corresponding argument values and/or the behaviour of expanding instances.
`:Instance`	An **instance** is an instantiation of a signature, template or base template. The instance must refer to a signature and provide arguments that match the corresponding parameters of the signature.
`:ExpansionModifier`	An **expansion modifier** is a flag or marker that is used to alter the behaviour of expanding the marked instance.
`:Argument`	An **argument** specifies an input value for a given instance.
`:ArgumentModifier`	An **argument modifier** is a flag or marker that is used to identify that the argument plays a special role in modified expansions. See also ExpansionModifier.

Two instance shapes Template instances are stated using the property `ottr:of` which specifies the template. Instances may be specified using two different shapes, called *compact* and *canonical*. The compact shape uses the property `ottr:values` and an RDF list to directly give the argument values of the instance, very similar to how instances are written in stOTTR; line 9 shows an example. The canonical shape uses the property `ottr:arguments` and an RDF list of arguments, where each argument, usually represented by blank node, has a `ottr:value` property that sets the argument value; line 15 shows an example. The canonical shape can be used in all cases, but must be used when more data than just the argument value is required, as line 15 exemplifies by marking the argument for list expansion.

5 Template Libraries

A template library is a collection of templates developed and curated for a particular purpose, such as representing patterns for a given vocabulary, domain, or project. The ability to share and reuse templates for common modelling patterns is a core feature of the OTTR framework. This section gives an overview of the vision behind OTTR template libraries and the support and developments made towards the vision. Large parts of this section are taken from previous publications [52, 34] and are included here to give a complete presentation of the OTTR framework.

The vision of template libraries is similar to the role APIs and repositories of API source code play in software engineering. Just as software projects rely on stable access to APIs and documentation to work and be understood and used, ontology engineering projects using OTTR must be able to rely on the availability and documentation of templates. For this reason, it is critical that a published template does not change in any way that may affect the expansion of

Table 4 wOTTR vocabulary properties, indicating their domain and range. The `ottr:` prefix is omitted.

Property	Domain	Range	Definition
`:parameters`	`:Signature`	List of `:Parameter`	Associates a signature with one required list of parameters.
`:annotation`	`:Signature`	`:Instance`	Associates a signature with an optional set of annotation instances.
`:variable`	`:Parameter`	`rdfs:Resource`	Sets the required variable of a parameter.
`:type`	`:Parameter`	(List of) `rdfs:Resource`	Sets an optional type of a parameter. A missing type implicitly sets the type to the most general type.
`:default`	`:Parameter`	`rdfs:Resource`	Sets an optional default value of a parameter. The default value is used in case an argument value is unspecified or is `ottr:none`.
`:pattern`	`:Template`	`:Instance`	Associates a template with an optional set of pattern instances.
`:name`		`xsd:token`	A human readable name or label.
`:of`	`:Instance`	`:Signature`	Associates an instance with its required signature.
`:arguments`	`:Instance`	List of `:Argument`	Associates an instance with a list of arguments.
`:values`	`:Instance`	List of `rdfs:Resource`	Associates an instance with a list of argument values
`:value`	`:Argument`	`rdfs:Resource`	Associates an argument with its argument value.
`:modifier`			

Table 5 wOTTR vocabulary individuals and their type. The `ottr:` prefix is omitted.

Named Individual	Class	Definition
`:optional`	`:ParameterModifier`	**optional** is a parameter modifier which makes the value none a permissible instance argument value for this parameter.
`:nonBlank`	`:ParameterModifier`	**nonBlank** is a parameter modifier which makes blank nodes illegal instance argument values for this parameter.
`:cross`	`:ExpansionModifier`	**cross** is an expansion modifier which sets the list expansion operation to cross product.
`:zipMax`	`:ExpansionModifier`	**zipMax** is an expansion modifier which sets the list expansion operation to zip, extending smaller list to the length of the longest list by appending empty values.
`:zipMin`	`:ExpansionModifier`	**zipMin** is an expansion modifier which sets the list expansion operation to zip with the shortest list as length.
`:listExpand`	`:ArgumentModifier`	**listExpand** is an argument modifier that selects arguments for list expansion.
`:none`	`rdfs:Resource`	**none** is an individual which is used to designate a missing argument value.
`:Triple`	`:BaseTemplate`	**Triple** is a base template that represents an RDF triple.
`:NullableTriple`	`:BaseTemplate`	**NullableTriple** is a base template that represents an RDF triple and permits none value arguments.

its instances, and that the expansion can be performed at any time. The meaning of a template instance must stay constant; an instance should be considered as semantically equivalent to its expansion. This places strong requirements on the availability and versioning of templates.

To support the quality of template libraries and the management of these, concepts and procedures for library governance, together with methods for library maintenance and methodologies for library construction have been developed. A documentation system together with a purpose-built set of documentation templates is available for annotating templates to generate user-friendly documentation pages for publishing template libraries. This is presented in more detail below.

5.1 Template Life-cycle Management

To aid the life-cycle management of templates in the library, a set of template statuses and an interpretation of versioning categories for templates has been proposed [52].

5.1.1 Status

A template's status indicates the maturity of the template and its level of support and endorsement. Each template has exactly one of the following statuses, here ordered from low to high:

$$\textit{incomplete} < \textit{draft} < \textit{candidate} < \textit{proposed recommendation} < \textit{recommendation}$$

A template should not depend on templates of lower status than the template's own status. A template may additionally have the status of *deprecated*. The statuses are described in more detail below.

An **incomplete** template is of the lowest status, which is the default if no status is stated for a template. The only requirement for an incomplete template is that it must be syntactically correct according to its serialization format, but need not otherwise be a valid template. This means is it permissible for an incomplete template to, for example, depend on templates that do not (yet) exist or that contain type errors. An incomplete template is typically a placeholder for future work and should not be published for public use.

A **draft** template must be a syntactically correct and well-founded template, i.e., the template is completely defined and does not contain any formal errors. A draft template should not be considered mature or stable. In terms of its life-cycle, it is published in order to be available to others, both for use and for further development.

A **candidate** template is a draft template which additionally contains a complete set of metadata and is endorsed by a named individual or organization that aims to promote the template to recommendation status. The endorser is expected to actively participate in the support and promotion of the template; failure to do so may result in the deprecation of the template. A candidate template should be considered stable.

After a period in candidate status, a candidate template may be proposed as a recommendation and given the status of **proposed recommendation**. This triggers a public vote to promote the template to recommended status. Relevant comments and issues collected during the voting phase must be addressed if the template can be given the status of recommended.

A **recommended** template is of the highest status. This means that the template is of high quality and is well-integrated into the library.

A **deprecated** template is discouraged from use. An explanation for why a template is deprecated should be given.

5.1.2 Versioning

Each time a template is published, a new version number must be assigned to the template using semantic versioning[13] and the numbering scheme *major.minor.patch*, e.g., 0.2.3. In our translation of these types of updates to OTTR templates the notions of the expansion of a template and its signature are central. Any changes to the signature of a template which make existing instances incompatible with the updated template are backwards *in*compatible changes. For OTTR templates we use the following definitions:

Patch updates are backwards compatible changes that do not affect the expansion of a template. These changes will neither affect existing instances of the updated template nor their expansion.

Minor updates are backwards compatible changes that may alter the expansion of the template. An example is adding or removing body instances. This means that existing instances of the template remain legal and valid for the new version of the template, but their expansion will be different.

Major updates are backwards incompatible changes. An example is adding or removing a parameter to/from the template signature, or restricting the type of a parameter. This means that existing instances of the template will not be legal instances of the updated template.

In the case that the update greatly changes the perceived meaning of the template, the update should be categorized as a greater change than according to the above descriptions.

Draft templates are exempted from the above rules. Following the semantic versioning specification we require that draft templates have a version number with the major version 0, which signals that anything may change at any time and the public API should not be considered stable. When a template reaches the next status of candidate, the major version must be set to 1.

5.2 Metadata

Template metadata, like the status and version of a template, but also textual explanations, examples and provenance information, is captured by annotation instances placed on templates. The `docttr` package of the core template library contains templates created specifically for this purpose. The following lists the templates in the package and the information they are designed to capture:
`Signature`: label, description, scope and editorial notes, related resources
`Provenance`: time of creation and update, authors and contributors
`Version`: status, version number, references to previous and next versions
`ChangeNote`: change description, including author and timestamp of the change
`Example`: explanatory examples of the template
`Deprecated`: mark the template as deprecated, including an explanation of why
`Parameter`: parameter description, example, notes
The documentation of the template package is found at `http://tpl.ottr.xyz/p/docttr/`.

5.3 docTTR: Template Documentation System

The documentation system for OTTR templates is called *docTTR*. docTTR reads as input all the templates that constitute a library and produces a set of interlinked HTML pages. All pages are presented in an HTML frameset layout, inspired by Javadoc,[14] a documentation system for Java.

[13] `https://semver.org/spec/v2.0.0.html`
[14] `https://www.oracle.com/java/technologies/javase/javadoc-tool.html`

Each template documentation page contains the metadata captured by annotation instances, a list describing its parameters, the template rendered in different serialization formats, a generated sample instance of the template in all formats with its resulting expansion, both visualized and in RDF Turtle format; an expansion of the sample instance, a dependency graph showing all the templates that the template, and a list of the vocabulary elements that the template introduces.

