Extensions of Generic DOL for Generic Ontology Design Patterns

Mihai CODESCU [a,1], Bernd KRIEG-BRÜCKNER [a,b] and Till MOSSAKOWSKI [c]

[a] *University of Bremen, Collaborative Research Center EASE, Bremen, Germany*

[b] *German Research Center for Artificial Intelligence (DFKI), CPS, Bremen, Germany*

[c] *Institute for Intelligent Cooperating Systems, Faculty of Computer Science, Otto-von-Guericke University of Magdeburg, Germany*

Abstract. Generic ontologies were introduced as an extension (*Generic* DOL) of the *Distributed Ontology, Modeling and Specification Language*, DOL, with the aim to provide a language for Generic Ontology Design Patterns. In this paper we present a number of new language constructs that increase the expressivity and the generality of Generic DOL, among them sequential and optional parameters, list parameters with recursion, and local sub-patterns. These are illustrated with non-trivial patterns: generic value sets and (nested) qualitatively graded relations, demonstrated as definitional building blocks in an application domain.

Keywords. ontology design patterns, recursive pattern definition, generic ontologies, generic DOL, qualitatively graded relations

1. Introduction

Ontology design patterns (ODPs) [1] have been introduced as a means to establish best practices for ontology design as well as a way to provide a set of carefully-designed building blocks for ontologies that may be reused in different contexts.

Several languages for representing ODPs, their instantiations and the relationships between them have been proposed. The OPLa language [2] makes use of OWL annotation properties to mark patterns and their relationships. OntoUML [3] has been extended with a pattern language based on graph transformations in [4]. OTTR [5] is a language for representing patterns as parameterized ontologies, instantiated via macro expansion.

With the exception of OTTR, in these languages ODPs are ontologies themselves. In [6], we introduced *generic* ontology design patterns (GODPs), using the language Generic DOL as an extension of the *Distributed Ontology, Modeling and Specification Language*, DOL [7]. DOL is a meta-language that enables modular development of ontologies[2] and allows specification of intended relationships (e.g. theory interpretation,

---

[1] This work has been partially supported by the German Research Foundation, DFG, as part of the Collaborative Research Center (Sonderforschungsbereich) 1320 "EASE - Everyday Activity Science and Engineering" (`http://www.ease-crc.org/`). Corresponding Author's E-mail: codescu@uni-bremen.de

[2] We ignore the modeling and specification aspects here and focus on ontologies only.

alignment, properties of extensions) between them. DOL is supported by the *Heterogeneous Tool Set*, Hets [8], that provides a parser for DOL specifications, an implementation of DOL semantics, and an interface to theorem provers. In the proposed extension of DOL, GODPs have *parameters* that may be *instantiated* in different ways, thus leading to an even greater and easier re-use of ODPs. Unlike OTTR, where parameters are just symbols of a certain kind, or lists of these, in Generic DOL parameters are ontologies themselves, capturing abstract properties that are expected to hold for the argument ontology in an instantiation.

Further examples of GODPs formulated in Generic DOL can be found in [9], in particular a role pattern from the literature, reformulated in a modular and reusable way. [9] also gives more motivation and describes the advantages of Generic ODPs over "classical" ODPs, i.e. of parameterization over subsumption (parametric polymorphism over subtype polymorphism, resp.).

[2] introduces a list of desired capabilities for a language for ODPs. We here list these, and will come back to how they are met by Generic DOL in the next sections:

C1. Compatibility with the OWL standard and OWL supporting tools.

C2. Support for identification of ODPs as distinct from ontologies, and identification of relevant parts of ODPs.

C3. Support for representing relevant relationships between patterns (refinement, generalization etc.)

C4. Support for identification of modules in ontologies generated using an ODP-based approach.

C5. Support for representing relationships between ontology modules and the ODPs that have been used as templates for these modules.

C6. Extensibility of the language by means of community-provided patterns for representing information about patterns and modules.

This paper is an update of [6], addressing many of the extensions of Generic DOL listed there as future work, on which we focus in this paper. To better illustrate the degree of generality provided by the new language extensions, we deliberately decided to present some of the examples used in [6] again in their new form. The semantics of Generic DOL follows the semantics of generic specifications in CASL [10]. Here we refrain from presenting the semantics of the new language constructs in full formal detail.

2. Generic DOL

Unions and Extensions in DOL. The building blocks of DOL are (basic) ontologies written as-is in existing ontology languages like OWL, inheriting its semantics. DOL also provides a construct for *uniting* ontologies, written O_1 **and** O_2, and one for *extending* an existing ontology O_1 by new declarations and axioms, written O_1 **then** O_2 (in this case, O_2 may be an ontology fragment that is only well-formed in the context of O_1, which is called the *local environment* for O_2). When all basic ontologies are written in OWL, this has the same expressivity as OWL imports.

```
pattern ReflexiveRelation
 [ObjectProperty: r]
 [Class: C] =
 ObjectProperty: r
  Domain: C  Range: C
  Characteristics:
    Reflexive
```

```
pattern TransitiveRelation
 [ObjectProperty: r]
 [Class: C] =
 ObjectProperty: r
  Domain: C  Range: C
  Characteristics:
    Transitive
```

```
pattern InverseRelation
 [ObjectProperty: r]
 [Class: D][Class: R]
 [ObjectProperty: ir] =
 ObjectProperty: r
  Domain: D  Range: R
 ObjectProperty: ir
  Domain: R  Range: D
  InverseOf: r
```

```
pattern SubProp
 [ObjectProperty: q]
 [Class: D][Class: R]
 [ObjectProperty: p
  Domain: D  Range: R] =
 ObjectProperty: q
  Domain: D  Range: R
  SubPropertyOf: p
```

Figure 1. `ReflexiveRelation`, `TransitiveRelation`, `InverseRelation` and `SubProp`.

Renaming. *Renaming* extends OWL's expressiveness: the notation O **`with`** σ means that O is translated along a signature morphism σ (mapping O's symbols to possibly different ones). This construct is particularly useful for avoiding unintended name clashes in unions of ontologies.

Unions, extensions and renamings can be used at a small-scale level, that is, within one DOL document, in contrast to OWL's import construct, which may only be used to import other documents.

The naming of ontologies within one document, written **`ontology`** ON = O, gives the name ON to some structured or unstructured DOL ontology O.

Generic DOL. The language *Generic* DOL (or GDOL) was proposed in [6] as an extension of DOL with parameterized ontologies (or generic ontologies, or patterns, for short), following generic specifications in CASL [10], and used for writing GODPs. Parameters are ontologies themselves, thus providing important semantic expressivity. A generic ontology may import ontologies, written after the list of parameters with the keyword **`given`**; the symbols of these ontologies are visible in the parameters and the body, but will not be instantiated. Thus, a generic ontology is written **`pattern`** `PN[`$\mathtt{P}_1$`]...[`$\mathtt{P}_k$`]` **`given`** `I = B`, where $\mathtt{P}_1, \ldots \mathtt{P}_k$ are the parameters of the pattern `PN`, `I` is the imported ontology, and `B` the pattern body.

The syntax for *instantiations* is `PN[`$\mathtt{A}_1$ **`fit`** σ_1`]...[`$\mathtt{A}_k$ **`fit`** σ_k`]`, where $\mathtt{A}_1, \ldots \mathtt{A}_k$ are the argument ontologies of the instantiation, and for each i, σ_i are symbol mappings, written `f |-> a` where `f` and `a` are symbols of the parameter $\mathtt{P}_i$ and argument $\mathtt{A}_i$, resp. The sequence of mapping items gives rise to a signature morphism between the parameter and the argument, called *fitting morphism*. The semantics of instantiation is a DOL ontology. Note that patterns are written using a different keyword than ontologies, and thus **C2** is met.

Several simple examples of generic ontology patterns are presented in Fig. 1, where we introduce patterns for very basic building blocks for ontologies, which should be self-explanatory. The pattern `SubProp` already makes use of a new language construct of Generic `DOL` (cf. the next section): it has, as fourth parameter, an ontology that contains two additional axioms involving symbols from the previous parameters.