5.4 Template Relations for Library Maintenance

Using the formal fundamentals of the OTTR language it is possible to define logical relationships between templates that describe characteristics that concern the quality of template libraries. 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. To this end, we use the following template relations.

▶ **Definition 63.** *Let $T_1 = ((t_1, P_1, A_1), B_1)$ and $T_2 = ((t_2, P_2, A_2), B_2)$ be two templates (that is, template T_i has name t_i, parameters P_i, annotations A_i and pattern B_i). We say that T_1* overlaps *T_2 if there exist sets of template instances $I_1 \subseteq B_1$ and $I_2 \subseteq B_2$ and substitutions ρ_1 and ρ_2 of respectively P_1 and P_2 such that $\rho_1(I_1) = I_2$ and $\rho_2(I_2) = I_1$.*

Furthermore, we say that T_1 contains *T_2 if there exists a substitution ρ of the parameters of T_2 such that $\rho(B_2) \subseteq B_1$.*

Using these relations, we can now formally define notions of redundancy in template libraries.

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 the pattern of S with a suitable instance of T.

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.

We have implemented an algorithm for efficiently identifying uncaptured patterns in large template libraries, and used it to successfully refactor a real-world template library [50].

5.5 Template Development Methodologies

The recursive structure of OTTR templates supports a top-down modelling approach to template library development, where complex modelling patterns are iteratively broken down into less complex patterns.

Lupp et al. [34] use this approach, and identify different informal layers to a template library:

Base templates are at the lowest level and specify patterns over an underlying data representation language.

Utility templates are used to improve template formulation by grouping base template instances to avoid unnecessary repetition. Utility templates represent patterns that are typically only meaningful for users familiar with the language that the base templates abstract over, e.g., RDF.

Logical templates represent ontological axioms and convenient combinations of axioms, such as subclass relationship and concept partitioning.

Domain templates represent modelling patterns in a specific domain. They are independent of specific input formats and represent common domain conceptualizations.

System/User-facing templates record patterns that represent end-user or system formats which typically involve complex combinations of domain statements. A system template is hence only required if the system representation differs from the domain conceptualization and if the format is useful to represent as a template. This may be the case when the format is common, e.g., if the system is a de facto standard in the domain.

Each layer provides an "interface" for the layers above it to use, hence hiding the complexity of lower layers. The layering helps with the construction and maintenance of a template library in that the various layers typically fit different competencies and may be managed by people with different expertise – possibly with the assistance of ontology experts.

Blum et al. [5] give insights from an OTTR-specific ontology engineering methodology similar to that of Lupp et al. [34]. They characterize their methodology as both bottom-up, in the sense that existing data is taken as the starting point for developing templates, and top-down, as the methodology exploits the recursive structure of OTTR templates to incrementally break down the more complex data-facing templates to OWL and RDF templates. In their work, they found that OTTR templates helped simplify the communication between subject-matter experts and ontology experts in that it allows them to focus on *what* to model – the information content, represented by template signatures – without the need to consider *how* to model it, which is later represented by template bodies.

5.6 Public Template Libraries

There are a handful of publicly available OTTR template libraries:

Core OTTR Library [52] This library is developed and maintained by the OTTR team. It contains about 200 templates that predominantly represent common patterns over the OWL, RDFS and RDF vocabularies. It is intended to be a central resource for all OTTR users and other template libraries. The library is available at `https://tpl.ottr.xyz`.

POSC Caesar Association The POSC Caesar Association (PCA) has published the Product Life-cycle Management Reference Data Library (PLM-RDL) OTTR Library at `https://rds.posccaesar.org/ontology/plm/tpl/0.1/`:

> *The PLM-RDL OTTR library is specifically intended to cover the process industry domain and the templates will, in general, make direct reference to resources from the PLM-RDL. If you wish to build ontologies that extend on the PLM-RDL, using these templates will help ensure that your modelling patterns are fully consistent with those of the PCA ISO 15926-14 compliant ontologies.*

DiProMag The DiProMag project has published an OTTR template library for modelling certain characteristics of chemical elements at `https://www.dipromag.de/dipromag_onto/`. (See also Section 8.4.)

Aspect OWL The ontology design patterns for representing context in ontologies using AspectOWL [45] are encoded by an OTTR template library available at `https://odp.aspectowl.xyz/`.

6 Instantiation Tools

The tabular format specified by OTTR template signatures is well-suited for consuming and converting tabular data to RDF by way of OTTR's expansion mechanism. Assuming the format of the input table matches the template signature, each row in the input table is naturally mapped to a template instance. To this end, the OTTR framework specifies two methods for selecting and converting tabular data to OTTR template instances: *tabOTTR* specifies a small markup language for mapping tabular data files to OTTR template, and *bOTTR* is an RDF vocabulary for specifying mappings from database query results to OTTR template instances.

6.1 tabOTTR: Tabular OTTR Template Instances

The intended use of tabOTTR is to annotate existing tabular datafiles, such as CSV files and spreadsheets, with instructions that specify the data to select and how to transform it to the correct datatypes for OTTR template instantiation.

In order to generalize over different tabular file formats we use the terms *file*, *table*, *column*, *row* and *cell*. Tables, columns and rows within a file have a unique index which is a positive integer number assumed to be numbered in a straight-forward ordered consecutive manner. A cell has a 3-dimensional coordinate given by the indices: (table, column, row). For CSV files, we call the entire contents of the file a table and assign this the index 1. For spreadsheets, such as in MS Excel files, each sheet in a file represents a table.

The results of processing a file according to the tabOTTR specification is a set of template instances. Processing instructions are inserted directly into tables with the token `#OTTR`. This token must appear in the first column of the table, and the cell to the right of the token must contain a processing instruction name with possible instruction arguments given in consecutive cells immediately to the right of this cell. Each table can contain multiple processing instructions, and when processing a file all tables in the file are processed. There are three such processing instructions: `end`, `prefix` and `template`, which will be discussed below. An instruction dictates the interpretation of subsequent rows, and the scope of the instruction is the following rows until a row containing a new instruction or until the end of the table. Rows which are not in the scope of an instruction are not processed. All processed cells are trimmed for leading and trailing whitespace. All IRIs may be given using a QName, using prefixes set with the `prefix` instruction.

The instructions are defined as follows:

`end` instruction The `end` instruction takes no arguments. It is a no-operation instruction that takes no arguments and whose purpose is simply to terminate the previous instruction. We recommend to always use the `end` instruction to terminate an instruction, so that the scope of an instruction is made explicit.

`prefix` instruction The `prefix` instruction declares namespace prefixes that may be used in other instructions and that will be included in the processed output. The `prefix` instruction takes no arguments. Each following row declares a namespace prefix, where the

- 1. column contains the prefix name, and the
- 2. column contains the namespace name.

All other columns are ignored. The scope of the defined prefixes is the whole input file. Conflicting declarations of the same prefix name must raise an error.

`template` instruction The result of processing a `template` instruction is a set of template instances. A `template` instruction takes one argument: a template IRI. Subsequent rows have special meaning; assume the `template` instruction is on a row with index r, then:

- Row $r + 1$ contains cells that specify the template argument index of the template instances. Cell values must be a non-negative integer or the empty string.
- Row $r + 2$ contains cells that specify the type instruction of the template instance arguments. Cell values must have a legal type instruction; they are listed in Table 6.
- Row $r + 3$ is ignored. This row can be used for informative content such as column headers.
- The following rows, rows $> r + 3$, in the scope of the `template` instruction specify each one instance of the template whose IRI is specified by the `template` instruction. The cells in these rows contain instance argument values. Each argument value is together with its

Table 6 Permissible type instructions and their RDF resource interpretation.

Type instruction	RDF value interpretation
`iri`	IRI
`blank`	blank node
`text`	untyped literal
an IRI	typed literal
`auto`	determined by value, see below
X`+`, where X is one of the above type instructions	RDF list with list items determined by X

corresponding type instruction (as specified in row $r+2$) translated into an RDF resource following Algorithm 1. Each row then represents an argument list to a template instance ordered according to the positive integer indices of row $r+1$.

▶ **Example 64.** This table contains a `prefix` instruction that declares two prefixes: `ex` and `foaf`. The `prefix` instruction is terminated by an `end` instruction.

```
#OTTR    prefix
ex       http://example.net#
foaf     http://xmlns.com/foaf/0.1/#
#OTTR    end
```

The result of processing the above table is equivalent to the following RDF Turtle.

```
@prefix ex:    <http://example.net#> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/#> .
```

Type instructions are required for describing how values in tabular data files, which may not have any typing information, are to be translated into RDF resources. This makes it possible to for instance create an IRI or an RDF literal from the string value `http://example.com/Bob`. The values from columns that are `auto` typed are translated based on their string value and may hence end up as RDF resources of different RDF types.

Example 65 gives a simple demonstration of the `template` instruction. Table 7 demonstrates the translation of `auto` typed values. Figure 10 on page 13 shows a spreadsheet that specify 22 instances of the `o-p:NamedPizza` template listed in Figure 5.

▶ **Example 65.** The table below contains one template instruction, declaring instances of the `ottr:Triple` template, as prescribed by the first row. The next row selects which columns to use as arguments to the template (which takes 3 arguments). Columns with a value `0` will be ignored and can be, e.g., used for comments or spreadsheet formula calculations of intermediate values. The next row sets the type instruction for the columns which contain arguments, in this example, all columns have the type instruction `iri` which means that argument values are interpreted as IRIs. The next row is ignored and may be used for a descriptive label. The following rows specify instances of the templates. The `end` instruction closes the scope of the `template` instruction.

```
#OTTR        template     ottr:Triple
2            1            3            0            0
iri          iri          iri
Predicate    Subject      Object
foaf:knows   ex:Ann       ex:Bob
foaf:knows   ex:Bob       ex:Carl
#OTTR        end
```

Algorithm 1 Pseudo algorithm for translating tabular file argument values to RDF resources.

```
function translate(typeInstruction, value)
    Output : RDF resource
    if value = empty string then
        return ottr:none
    else if typeInstruction is of the form X+ then
        items ← split value by | ;
        foreach item in items do item ← translate(X, item);
        return RDF list of items
    else if typeInstruction = iri then
        return IRI of value, accepting QNames and full IRIs
    else if typeInstruction = blank then
        if value = * then
            return Fresh blank node
        else
            return Blank node with label value
        end
    else if typeInstruction is of the form of a QName or full IRI then
        return Typed literal with lexical value value and datatype typeInstruction
    else if typeInstruction = text then
        return Untyped literal with lexical value value
    else if typeInstruction = auto then
        if value is of the form text^^type then
            return Typed literal with the lexical value text and type
        else if value is of the form text@@lang then
            return Language tagged literal with the lexical value text and language tag lang
        else if value = true, TRUE, false or FALSE then
            return Literal with boolean value of value and datatype xsd:boolean
        else if value is of the form of a decimal number then
            return Literal with lexical value value and datatype xsd:decimal
        else if value is of the form of an integer number then
            return Literal with lexical value value and datatype xsd:integer
        else if value is of the form of an RDF Turtle labelled blank node then
            return Blank node with label value
        else if value is of the form of a QName or full IRI then
            return IRI of value
        else
            return Untyped literal with lexical value value
        end
    else
        return Untyped literal with lexical value value
    end
```

Table 7 Examples of the translation using `auto` type instruction.

value with `auto` instruction	RDF resource	RDF type
`*`	`[]`	fresh blank node
`_:myBlank`	`_:myBlank`	labelled blank node
`myBlank`	`"myBlank"`	untyped literal
`ex:Ann`	`ex:Ann`	IRI
`http://other-example#Bob`	`<http://other-example#Bob>`	IRI
`Carl`	`"Carl"`	untyped literal
	`ottr:none`	IRI, special *none* value
`true`	`"true"^^xsd:boolean`	`xsd:boolean`
`True`	`"True"`	untyped literal
`1`	`"1"^^xsd:integer`	`xsd:integer`
`-1.2`	`"-1.2"^^xsd:decimal`	`xsd:decimal`

The table specifies the following template instances:

```
ottr:Triple(ex:Ann, foaf:knows, ex:Bob) .
ottr:Triple(ex:Bob, foaf:knows, ex:Carl) .
```

6.2 bOTTR: Batch Instantiation of OTTR Templates

bOTTR is an RDF vocabulary for specifying mappings between queries over the sources to given templates. bOTTR hence allows multiple data sources on different formats to be integrated via OTTR templates into a single RDF/OWL representation. The bOTTR vocabulary is specified by an OWL ontology that extends the wOTTR vocabulary (See Section 4.2), and provides terms for specifying sources, queries over these sources, and how to map the query results into instances of a specified OTTR template. The bOTTR specification is developed in GitLab,[15] and published at `ottr.xyz`[16] and Zenodo.[17]

The classes `ottr:InstanceMap`, `ottr:Source` and its subclasses, and `ottr:ArgumentMap`, which specify respectively the mappings, the source and the translation of source values to template instance arguments, are described below.

6.2.1 InstanceMap

The central notional of bOTTR is `ottr:InstanceMap`. An `ottr:InstanceMap` specifies a mapping between one `ottr:query` over a given `ottr:Source` and one `ottr:Template`. The result of applying the mapping is that each record in the query result set becomes an `ottr:Instance` of the specified `ottr:Template`. The `ottr:argumentMaps` specify how source values are translated to instance arguments.

An `ottr:InstanceMap` must specify a `ottr:template`, a `ottr:source`, a `ottr:query`, which must be a valid query over the specified `ottr:Source`, and optionally a list of `ottr:argumentMaps`. If set, the size of the argument map list must match the size of the query result records. These specify how query result values are translated to instance argument values. The result of processing a set of `ottr:InstanceMaps` is the set of instances resulting from each of the `ottr:InstanceMaps`.

[15] `https://gitlab.com/ottr/spec/bOTTR`
[16] `https://spec.ottr.xyz/bOTTR/0.1.2/`
[17] `https://zenodo.org/records/12607264`

6.2.2 Sources

A `ottr:Source` defines a source of data and how it can be accessed. The location of the source is specified with `ottr:sourceURL`, this can be either a URL or a file path. There are two main types of `ottr:Sources`: `ottr:StringSource` and `ottr:RDFSource`. These source types again have subclasses. All query result values from a `ottr:StringSource` are assumed to be strings, and if no `ottr:argumentMaps` are set, then these source values are represented as untyped literals when transforming them into instance arguments. A `ottr:StringSource` makes no assumption on the type of its `ottr:query`, this must be specified by using a subclass of `ottr:StringSource`. For an `ottr:RDFSource`, all values are assumed to be RDF resources. If no `ottr:argumentMaps` are set then the values are left as they are when used as instance arguments. An `ottr:RDFSource` accepts only SPARQL queries.

There are four types of sources:

`ottr:RDFFileSource` is an `ottr:RDFSource` which is specified by one or more RDF files using the `ottr:sourceURL` property.

`ottr:SPARQLEndpointSource` is an `ottr:RDFSource` which is specified by a single SPARQL endpoint address using the `ottr:sourceURL` property.

`ottr:JDBCSource` is a `ottr:StringSource` which is specified using a single JDBC connection string using the `ottr:sourceURL` property. Additionally, a `ottr:JDBCSource` can specify a `ottr:jdbcDriver`, `ottr:username`, and `ottr:password` for connecting to the database, and set a `ottr:fetchSize` to indicate the number of query result rows retrieved on each fetch.

`ottr:H2Source` specifies a temporary H2 database source which is useful for loading and querying a CSV file using functionality supported by the H2 SQL query language. The path given to `CSVREAD` may use the special constant `@@THIS_DIR@@` within the path to denote the directory of the bOTTR input file.

An example of each source can be seen in Example 66.

► **Example 66.** Below are examples of all the four sources. For the `ottr:H2Source`, we give a full instance map to illustrate the use of the `CSVREAD` function in the query.

```
[] a ottr:RDFFileSource ;
  ottr:sourceURL <http://example.com/file1.ttl>, <http://example.com/file2.ttl> .

[] a ottr:SPARQLEndpointSource ;
  ottr:sourceURL <http://example.com/sparql/> .

[] a ottr:JDBCSource ;
  ottr:sourceURL "jdbc:mysql://localhost/mydb" ;
  ottr:jdbcDriver "com.mysql.jdbc.Driver" ;
  ottr:username "Ann" ;
  ottr:password "password123" .

[] a ottr:InstanceMap ;
  ottr:source [ a ottr:H2Source ] ;
  ottr:query """
    SELECT 'First Name', 'Age', 'City', 'Homepage'
    FROM CSVREAD('@@THIS_DIR@@/data/address.csv');
  """ ;
  ottr:template ex:Person .
```

6.2.3 ArgumentMap

An `ottr:ArgumentMap` specifies how source values are mapped to OTTR template instance arguments, i.e., RDF terms. In case no argument maps are specified, defaults apply from the choice of `ottr:Source` defined above. When applying an argument map to a source value, we sometimes refer to the **string value** of the source value. In case the source is a `ottr:StringSource`, then the string value of a source value x is x. In case the source is an `ottr:RDFSource`, the string value is of an RDF resource x is:

- the lexical value of x, if x is a literal
- the IRI of x, if x is a IRI,
- the blank node label of x, if x is a blank node

An `ottr:ArgumentMap` is defined by the following properties, each specifies how values are transformed to RDF terms. The properties are applied in the order they are presented below.

`ottr:nullValue` Specifies the argument value to be used in case the source value is unspecified or `NULL` (as defined in the source's format). This value may be any RDF resource.

`ottr:labelledBlankPrefix` Selects which values to translate to labelled blank nodes. The default value is `_:`. If the string value of the source value starts with and is longer than the `ottr:labelledBlankPrefix`, then the argument value becomes a labelled blank node where the string value of the source value following the `ottr:labelledBlankPrefix` becomes the label. Example: if the `ottr:labelledBlankPrefix` is `"ABC"` and the source value is `"ABCDEF"`, then a labelled blank node `_:DEF` is created.