Just as for OWL (and `DOL`), the *"Same Name – Same Thing" principle* is used, which means that the definition of an entity may be repeated without introducing multiple occurrences of that entity. For Generic `DOL`, this means that if the body of a GODP declares an entity, the union of multiple instantiations of that GODP will contain only one occurrence of that entity. If this was not the intention, the entity should rather become a parameter of the GODP, such that each instantiation can assign it a different name.

`DOL`'s notion of refinement: **`refinement`** `R:` O_1 **`refined via`** σ **`to`** O_2 expresses that O_1 may be refined to (or is a generalization of) O_2 via the signature morphism σ. Refinements have a formal semantics and can be easily extended to parameterized ontologies over the same parameters; this addresses **C3**.

3. Extensions of Generic `DOL`

In the interest of simplicity (of writing and reading), parameters of generic ontologies should be kept as small as needed. The aim is to avoid having to explicitly provide symbol mappings when making instantiations of generic ontologies: if a parameter and its corresponding argument consist each of only one symbol, `Hets` will automatically derive the unique way of mapping the one to the other. To make the notation more compact, the parameters of a GODP and the arguments in an instantiation may be written in Generic `DOL` as a semi-colon separated list, e.g. `TransitiveRelation[olderThan;Person]` for `TransitiveRelation[olderThan][Person]`.[3]

Sequential semantics of parameters. The first significant extension of Generic `DOL` that we introduce is a modification of the semantics of generic ontologies. In the semantics of generics in [10,6], each parameter forms its own environment. In the context of keeping the parameters small, we decided to allow each parameter to share the environment of all parameters preceding it along a chain of inclusions. We call this *sequential semantics* for parameters of generic ontologies.

As an example, the GODP `SubProp` in Fig. 1 takes as parameters an object property `q`, two classes `D` and `R`, and finally an ontology extending the previous parameters with the declaration of an object property `p` with domain `D` and range `R`; its body adds axioms that domain and range of `q` are also `D` and `R`, respectively, and moreover `q` is a subproperty of `p`. Note that with the sequential semantics it has become possible to refer to `D` and `R` in the axioms for `p`; the effect of including the domain and range axioms for the parameter `p` now allows (indeed requires) the checking of these as constraints on its argument in each instantiation, as we shall see below.

---

[3] For OWL we may use semicolons as separators between parameters and arguments because they are not used as separators between declarations at the basic level. When this is the case, e.g. for the CASL logic, where one may write `sort s; op c : s`, we use curly brackets to mark the begin and the end of an ontology. Thus, `G[sort s; sort t]` will be parsed as `G[{sort s; sort t}]` – an ontology declaring two sorts, while a generic ontology with two sorts as parameters or arguments will be written `H[{sort s};{sort t}]`.

The semantics of instantiation in [10,6] imposes a compatibility condition between the fitting morphisms for the different parameters: if a symbol occurs in multiple formal parameters, it must be mapped by the different fitting morphisms in a unique way. This compatibility condition remains in our extension, and the user can rely on it to provide symbol mappings only for the new symbols of a parameter; by compatibility, the way the old symbols are mapped is already defined.

To illustrate this, let us assume we want to define the `isAncestorOf` property between `Person`s as a transitive relation and with subproperty `isParentOf` (cf. Fig. 2). We first instantiate the pattern `TransitiveRelation` to obtain that `isAncestorOf` is transitive and has domain and range `Person`. We would like to write `isAncestorOf` as a shorthand notation for the ontology `TransitiveRelation[isAncestorOf;Person]` as the last argument of `SubProp`, or, to be fully correct, as this ontology has more than one object property, as a shorthand notation for the even longer form where this is followed by `fit p |-> isAncestorOf`.

Note that Fig. 2 also expresses that ontology `PersonRels` has been formed using patterns `TransitiveRelation` and `SubProp`, showing that GDOL meets **C4** above. Also **C5** is met, because the relation between the ODPs (here `TransitiveRelation` and `SubProp`) and the resulting ontology (here `PersonRels`), in other words, the relation between the GODPs and one of their instantiations, becomes visible and gives rise to theorem links (=theory interpretations) in Hets' development graphs.

```
ontology PersonRels =
 TransitiveRelation [isAncestorOf; Person]
then SubProp[isParentOf; Person; Person; isAncestorOf]
```

Figure 2. Using the patterns for a concrete design.

Local environments and compact notation for arguments. This requires another language extension regarding instantiation of parameters of generic ontologies. Firstly, in the case of some DOL ontology `O1` followed by an instantiation, written `O1 then G[AP1]`, the *local environment* `O1` of previous declarations that is being extended is implicitly added to the argument, i.e. this expands to `O1 then G[O1 then AP1]`. In the case of ontologies with imports, the local environment of an instantiation will include them. Secondly, we introduce a shorthand notation for the instantiation of those parameters that define only one new symbol (recall that we assume sequential semantics of parameters, thus the symbols of all previous parameters are visible at each step). Consider that the name of this unique new symbol of a parameter is `N` and its kind (class, object property, etc.) is *k*. For an instantiation of that parameter, in [10,6] an ontology is required as an argument, which may be given in two forms:

- as a named ontology `O`. Then we must be able to derive uniquely how `N` is mapped to a symbol of kind *k* in `O`, otherwise we must explicitly provide a symbol mapping of the form `N |-> N'` where `N'` is a symbol of kind *k* in `O`.
- as an anonymous ontology consisting of a sequence of symbol declarations and axioms. A special case is that of a single symbol defined with an explicit kind; then this kind must also be *k*. In such a case, this unique symbol is considered newly declared and acts as an argument, and the symbol mapping is uniquely determined.

We propose a third option here:

- the name `M` of a symbol of kind k from the local environment `E` is passed as an argument. The argument expands to `E` **`fit`** `N |-> M`. Thus any properties that `N` must have, as specified in the parameter, are checked for the symbol `M` in the local environment.

In general, an instantiation `SubProp[sr;A;B;r]` of `SubProp` where `A`, `B` are classes and `sr`, `r` are relations from the local environment, can be done only if `r` has domain `A` and range `B`. The result is that `sr` becomes a sub-property of `r` and moreover it gets domain `A` and range `B` (if this was not already available in the local environment). An instantiation `SubProp[sr;A;B;r]` where `sr` is not visible in the local environment also requires that `r` has domain `A` and range `B`. The result is that a new relation `sr` is defined and added to the local environment (again, with domain `A` and range `B`, and as a sub-property of `r`).

In our example, we may then write `isAncestorOf` as an argument for the fourth parameter of `SubProp`; with the third case listed above, this means that we refer to the symbol declared in the instantiation of `TransitiveRelation`, and therefore the expected domain and range axioms hold for `isAncestorOf`.

`OrderRelationExtension` in Fig. 3 makes use of the simple patterns defined in Fig. 1 to extend a simple order (the transitive `greater[Val]` relation on a class `Val`): we define its inverse `less[Val]`, a `greaterOrEqual[Val]` and a `lessOrEqual[Val]` relation that are inverse to each other such that `less[Val]` is a sub-property of `lessOrEqual[Val]` and `greater[Val]` is a sub-property of `greaterOrEqual[Val]`.

Parameterized names. Here the symbols declared in the two patterns have *parameterized names*, to make explicit that they depend on the names of the parameters. The notation for parameterized names is $Name[P_1,...,P_N]$, if the name of the new symbol depends on N parameters. During instantiation, the names of the arguments are substituted in the parameterized name, e.g. `greater[Val]` becomes `greater[Significance]` if the value provided for `Val` is `Significance`. Hets also offers the possibility of *stratifying* these names for the result of an instantiation: the name `greater[Significance]` is replaced with `greater_Significance`, thus obtaining a legal OWL identifier. This is the final step needed to make an ontology involving the instantiation of one or several Generic DOL patterns compatible with the OWL standard, as the DOL constructs can be *flattened*: a structured DOL ontology is replaced with an equivalent unstructured one. [4] Thus Generic DOL fulfills **C1**.