`ottr:languageTag` Specifies the language tag of the argument value. If this value is set, then the source value becomes a language tagged literal (and automatically gets `ottr:type rdf:langString`) where the lexical value is the string value of the source value.

`ottr:listStart`, `ottr:listEnd`, `ottr:listSep` Specifies how to translate source values into lists using the string value of the source value. The default values are respectively: `(` , `)` and `,`. A value may represent a nested list of arbitrary depth. `ottr:listStart` and `ottr:listEnd` must be a one-character string that specifies respectively the start and end of a list. `ottr:listSep` is a string that specifies how the list elements are separated. The list element values are trimmed for white space. These properties may only be used if the `ottr:type` is a list type. Example: if the source value is `"(( a , b ), (c , d))"`, and the `ottr:type` is `(rdf:List rdf:List xsd:string)`, then the value becomes the RDF list `(( "a" "b" )("c" "d"))`.

`ottr:type` Specifies the `ottr:Type` of the argument value (using the wOTTR encoding of our type system). If the source is an `ottr:RDFSource` and the type of the source value is compatible with the specified `ottr:type`, then the argument value is equal to the source value (the argument map leaves the value unchanged). If the value is not compatible or the source is a `ottr:StringSource`, then the string value of the source value is cast to the specified `ottr:type`. This may result in an error if the cast is not possible. This property may not be used if `ottr:labelledBlankPrefix` is used.

▶ **Example 67.** This example is an adaption of Example 66 to illustrate the functionality of argument maps.

```
[] a ottr:InstanceMap ;
  ottr:source [ a ottr:H2Source ] ;
  ottr:query """
    SELECT 'Homepage', 'First Name' || ' ' || 'Last Name' AS fullName, 'Bdate'
    FROM CSVREAD('people.csv');
  """ ;
  ottr:template ex:Person ;
```

```
ottr:argumentMaps (
  [ ottr:type ottr:IRI ]
  [ ] ## empty Argument map
  [ ottr:type xsd:date; ottr:nullValue ottr:none ]
) .
```

Assuming that the contents of `people.csv` are

First Name	Last Name	Bdate	City	Homepage
Ann	Annsen	1990-12-01	Amsterdam	`http://example.com/Ann`
Bob	Bobson	1987-03-23	Berlin	`http://example.com/Bob`
Carl	Carlson	NULL	Cairo	`http://example.com/Carl`

then the instance map will produce the following instances:

```
ex:Person( <http://example.com/Ann>, "Ann Annsen", "1990-12-01"^^xsd:date ) .
ex:Person( <http://example.com/Bob>, "Bob Bobson", "1987-03-23"^^xsd:date ) .
ex:Person( <http://example.com/Carl>, "Carl Carlson", none ) .
```

7 Implementations

There exist multiple implementations of the OTTR framework. One of these is a reference implementation that is developed and maintained by the OTTR project and which supports all the OTTR specifications. The other implementations are developed independently of the OTTR project and are motivated in part by the need to make the OTTR framework available in other platforms and programming languages than that of the reference implementation.

7.1 Lutra: The Reference Implementation of OTTR

Lutra is the reference implementation for the OTTR framework, written and actively maintained by the developers of OTTR. It is an open-source project under an LGPL licence, available at GitLab,[18] at a mirror repository at GitHub,[19] and with released sources also stored in Zenodo.[20] Lutra is written in Java and is built with Maven; the Maven artefacts are available through the Sonatype repository under the Java namespace `ottr.lutra`.[21] The codebase is developed following various established best practices: semantic versioning,[22] code style profile, static code analysis, unit testing, continuous integration and continuous deployment (CI/CD),[23] and the git-flow branching model.[24]

Lutra supports all the specifications presented in this paper:

- Expanding instances and templates
- Type checking templates and instances according to the type hierarchy of Table 1
- Reading and writing templates and instances in the stOTTR and wOTTR serialization formats
- Generating docTTR documentation for template libraries
- Processing tabOTTR spreadsheets, supporting only Excel spreadsheet input
- Processing bOTTR specifications, supporting all types of sources mentioned in Section 6.2

[18] `https://gitlab.com/ottr/lutra/lutra`
[19] `https://github.com/rtto/lutra-mirror`
[20] `https://zenodo.org/records/10954639`
[21] `https://central.sonatype.com/search?q=ottr.lutra`
[22] `https://semver.org/`
[23] `https://about.gitlab.com/topics/ci-cd/`
[24] `https://nvie.com/posts/a-successful-git-branching-model/`

```
Usage: lutra [-fhV] [--debugStackTrace] [--quiet] [--stdout] [-F=<fetchFormat>]
             [--haltOn=<haltOn>] [-I=<inputFormat>] [-L=<libraryFormat>]
             [-m=<mode>] [--messageLinePrefix=<linePrefix>] [-o=<out>]
             [-O=<outputFormat>] [-p=<prefixes>] [-e=<extensions>[,
             <extensions>...]]... [-E=<ignoreExtensions>[,
             <ignoreExtensions>...]]... [-l=<library>]... [<inputs>...]
```

Figure 12 Lutra's command line interface options and flags as reported by its `-help` option. (The help output also gives an explanation of the options and flags which is not shown here.)

There are three available modes of using Lutra: (1) as a command line interface (CLI) serviced by a Java executable JAR file, (2) as a Java API, or (3) as a web application. The available options and flags of CLI are listed in Figure 12. Possible modes of operation (set with option `-m`) are:

`expand` expands the input instances according to the specified template libraries and fetched templates, and writes the expansion result to the specified output format.
`format` (re)formats the input instances to the specified output format.
`expandLibrary` expands the specified template libraries.
`formatLibrary` (re)formats the specified template libraries to the specified output format.
`docttrLibrary` generates documentation pages for the specified template libraries.
`lint` checks the input instances or templates for errors.
`checkSyntax` runs a syntax check of the input, typically for use as an external service for editors.

The API provided by the Java project is documented using Javadoc.[25] The central class is `TemplateManager`[26] that is responsible for orchestrating the reading and writing of templates and instances. It contains methods for reading a library of templates from a file or folder, checking its correctness, reading instances from files, folders or mappings (both bOTTR and tabOTTR), expanding the instances and checking that the instances are correct with respect to a template library, translate instances or templates from one serialization to another, and more.

A web application variant of the CLI, called *WebLutra*, is available at `https://weblutra.ottr.xyz`. The intention of WebLutra is to make the OTTR framework available for simple use and experimentation without the need for installing and running software locally. WebLutra also serves the interactive examples that are part of the online primer for OTTR available at `https://primer.ottr.xyz`. It is implemented as a simple "wrapper" over the CLI interface exposing certain operations through a simple web form. WebLutra is packaged as a Docker image that is available from GitLab's container registry.[27]

7.2 maplib: Support for Data Frame Mappings with OTTR

maplib [1] is a library written in Rust using Apache Arrow,[28] Polars[29] DataFrames and with Python bindings, which allows data frames, a popular data structure used in data analytics tools, to be mapped to RDF by way of OTTR templates.[30] By exploiting libraries and formats built for

[25] `https://javadoc.io/doc/xyz.ottr.lutra`
[26] `https://javadoc.io/doc/xyz.ottr.lutra/lutra-core/latest/xyz/ottr/lutra/TemplateManager.html`
[27] `https://gitlab.com/ottr/lutra/lutra/container_registry/1986127`
[28] `https://arrow.apache.org/`
[29] `https://pola.rs/`
[30] `https://pypi.org/project/maplib/`

data processing, maplib is able to perform OTTR instance expansion with high performance. It also brings OTTR closer to use for data analytics and in platforms such as Jupyter.[31] From the documentation the library appears to only support templates in stOTTR syntax and it is not clear if the library supports all language features of OTTR. maplib is open source[32] and in active development.

7.3 OTTR Extension: Semantic MediaWiki Extension

OTTR Extension[33] is an extension for Semantic MediaWiki (SMW) [30] which enables some of OTTR's functionality within SMW [5]. The OTTR Extension allows templates and instances in stOTTR format to be entered directly into SMW pages. Taking an OTTR template as input, the OTTR Extension can generate an input web form in SMW that matches the signature of the template and with which instances of the template can be created by filling in the form. These instances expand to triples, as per its template definition, that are available in the page of the created template instance. The latest release is dated March 2023.

7.4 pyOTTR: Python Packages

There are two packages that implement OTTR functionality for Python, one developed by GitHub user Callidon[34] and one developed by GitHub user michalporeba.[35] Both packages are called *pyOTTR*, although they appear as two separate projects.

Callidon's pyOTTR package supports reading templates and instances in stOTTR format and expanding instances to RDF. The package is limited to supporting only the stOTTR format and does not support the complete functionality of the OTTR language, in particular, list expansion modes are listed as in development. However, the project appears abandoned as the latest code update was five years ago.

michalporeba's pyOTTR package appears to support reading templates in stOTTR format, and expanding instance data read from CSV files to RDF. Whether the package supports the full OTTR language is not clear from the documentation. The package appears to be in a state of early and active development.

7.5 emacs-ottr-toolkit

The emacs-ottr-toolkit[36] is a collection of Emacs scripts to facilitate OTTR operations in GNU Emacs org-mode.[37] It uses existing Emacs extensions like Flycheck[38] and the Lutra CLI executable (Section 7.1) to offer shortcut commands and code snippet functionality that simplifies operations such as writing templates, syntax checking, and template instantiation from tables and query result sets.