As an argument of `OrderRelationExtension` we could provide any transitive relation, in particular, a strict order. Since OWL does not support transitive and asymmetric relations, the argument would have to be given in a logic where this can be expressed, e.g. OWL without restrictions [11] or first-order logic. The theory presented informally in this paper is actually independent of the underlying formalism used for writing ontologies (OWL in the examples here) and moreover provides support for heterogeneous specifications as in the above example: the parameter may be instantiated with an argument in another logic along an encoding of the logic of the parameter to the logic of the argument.

---

[4] This is true for the DOL structuring constructs presented here, but not for all of DOL, see [7].

```
pattern SimpleOrder [Class: C] =
 TransitiveRelation[greater[C]; C]


pattern OrderRelationExtension
 [Class: Val; TransitiveRelation[greater[Val]; Val] ]
= %% greater[Val] + greaterOrEqual[Val],less[Val],lessOrEqual[Val]
     InverseRelation[less[Val];          Val;Val;greater[Val]]
 and InverseRelation[greaterOrEqual[Val];Val;Val;lessOrEqual[Val]]
 and ReflexiveRelation[lessOrEqual[Val]; Val]
then TransitiveRelation[less[Val];           Val]
 and TransitiveRelation[greaterOrEqual[Val]; Val]
 and TransitiveRelation[lessOrEqual[Val];    Val]
 and SubProp[greater[Val]; Val; Val; greaterOrEqual[Val]]
 and SubProp[less[Val];    Val; Val; lessOrEqual[Val]]
```

Figure 3. `SimpleOrder` and `OrderRelationExtension`.

Local sub-patterns. A pattern may be structured into smaller sub-patterns; often we want to make these visible only in the pattern where they are introduced. For this, we allow *local definition of sub-patterns* before the body of a GODP, using a `let` notation. The local sub-patterns share the parameters of the main pattern where they are defined. Note that this considerably abbreviates the notation; in effect, it corresponds to a partial instantiation of a corresponding pattern declared outside of the body (cf. [6]). The body of the main pattern may, and in most cases will, make use of instantiations of the local sub-patterns.

Optional parameters. We may mark parameters as *optional*, written `?[FP]` (as in OTTR [5]), where `FP` is a parameter, or `[ ...; ? FP; ...]` in the notation with semicolons. At instantiation, if an argument is not provided for an optional parameter (written `[]` or as a whitespace between semicolons `; ;`), all occurrences of that parameter in the body are replaced with the empty pattern, and all symbols and sentences containing symbols from that parameter are removed.

List parameters and recursion. We also introduce language constructs for *list parameters*, in spirit similar to those in OTTR [5]. While OTTR patterns support only iteration and zip over list parameters, we allow recursive calls of patterns over lists in Generic `DOL`, which would be considered illegal in OTTR because they introduce cyclic dependencies between patterns. A list is written `X :: Xs`, where `X` is an ontology and `Xs` denotes the tail of the list. If `X` is an ontology declaring only one symbol of a certain kind, it is assumed that all the ontologies `Xs` are of the same form. We may refer to such a list as a list of symbols of that kind. For example, `Class: C :: Cs` is a list of ontologies each consisting only of a single class declaration. A pattern with such a list as a parameter is written **`pattern`** `G [Class: C :: Cs] = ...`. The empty list is written `[`**`empty`**`]` and is treated as an empty optional argument.

Notations. In the argument of an instantiation of a GODP `G`, we may write

```
pattern ValSet [Class: Val; Individual: v0 :: vS;
                 ? ObjectProperty: greater       ]
= %% all individuals vi from v0::vS become members of Val
let pattern OrderStep [Individual: vi; Individual: vj :: vS] =
     Individual: vj Types: Val Facts: greater vi
    then OrderStep[vj; vS]
in   Individual: v0 Types: Val
then SimpleOrder[greater;Val] and OrderStep[v0; vS]
then { DifferentIndividuals: {v0 :: vS}
       Class: Val EquivalentWith: {v0 :: vS} }
```

Figure 4. `ValSet`.

```
ontology ValSet_CrustStyle =
 ValSet[CrustStyle;
        [bottomCrust,
         topCrust,
         singleCrust,
         twoCrust,
         turnoverCrust,
         strudelCrust ];
        %% no order
       ]
```

```
ontology ValSet_Significance =
 ValSet[Significance;
        [0Insignificant,
         1Subordinate,
         2Essential,
         3Dominant     ];
        greater[Significance] ]
```

Figure 5. Instantiations of `ValSet`.

- `[]` for `[`**`empty`**`]`,
- `[X]` for `[X::`**`empty`**`]`, and
- [X_1, …, X_n] for [X_1 :: … :: X_n :: **empty**].

Value Sets. Qualitative values, corresponding to abstractions from quantitative data, occur quite often in practice, cf. grading below. As we know from cognitive science, they are related to the human need for doing away with irrelevant detail (precision in this case); here (and there) they allow us to simplify abstract reasoning (cf. [12]).

With the new constructions introduced above, the pattern `ValSet` (Fig. 4) has as arguments: a class of values, a list of value individuals, and an optional relation between these values. The sub-pattern `OrderStep` introduces the fact that a value belongs to the set of values and is optionally greater than the value introduced in the set at the previous step. Once the list `vS` is empty, the recursion stops. All this is put together in the body of `ValSet`: the value is created for the first element of the list of value individuals, the relation `greater` is defined to be a simple order on `Val`, the iteration creates the rest of the values, and finally the values are declared to be different from each other and the set of values is defined to be the disjoint union of all values.

The optional parameter for `ValSet` allows to create instances of this pattern both for the case when the values are ordered (`ValSet_Significance` in Fig. 5), and for the case when the values in the set are not ordered (`ValSet_CrustStyle` in Fig. 5). The

```
pattern ValSetWithOrder[Class: Val; Individual: v :: Vs] =
 OrderRelationExtension[Val; ValSet[Val; v::Vs; SimpleOrder[Val]]]
```

Figure 6. Extending the order on `ValSet`.

expansion of `ValSet_Significance` is precisely the pattern `GradedRelations4Exp` in Fig. 3 of [6]. We may also extend the order relation `greater[Val]` on the value set with its inverse `less[Val]`, its reflexive version `greaterOrEqual[Val]` and the inverse of its reflexive version `lessOrEqual[Val]`, as illustrated in Fig. 6.

Graded Relations. In [6] we introduced a pattern for graded relations with a grade domain with 4 values and stated that analogous patterns must be provided for each number of values. The main idea of the pattern [13] is to introduce a qualitative metric, arbitrarily fine and usually represented as an ordered set, for an object property. Typical examples include the significance of an ingredient in a recipe, or how much a person is affected by an impairment. Instead of using reification for the ternary relation thus obtained, the solution proposed in [13] is to encode the grading with a sheaf of relations, one for each grade, using an encoding by parameterized names. The intended meaning is that

`hasTarget(?s,?t,Val)` ≡ `hasTarget_Val(?s,?t)`

for a ternary relation `hasTarget` with grade value `Val` as third argument.

Using list parameters and recursive sub-patterns, we can now provide one pattern that covers all necessary numbers of values, as in Fig. 7. The last parameter of `GradedRels` is a list of ontologies, with the assumption that each of them declares an individual of type `Val`. The local sub-pattern `Step` has as parameter a list of ontologies such that each of them declares an individual. In the instantiation `Step[v::valS]`, the first element of the argument list is the pattern obtained by expanding the notation `v`, i.e. the local environment that we denote by `Env`. In this case, this is the union of all formal parameters; and it contains a declaration for the individual `v`. The argument expands then to `Env` **`fit`** `x |-> v`. By assumption, each element of the list of ontologies `valS` declares an individual (and an axiom about its type, that is not needed here), so we can use it as an argument for `xs`, which is a list of ontologies each declaring an individual.

Template matching for list parameters. We may make use of the list constructor `::` to give different definitions for the same pattern according to the argument of the list parameter of that pattern. This is a case distinction similar to pattern matching in functional programming, that we call *template matching* here to avoid the overlap with ontology design patterns. In an instantiation, Hets goes sequentially through the list of all definitions for a pattern and checks whether the argument matches the parameter template. When a match is found, the body given in that definition is used for instantiation. If the argument for the list parameter is empty and no other definition for this case was provided, the instantiation is the empty ontology. If no match is found, the instantiation is incorrect.