[31] `https://jupyter.org/`
[32] `https://github.com/DataTreehouse/maplib`
[33] `https://www.mediawiki.org/wiki/Extension:OttrParser`
[34] `https://github.com/Callidon/pyOTTR`
[35] `https://github.com/michalporeba/pyOTTR`
[36] `https://ottr.xyz/event/2021-06-18-user-forum/ottr-toolkit-20210618.html`
[37] `https://orgmode.org/`
[38] `https://www.flycheck.org`

8 Uses

The OTTR project has nurtured a healthy collaborative environment with its users particularly through the organization of several *OTTR user forums*[39] where OTTR developers and users have met to exchange experiences and future directions. These events were hosted by SIRIUS, a Norwegian Centre for Research-driven Innovation to address the problems of scalable data access in the oil & gas industry, and therefore attracted participants specifically from this and supporting industries. The findings from these forums are that industrial users of OTTR share similar requirements to the construction and management of knowledge graphs that fit well with the features provided by OTTR. Summarized they need support for:

- Large-scale knowledge graph development, since ontologies can reach sizes of 100,000's of classes.
- Uniform modelling: this is necessary to ensure that access to and use of the constructed knowledge base can be performed in a uniform and predictive manner, and to reduce management and maintenance costs.
- Transformation of data sourced from different formats and schemas: the input data to create the knowledge graph that typically comes from different domain standard representations that represent similar types of data differently needs harmonization.
- Subject-matter expert (SME) involvement: SMEs need to partake in the design of the information in the knowledge graph and be able to verify its quality.
- Collaborative development environments: Development is often performed in teams, with additional stakeholders that need to interact with the results in different ways.
- Automated mechanisms for quality assessment and verification are required due to the size and complexity of the modelling and representation task.

The current tools on offer for large-scale ontology development are few. The participants of the OTTR user forums report that editors like Protégé do not meet their requirements since this mode modelling with a desktop/client application is unsuited for uniform and collaborative modelling at the scale required. Also, the fact that these tools typically operate on low-level constructs, such as OWL axioms, makes them difficult for SMEs to understand and use.

In the following, we report from selected prominent industrial and academic uses of the OTTR framework to serve as examples of its utility.

8.1 Grundfos' Industrial Ontology Engineering Platform

At Grundfos, Brynildsen et al. have built what they call the *Industrial Ontology Engineering Platform (IOEP)*, where OTTR is an integral part, "to support domain experts in using and implementing ontologies while reducing the workload for ontology engineering specialists" [6]. The tool provides web-based tabular input user interfaces that are aligned with signatures of OTTR templates prepared by ontology experts. Subject-matter experts build ontologies by populating the user interface, supported by helpful editor features such as auto-generated IRIs and metadata, label lookups and searches. The tabular data is transformed using the OTTR templates to an ontology with a build service that relies on Lutra (See Section 7.1). The deployment of the tool is reported as successful in that improves ontology engineering scalability: the reuse of OTTR templates across different ontology development projects reduces ontology design efforts and

[39] `https://www.ottr.xyz/event/2021-01-28-user-forum/`,
`https://www.ottr.xyz/event/2021-06-18-user-forum/`,
`https://www.ottr.xyz/event/2022-05-11-user-forum/`

hence the workload for ontology engineering practitioners. Also, OTTR templates have helped to position SMEs as the primary owners and contributors of new ontological models by hiding the complexity of upper-level ontology away from the SME [6].

8.2 Bosch's Ontology-Enhanced Machine Learning System

A similar approach to that of Grundfos' is followed at Bosch [55]. In their *Ontology-Enhanced Machine Learning (SemML)* system, OTTR templates are part of the *Ontology extender* component "that allows domain experts to describe domains in terms of an upper-level ontology by filling in templates. Data scientists then also use templates to annotate domain terms with quality-related information" [55]. Domain experts describe the domain by filling in simple forms that are generated to match the signature of OTTR templates. The forms produce template instances that are expanded to OWL axioms. "Templates guarantee uniformity of the updates and the consistency of the updated ontology, as well as the relative simplicity of the ontology extension process." [55] The ontology extender component is evaluated in a user study giving positive results with respect to correctness of use.

8.3 CapGemini and Norwegian Maritime Authority: Modelling Regulatory Requirements as SHACL Shapes

CapGemini and Norwegian Maritime Authority use the OTTR framework as part of a tool-chain where regulatory requirements are expressed as SHACL shapes [14]. Requirements are extracted from text using natural language processing and named entity recognition techniques. The resulting data is then transformed to RDF and SHACL using an OTTR template library for SHACL shapes.

8.4 DiProMag: Ontology of Magnetocaloric Materials

The goal of the DiProMag project is the digitization of a process chain for the production, characterization and prototypical application of magnetocaloric alloys.[40] To support this work, Blum et al. [5] have created OTTR Extension (see Section 7.3) and a template library to allow domain experts to build an ontology of magnetocaloric materials by filling in generated forms.

8.5 Aibel: Identifying Redundancies in a Large Template Library

Aibel, a global engineering company, has developed the "Material Master Data ontology", a comprehensive representation of engineering requirements and specifications that contains in the range of 100,000 classes across a hierarchy of OWL ontologies.[41] Aibel applies automated reasoning and querying to support the selection of designs, and matching designs with products [49]. The ontology is predominately generated from numerous spreadsheets, prepared by ontology experts and populated by domain experts. A custom-built pipeline translates the spreadsheets into template instances, and then into ontologies following template specifications

To evaluate the feasibility of the OTTR approach to ontology engineering and maintenance, the complete set of spreadsheet specifications was translated into a library of 1185 OTTR templates, some of which contain more than 30 parameters. Analysis of the generated template

[40] `https://www.dipromag.de/`

[41] An obfuscated version of the ontology is published at `https://github.com/Sirius-sfi/aibel-mmd-ontology` with the "intent of providing researchers and software developers with free access to a large-scale real industrial ontology".

library revealed many redundancies, largely attributable to the limited expressive power of the spreadsheet template language. Using a semi-automated approach, we were able to identify uncaptured patterns representing relevant domain conceptualizations, dramatically reducing the number of redundancies [50]. This use case shows that the expressive power of OTTR templates can bring substantial benefits, in particular for industrial domains where there is a high degree of regularity in the patterns used. It also demonstrates the usefulness of maintenance techniques of template libraries.

9 Related Work

In this section, we introduce work that is related to the OTTR framework when considering it as a *framework for pattern-based ontology engineering supported by a shared library of reusable patterns and tools for practical pattern instantiation for large-scale knowledge graph construction and maintenance*. This makes for a very wide scope of related work, so our discussion will necessarily be limited to a selection of illustrative works. Yet, there are to our knowledge few works that fall into the same description as the OTTR framework, so our presentation will cover the most relevant ones.

Pattern-based ontology engineering is closely connected to ontology design patterns (ODPs) [11, 15]. Similar to software design patterns, ODP is a modelling solution to solve a recurrent ontology design problem that is recorded and shared to simplify ontology development and use. ODPs serve the purpose of alleviating some of the difficulties involved with creating ontologies by offering reusable, best-practice building blocks and structures for ontology construction, commonly implemented and published as small OWL ontologies. Methods for combining and instantiating ODPs are described [43, 13], and a methodology for building ontologies using patterns exists [3]. ODPs are documented and published in open repositories, such as the OntologyDesignPatterns.org portal,[42] MODL [47], and the Manchester Catalogue.[43] There are tools, such as XDP [12] and CoModIDE [46], which are built on top of WebProtégé [57], as convenient tools for modelling with and instantiating ODPs. However, while ODPs are often presented as "practical building blocks" [43], we argue that ODPs in their current form, i.e., as found at OntologyDesignPatterns.org, featuring a graphical representation, a description and a "reusable OWL building block", are not practical enough, especially for the development of large ontologies. Using and adapting ODPs to a particular modelling task will normally require considerable manual work and ODPs do not describe a precise manner in which instances of the pattern may be created at scale. Frameworks that offer this functionality, in addition to the OTTR framework, are GDOL [29], Dead simple OWL design patterns (DOS-DPs) [39] and the Ontology Pre-Processor Language *OPPL* [20].

Generic DOL (GDOL) [29] is an extension of the Distributed Ontology, Modelling, and Specification Language (DOL) that supports a parametrization mechanism for ontologies. GDOL/DOL 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. The syntax for OWL in GDOL is the Manchester OWL Syntax, extended by parameters. DOL is supported by Ontohub [7] (an online ontology and specification repository) and Hets [35] (parsing and inference back-end of DOL). The difference between OTTR and GDOL is that the foundations of GDOL build on the more abstract and powerful concept of generics, rather than the simpler concept of macros. Also, GDOL is based on the comprehensive framework of DOL which supports expressions in different

[42] `http://ontologydesignpatterns.org`
[43] `http://odps.sourceforge.net/odp/html/`

logics, while OTTR is developed specifically to be used for semantic web technologies, e.g., with a type system developed for semantic web data. GDOL is more powerful than OTTR and is also therefore arguably more complex in use for the tasks that OTTR is developed for.

Dead Simple OWL Design Patterns (DOS-DP) [39] provides a simple pattern mechanism where patterns are expressed in template pattern files following YAML syntax and where property values refer to named variables. Pattern instances are created by providing filler values that correspond to the template pattern's named variables. Technically, instantiation is simply performed by a `printf` function that replaces the variable placeholder with the filler value. DOS-DP is designed to be simple and easy to use and does therefore not support inheritance or composition of patterns. The DOS-DP framework appears to be in active use; the Mondo Disease Ontology[44] is a user of DOS-DP.

The Ontology Pre-Processor Language (OPPL) [20] was originally developed as a language for manipulating OWL ontologies. Hence, 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. OPPL patterns are parameterized expressions which can be nested and can specify pattern instances and patterns directly in OWL ontologies. The syntax of OPPL is however distinct from that of RDF and OWL and requires separate tools for viewing and editing such patterns. A Protégé plugin for version 4.x exists, in addition to a tool called *Populous* [22] 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. The application focus of OPPL is somewhat different from that of OTTR: OPPL patterns are intended to be fully expanded once they are used in the ontology. In contrast, we believe that OTTR template instances can appear as instances lifted or lowered to the abstraction level suited for the given user. For instance, an ontology expert may prefer to examine an ontology formatted as a set of OWL axiom OTTR templates, while domain experts might prefer to see only the user-facing template instances. Additionally, OPPL patterns are limited to OWL expressions in Manchester syntax [17], while OTTR supports abstractions over any underlying language, although it is designed specifically for RDF. OPPL appears to no longer be actively maintained; the last release of the project was in 2013[45] the last update of its Git repository[46] was in 2020.

There are many tools for generating RDF and OWL knowledge bases from different sources using various mechanisms and mapping languages; here we mention a few that have served as inspiration for the OTTR framework. *SPARQL-generate* [32] is a template-based language and an extension of SPARQL with which one can generate RDF streams or text streams from RDF datasets and document streams in arbitrary formats. *Tawny OWL* [33] 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 and testing frameworks can be used. The *M^2 mapping language* [38] allows spreadsheet references to be used in ontology axiom patterns. Finally, *XLWrap* [31] defines a mapping language to convert spreadsheets into RDF.

[44] `https://github.com/monarch-initiative/mondo`
[45] `https://oppl2.sourceforge.net/`
[46] `https://github.com/owlcs/OPPL2`

10 Future Work

In this section, we list interesting ideas for extensions and further developments of the OTTR framework.

Support for call-by-name We wish to support *named parameters* or *call-by-name* in OTTR. This feature would increase the readability of template instances and simplify template instantiation slightly by allowing arguments to be given in arbitrary order and missing arguments to be omitted. Instances of the `o-p:NamedPizza` template using named parameters could then look like the following:

```
o-p:NamedPizza(
  pizza = ex:Margherita,
  country = ex:Italy,
  toppings = (ex:Mozzarella, ex:Tomato)
  ) .
o-p:NamedPizza(
  toppings = (ex:Potato, ex:Rosemary),
  pizza = ex:PotatoPizza
  ).
```

Support for tuple or struct types In Example 44, we see that when one needs to create a template that creates multiple instances describing objects with multiple attributes (such as the nuclear family of persons with IRIs and names), we need to zip multiple lists, each list with values for one attribute. It would be preferable to rather group the attributes of each object together in one struct or tuple, e.g., `(ex:bob, 'Bob Green', none)` with type `Tuple<owl:NamedIndividual, xsd:string, xsd:date>`, combined with functionality for extracting the tuple's elements. The `ex:NuclearFamily` template could then simply accept two lists of such tuples (instead of four lists, as defined above). Thus, extending OTTR with tuple/struct terms and types would make expressing such complex templates more convenient.

Improved dependency management The organization of templates into collections is currently done informally by placing closely related templates under the same namespace and in the same file or folder. We wish to strengthen the management of template libraries by formally characterizing the notion of template modules and allowing for explicit dependencies between modules to be expressed. The aim is to support systems to handle such template modules much like software project management systems like Maven[47] do and improve the robustness and stability of template instance expansion.

Customizable expansions A main requirement in the design of the OTTR framework is that an instance should be considered as semantically equivalent to its expansion. An enhancement to the framework would be to support different expansions of the same instance while still supporting the above requirement. A motivation for this enhancement is for instance to experiment with different representations of the same statement, or to support multiple equivalent representations of the same statement. Examples of this could be to expand instances that represent an ontology into different OWL profiles [36], or to expand a knowledge graph not only into RDF, but also other knowledge graph formats [16]. A way to support this is to introduce a more complex notion of a dataset that includes somehow instructions on what template libraries and base templates to use for expansion, and hence possibly allow a template signature to be associated with multiple different template bodies.

[47] `https://maven.apache.org/`

Test suite and benchmark The availability of multiple implementations of the OTTR framework (Section 7) raises the question of what features of the OTTR framework they support and their relative performance. We wish to establish a test suite to be able to measure the conformance of the implementations against the specifications of the OTTR framework, and a benchmark to measure their performance.

Term manipulation support The OTTR framework does not naively provide a facility for manipulating terms, such as generating IRIs by concatenating strings, or mathematical calculations. Some of this functionality is covered by the use of the query languages within bOTTR, but this is arguably not optimal. FROG [21] is a declarative term manipulation language introducing pure functions that can be applied to terms within template definitions, and leverages the OTTR type system and extends it by introducing type generics. Updating the OTTR framework to support FROG remains future work.

OTTR as query language In this paper we have seen OTTR templates being used to transform complex statements in the form of template instances into expressions over a different data format such as RDF. An interesting approach is to also allow OTTR templates to be used "in reverse" as queries and use them to extract and assemble lower-level statements to more statements at a higher level of abstraction. Using a well-designed template library to both generate and extract information would be a clear benefit in terms of usability and maintainability. OTTR as a query language has been characterized under the name Reverse OTTR [53] and a prototypical implementation exists, yet proper integration into the OTTR framework remains.

OTTR vs. Ontology-based data access (OBDA) bOTTR provides a mapping mechanism from databases to ontologies via templates, similar to what is known as ontology-based data access (OBDA) [42]. The similarities and differences between bOTTR and OBDA warrant investigation, both from a materialization and a querying perspective. Furthermore, when bOTTR is used to construct a large knowledge graph updates to the mapped sources may change, thus requiring a corresponding update to the knowledge graph, preferably without needing to reconstruct the entire graph. This is not currently supported by the OTTR framework, but preliminary work has been done by Eckhoff and Zahl [8], and incorporating this into OTTR would be desirable.

Template authoring support Developing a template library is currently manual work using text editors. Improved tool support for the existing methodology and library maintenance techniques, as well as a graphical language and graphical user interface or integrated development environment (IDE) for template authoring, would arguably increase the efficiency and quality of creating and managing template libraries and lower the bar for new users.

Static analysis of template libraries The OTTR framework supports static analysis of template libraries in the form checking type correctness. Such analysis could be extended to also consider the semantics of the vocabularies used, and for instance, identify templates or combinations of templates that can result in inconsistent OWL ontologies or unsatisfiable concepts.

Template bootstrapping To advance the development of template libraries, pattern recognition and discovery methods could be applied to identify patterns in existing knowledge bases and databases and to capture these as well-designed template libraries.

User evaluation of OTTR Many of the benefits we claim about OTTR are justified through accepted benefits of user-defined abstraction from software engineering and information modelling, yet no formal user evaluation exists. Such an evaluation could examine what parts of OTTR are actually used – and by whom, what the learning curve of OTTR is, and identify possible new enhancements to the framework.

Complexity analysis of OTTR Snilsberg et al. [54] have developed a mathematical formalization of a subset of the OTTR language, and give a preliminary report on the characterization of the theoretical size of instance expansions and decision problems associated with the language and its fragments. A rigorous analysis of the current expressivity of OTTR, versus desired or optimal expressivity, and comparison with other formalism is a clear candidate for future work, providing direction on what features one might include or omit.

What is a good template mechanism? The OTTR framework, GDOL, DOS-DP and OPPL provide different and partially overlapping functionality. There is also a proposal for powerful extensions to the OTTR language, called *Generators* and *GBoxes* [9], that are rules formulated over the OTTR templates. Kindermann et al. [24] identify characteristics that are deemed necessary for an ontology template mechanism which OTTR implements. Kindermann et al. [25] also investigate the use of ontology design patterns in practice. However, more investigation into the similarities and differences between the available pattern frameworks and experience into what is practically useful is needed to identify the correct balance between expressivity, complexity and usability, and to formulate metrics to characterize "good" template mechanisms and libraries.

11 Conclusion

This paper has given a complete overview of the current state of the OTTR framework, detailing its formal foundation, RDF and OWL adaptions, different serializations, template library management and support, mapping languages, multiple (independent) implementations, and real-world use and impact. We have also presented related and future work. We believe this paper illustrates the maturity of OTTR as a framework, a framework that has moved beyond the phase of prototypes and purely academic study, into a useful and practical framework that has a strong theoretical foundation and is based on sound engineering principles that are suitable for real-world, large-scale, maintainable ontology and knowledge graph construction.

References

1 Magnus Bakken. maplib: Interactive, literal RDF model mapping for industry. *IEEE Access*, 11:39990–40005, 2023. `doi:10.1109/ACCESS.2023.3269093`.

2 David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. RDF 1.1 turtle: Terse RDF triple language. Technical report, W3C, 2014. URL: `https://www.w3.org/TR/turtle/`.

3 Eva Blomqvist, Karl Hammar, and Valentina Presutti. Engineering ontologies with patterns - the extreme design methodology. In Pascal Hitzler, Aldo Gangemi, Krzysztof Janowicz, Adila Krisnadhi, and Valentina Presutti, editors, *Ontology Engineering with Ontology Design Patterns - Foundations and Applications*, volume 25 of *Studies on the Semantic Web*, pages 23–50. IOS Press, 2016. `doi:10.3233/978-1-61499-676-7-23`.

4 Eva Blomqvist, Pascal Hitzler, Krzysztof Janowicz, Adila Krisnadhi, Tom Narock, and Monika Solanki. Considerations regarding ontology design patterns. *Semantic Web*, 7(1):1–7, 2016. `doi:10.3233/SW-150202`.

5 Moritz Blum, Basil Ell, and Philipp Cimiano. Insights from an OTTR-centric ontology engineering methodology. In Raghava Mutharaju, Agnieszka Ławrynowicz, Pramit Bhattacharyya, Eva Blomqvist, Luigi Asprino, and Gunjan Singh, editors, *Proceedings of the 14th Workshop on Ontology Design and Patterns (WOP 2023)*, volume 3636. CEUR-WS.org, 2023. URL: `https://ceur-ws.org/Vol-3636/paper8.pdf`.

6 Mikkel Haggren Brynildsen, Claus Jakobsen, Nicolaj Abildgaard, and Caitlin Woods. Building an Industrial Ontology Engineering Platform. In *Posters, Demos, and Industry Tracks at ISWC 2023*. CEUR-WS.org, 2023. URL: `https://ceur-ws.org/Vol-3632/ISWC2023_paper_502.pdf`.

7 Mihai Codescu, Eugen Kuksa, Oliver Kutz, Till Mossakowski, and Fabian Neuhaus. Ontohub: A semantic repository engine for heterogeneous ontologies. *Appl. Ontology*, 12(3-4):275–298, 2017. `doi:10.3233/AO-170190`.

8 Magnus Wiik Eckhoff and Preben Zahl. Efficient update of OTTR-constructed triplestores. Master's thesis, University of Oslo, 2023.

9 Henrik Forssell, Christian Kindermann, Daniel P. Lupp, Uli Sattler, and Evgenij Thorstensen. Generating ontologies from templates: A rule-based approach for capturing regularity. *CoRR*, abs/1809.10436, 2018. `arXiv:1809.10436`, `doi:10.48550/arXiv.1809.10436`.

10 Henrik Forssell, Daniel P. Lupp, Martin G. Skjæveland, and Evgenij Thorstensen. Reasonable macros for ontology construction and maintenance. In Alessandro Artale, Birte Glimm, and Roman Kontchakov, editors, *Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017*, volume 1879

of *CEUR Workshop Proceedings*, 2017. URL: `https://ceur-ws.org/Vol-1879/paper34.pdf`.

11 Aldo Gangemi and Valentina Presutti. *Ontology Design Patterns*, pages 221–243. Springer, 2009. `doi:10.1007/978-3-540-92673-3_10`.

12 Karl Hammar. Ontology design patterns in webprotege. In Serena Villata, Jeff Z. Pan, and Mauro Dragoni, editors, *Proceedings of the ISWC 2015 Posters & Demonstrations Track, 2015*, volume 1486 of *CEUR Workshop Proceedings*, 2015. URL: `https://ceur-ws.org/Vol-1486/paper_50.pdf`.

13 Karl Hammar and Valentina Presutti. *Template-Based Content ODP Instantiation*, volume 32 of *Studies on the Semantic Web*, pages 1–13. IOS Press, 2016. `doi:10.3233/978-1-61499-826-6-1`.

14 Veronika Heimsbakk and Kristian Torkelsen. Using the shapes constraint language for modelling regulatory requirements, 2023. `doi:10.48550/arXiv.2309.02723`.

15 Pascal Hitzler, Aldo Gangemi, Krzysztof Janowicz, Adila Krisnadhi, and Valentina Presutti, editors. *Ontology Engineering with Ontology Design Patterns - Foundations and Applications*, volume 25 of *Studies on the Semantic Web*. IOS Press, 2016.

16 Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Comput. Surv.*, 54(4):71:1–71:37, 2022. `doi:10.1145/3447772`.

17 Matthew Horridge and Peter F. Patel-Schneider. OWL 2 Web Ontology Language Manchester Syntax. W3c working group note, W3C, 2012. URL: `https://www.w3.org/TR/owl2-manchester-syntax/`.

18 Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. W3c member submission, W3C, 2004. URL: `http://www.w3.org/Submission/SWRL/`.

19 Bernadette Hyland, Ghislain Atemezing, and Boris Villazón-Terrazas. Best practices for publishing linked data. W3c working group note, W3C, 2014. URL: `https://www.w3.org/TR/ld-bp/`.

20 Luigi Iannone, Alan L. Rector, and Robert Stevens. Embedding Knowledge Patterns into OWL. In *ESWC*, pages 218–232, 2009. `doi:10.1007/978-3-642-02121-3_19`.

21 Marlen Jarholt. Frog: Functions for ontologies—an extension for the OTTR-framework. Master's thesis, University of Oslo, 2022.

22 Simon Jupp et al. Populous: a tool for building OWL ontologies from templates. *BMC Bioinformatics*, 13(S-1):S5, 2012. `doi:10.1186/1471-2105-13-S1-S5`.

23 C. M. Keet. *An Introduction to Ontology Engineering.* College Publications, 2018.

24 Christian Kindermann, Daniel P. Lupp, Martin G. Skjæveland, and Leif Harald Karlsen. Formal relations over ontology patterns in templating frameworks. In Eva Blomqvist, Torsten Hahmann, Karl Hammar, Pascal Hitzler, Rinke Hoekstra, Raghava Mutharaju, María Poveda-Villalón, Cogan Shimizu, Martin G. Skjæveland, Monika Solanki, Vojtech Svátek, and Lu Zhou, editors, *Advances in Pattern-Based Ontology Engineering, extended versions of the papers published at the Workshop on Ontology Design and Patterns (WOP)*, volume 51 of *Studies on the Semantic Web*, pages 120–133. IOS Press, 2021. `doi:10.3233/SSW210010`.

25 Christian Kindermann, Bijan Parsia, and Uli Sattler. Detecting influences of ontology design patterns in biomedical ontologies. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I*, volume 11778 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2019. `doi:10.1007/978-3-030-30793-6_18`.

26 Johan W. Klüwer, Martin G. Skjæveland, and Magne Valen-Sendstad. ISO 15926 templates and the Semantic Web. Technical report, W3C, 2008. W3C Workshop on Semantic Web in Oil & Gas Industry.

27 Graham Klyne, Jeremy J. Carroll, and Brian McBride. RDF 1.1 Concepts and Abstract Syntax. W3c recommendation, W3C, 2014. URL: `https://www.w3.org/TR/rdf11-concepts/`.

28 Holger Knublauch and Dimitris Kontokostas. Shapes Constraint Language (SHACL). W3c recommendation, W3C, 2017. URL: `https://www.w3.org/TR/shacl/`.

29 Bernd Krieg-Brückner and Till Mossakowski. Generic ontologies and generic ontology design patterns. In Eva Blomqvist, Óscar Corcho, Matthew Horridge, David Carral, and Rinke Hoekstra, editors, *Proceedings of the 8th Workshop on Ontology Design and Patterns (WOP), 2017*, volume 2043 of *CEUR Workshop Proceedings*, 2017. URL: `https://ceur-ws.org/Vol-2043/paper-02.pdf`.

30 Markus Krötzsch, Denny Vrandecic, and Max Völkel. Semantic mediawiki. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*, pages 935–942. Springer, Springer, 2006. `doi:10.1007/11926078_68`.

31 Andreas Langegger and Wolfram Wöß. Xlwrap - querying and integrating arbitrary spreadsheets with SPARQL. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, volume 5823 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2009. `doi:10.1007/978-3-642-04930-9_23`.

32 Maxime Lefrançois, Antoine Zimmermann, and Noorani Bakerally. A SPARQL extension for generating RDF from heterogeneous formats. In Eva Blomqvist, Diana Maynard, Aldo Gangemi, Rinke Hoekstra, Pascal Hitzler, and Olaf Hartig, editors, *The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part I*, volume 10249 of *Lecture Notes in Computer Science*, pages 35–50, Portoroz, Slovenia, May 2017. doi:10.1007/978-3-319-58068-5_3.

33 Phillip Lord. The semantic web takes wing: Programming ontologies with tawny-owl. In Mariano Rodriguez-Muro, Simon Jupp, and Kavitha Srinivas, editors, *Proceedings of the 10th International Workshop on OWL: Experiences and Directions (OWLED), 2013*, volume 1080 of *CEUR Workshop Proceedings*, 2013. URL: https://ceur-ws.org/Vol-1080/owled2013_16.pdf.

34 Daniel P. Lupp, Melinda Hodkiewicz, and Martin G. Skjæveland. Template libraries for industrial asset maintenance: A methodology for scalable and maintainable ontologies. In Thorsten Liebig, Achille Fokoue, and Zhe Wu, editors, *Proceedings of the 12th International Workshop on Scalable Semantic Web Knowledge Base Systems co-located with 19th International Semantic Web Conference (ISWC 2020), Athens, Greece, November 2, 2020*, volume 2757 of *CEUR Workshop Proceedings*, pages 49–64. CEUR-WS.org, 2020. URL: https://ceur-ws.org/Vol-2757/SSWS2020_paper4.pdf.

35 Till Mossakowski, Christian Maeder, and Klaus Lüttich. The heterogeneous tool set, hets. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer, 2007. doi:10.1007/978-3-540-71209-1_40.

36 Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web Ontology Language Profiles (Second Edition). W3c recommendation, W3C, 2012. URL: http://www.w3.org/TR/owl-profiles.

37 Mark A. Musen. The protégé project: a look back and a look forward. *AI Matters*, 1(4):4–12, 2015. doi:10.1145/2757001.2757003.

38 Martin J. O'Connor, Christian Halaschek-Wiener, and Mark A. Musen. M^2: A language for mapping spreadsheets to OWL. In Evren Sirin and Kendall Clark, editors, *Proceedings of the 7th International Workshop on OWL: Experiences and Directions (OWLED), 2010*, volume 614 of *CEUR Workshop Proceedings*, 2010. URL: https://ceur-ws.org/Vol-614/owled2010_submission_17.pdf.

39 David Osumi-Sutherland, Mélanie Courtot, James P. Balhoff, and Christopher J. Mungall. Dead simple OWL design patterns. *J. Biomed. Semant.*, 8(1):18:1–18:7, 2017. doi:10.1186/s13326-017-0126-0.

40 Bijan Parsia, Peter Patel-Schneider, and Boris Motik. Owl 2 web ontology language structural specification and functional-style syntax. W3c recommendation, W3C, 2012. URL: https://www.w3.org/TR/owl2-syntax/.

41 Peter F. Patel-Schneider and Boris Motik. OWL 2 Web Ontology Language Mapping to RDF Graphs. W3c recommendation, W3C, 2012. URL: https://www.w3.org/TR/owl-mapping-to-rdf/.

42 Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *J. Data Semantics*, 10:133–173, 2008. doi:10.1007/978-3-540-77688-8_5.

43 Valentina Presutti and Aldo Gangemi. Content ontology design patterns as practical building blocks for web ontologies. In Qing Li, Stefano Spaccapietra, Eric S. K. Yu, and Antoni Olivé, editors, *Conceptual Modeling - ER 2008, 27th International Conference on Conceptual Modeling, Barcelona, Spain, October 20-24, 2008. Proceedings*, volume 5231 of *LNCS*, pages 128–141. Springer, 2008. doi:10.1007/978-3-540-87877-3_11.

44 Leo Sauermann and Richard Cyganiak. Cool uris for the semantic web. Technical report, W3C, 2008. URL: http://www.w3.org/TR/cooluris/.

45 Ralph Schäfermeier, Adrian Paschke, and Heinrich Herre. Ontology design patterns for representing context in ontologies using aspect orientation. In Krzysztof Janowicz, Adila Alfa Krisnadhi, María Poveda Villalón, Karl Hammar, and Cogan Shimizu, editors, *Proceedings of the 10th Workshop on Ontology Design and Patterns (WOP 2019) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 27, 2019*, volume 2459 of *CEUR Workshop Proceedings*, pages 32–46. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2459/paper3.pdf.

46 Cogan Shimizu and Karl Hammar. Comodide - the comprehensive modular ontology engineering IDE. In Mari Carmen Suárez-Figueroa, Gong Cheng, Anna Lisa Gentile, Christophe Guéret, C. Maria Keet, and Abraham Bernstein, editors, *Proceedings of the ISWC 2019 Satellite Tracks (Posters & Demonstrations, Industry, and Outrageous Ideas) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 26-30, 2019*, volume 2456 of *CEUR Workshop Proceedings*, pages 249–252. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2456/paper65.pdf.

47 Cogan Shimizu, Quinn Hirt, and Pascal Hitzler. MODL: A modular ontology design library. In Krzysztof Janowicz, Adila Alfa Krisnadhi, María Poveda-Villalón, Karl Hammar, and Cogan Shimizu, editors, *Proceedings of the 10th Workshop on Ontology Design and Patterns (WOP 2019) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 27, 2019*, volume 2459 of *CEUR Workshop Proceedings*, pages 47–58. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2459/paper4.pdf.

48 Martin G. Skjæveland, Henrik Forssell, Johan W. Klüwer, Daniel P. Lupp, Evgenij Thorstensen, and Arild Waaler. Pattern-based ontology design

and instantiation with reasonable ontology templates. In Eva Blomqvist, Óscar Corcho, Matthew Horridge, David Carral, and Rinke Hoekstra, editors, *Proceedings of the 8th Workshop on Ontology Design and Patterns (WOP 2017) co-located with the 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 21, 2017*, volume 2043 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL: https://ceur-ws.org/Vol-2043/paper-04.pdf.

49 Martin G. Skjæveland, Anders Gjerver, Christian M. Hansen, Johan Wilhelm Klüwer, Morten R. Strand, Arild Waaler, and Per Øyvind Øverli. Semantic Material Master Data Management at Aibel. In *Proceedings of the ISWC 2018 Industry Track*, volume 2180 of *CEUR Workshop Proceedings*, 2018. URL: https://ceur-ws.org/Vol-2180/paper-90.pdf.

50 Martin G. Skjæveland, Daniel P. Lupp, Leif Harald Karlsen, and Henrik Forssell. Practical ontology pattern instantiation, discovery, and maintenance with reasonable ontology templates. In Denny Vrandecic, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web - ISWC 2018*, volume 11136 of *LNCS*, pages 477–494. Springer, 2018. doi:10.1007/978-3-030-00671-6_28.

51 Martin G. Skjæveland, Daniel P. Lupp, Leif Harald Karlsen, and Johan W. Klüwer. OTTR: formal templates for pattern-based ontology engineering. In Eva Blomqvist, Torsten Hahmann, Karl Hammar, Pascal Hitzler, Rinke Hoekstra, Raghava Mutharaju, María Poveda-Villalón, Cogan Shimizu, Martin G. Skjæveland, Monika Solanki, Vojtech Svátek, and Lu Zhou, editors, *Advances in Pattern-Based Ontology Engineering, extended versions of the papers published at the Workshop on Ontology Design and Patterns (WOP)*, volume 51 of *Studies on the Semantic Web*, pages 349–377. IOS Press, 2021. doi:10.3233/SSW210025.

52 Martin G. Skjæveland. *The Core OTTR Template Library*, volume 51 of *Studies on the Semantic Web*, chapter 23, pages 378–393. IOS Press, 2021. doi:10.3233/SSW210026.

53 Erik Snilsberg. Reverse OTTR: A query language for RDF. Master's thesis, University of Oslo, 2022.

54 Erik Snilsberg, Leif Harald Karlsen, Egor V. Kostylev, and Martin G. Skjæveland. Foundations of ontology template language OTTR (extended abstract). In Laura Giordano, Jean Christoph Jung, and Ana Ozaki, editors, *Proceedings of the 37th International Workshop on Description Logics (DL 2024), Bergen, Norway, June 18-21, 2024*, volume 3739 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2024. URL: https://ceur-ws.org/Vol-3739/abstract-24.pdf.

55 Yulia Svetashova, Baifan Zhou, Tim Pychynski, Stefan Schmidt, York Sure-Vetter, Ralf Mikut, and Evgeny Kharlamov. Ontology-enhanced machine learning: A bosch use case of welding quality monitoring. In Jeff Z. Pan, Valentina A. M. Tamma, Claudia d'Amato, Krzysztof Janowicz, Bo Fu, Axel Polleres, Oshani Seneviratne, and Lalana Kagal, editors, *The Semantic Web - ISWC 2020 - 19th International Semantic Web Conference, Athens, Greece, November 2-6, 2020, Proceedings, Part II*, volume 12507 of *Lecture Notes in Computer Science*, pages 531–550, Berlin, Heidelberg, 2020. Springer. doi:10.1007/978-3-030-62466-8_33.

56 Tania Tudorache. Ontology engineering: Current state, challenges, and future directions. *Semantic Web*, 11(1):125–138, December 2020. doi:10.3233/SW-190382.

57 Tania Tudorache, Csongor Nyulas, Natalya Fridman Noy, and Mark A. Musen. Webprotégé: A collaborative ontology editor and knowledge acquisition tool for the web. *Semantic Web*, 4(1):89–99, 2013. doi:10.3233/SW-2012-0057.

58 Denny Vrandecic. Explicit knowledge engineering patterns with macros. In *Proceedings of the Ontology Patterns for the Semantic Web Workshop at the 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005. Ed.: Chris Welty*. Galway, 2005.