As an example, we provide a GODP for extending a sheaf of graded relations with subsumption relations, see Fig. 8. The idea is to introduce relations for expressing that a property holds with at least or at most a grade, when the grades can be compared, and to create a subsumption hierarchy between the relations `p_G` and `p_atLeast_G`: a property `p` holds with a grade at least `G`, if it holds with grade `G` or it holds at least

```
pattern GradedRels
  [Class: S; Class: T;
   ObjectProperty: p Domain: S Range: T;
   Class: Val;
   {Individual: v Types: Val} :: valS  ]
= %% a sheaf of graded relations p[vi], one for each vi in v::valS
let pattern Step[Individual: x :: xs] =
     ObjectProperty: p[x] Domain: S Range: T SubPropertyOf: p
     then Step[xs]
in Step[v::valS] and
   { ObjectProperty: has[Val] Domain: T Range: Val }
```

```
ontology GradedRels_Significance =
 GradedRels[PhysicalObject;PhysicalObject;hasIngredient;
  Significance;[0Insignificant,1Subordinate,2Essential,3Dominant]]
```

Figure 7. `GradedRels` and instantiation `GradedRels_Significance`.

with a grade less than `G`. In this example, the recursion is shown both for a less-or-equal order (`atLeast`) and a greater-or-equal order (`atMost`); in the former, an initial step `AtMostInitial` is needed, while in the latter two cases for the recursion of `AtLeastStep` have to be distinguished to define a special final step for recursion termination. When `GradedRelsSub_Significance` (Fig. 8) has been expanded and the names stratified, we obtain a relation subsumption hierarchy between the graded relations as follows (only the `atLeast` relations are shown):

```
hasIngredient_atLeast_0Insignificant
 hasIngredient_0Insignificant
 hasIngredient_atLeast_1Subordinate
  hasIngredient_1Subordinate
  hasIngredient_atLeast_2Essential
   hasIngredient_2Essential
   hasIngredient_3Dominant
```

4. Conclusions and Future Work

As we have shown, generic DOL meets desiderata **C1**-**C5** concerning the expected capabilities of a language for ODPs enumerated in the introduction. **C6**, extensibility of the language, is not supported yet (a higher-order extension of the language is under consideration). A structured repository and a meta-ontology relating the GODPs in this repository are presently under development.

An important aspect is how to make the use of GODPs more intuitive for ontology developers. A good GODP would have to provide

- a good choice of names for the pattern and for the parameters,

```
pattern GradedRelsSub
 [Class: S; Class: T; ObjectProperty: p Domain: S Range: T;
  Class: Val; {Individual: v0 Types: Val} ::
              {Individual: v1 Types: Val} :: valS         ]
= %% GradedRels + subrelations p[atMost[vi]], p[atLeast[vi]]
let
 pattern Sub2[ObjectProperty: r;
              ObjectProperty: r1; ObjectProperty: r2] =
  ObjectProperty: r  Domain: S Range: T
  ObjectProperty: r1 SubPropertyOf: r
  ObjectProperty: r2 SubPropertyOf: r
 pattern AtMostInitial[Individual: x; Individual: y :: empty] =
  Sub2[p[atMost[y]];  p[x];          p[y]]
 pattern AtMostStep [Individual: x; Individual: y :: ys] =
  Sub2[p[atMost[y]];  p[atMost[x]];  p[y]] then AtMostStep[y; ys]
 pattern AtLeastStep[Individual: x; Individual: y :: empty] =
  Sub2[p[atLeast[x]]; p[x];          p[y]]
 pattern AtLeastStep[Individual: x; Individual: y :: ys] =
  Sub2[p[atLeast[x]]; p[atLeast[y]]; p[x]] then AtLeastStep[y;ys]
in  GradedRels[S;T;p;Val;v0::v1::valS]
and AtMostInitial[v0;v1] and AtMostStep[v1;valS]
and AtLeastStep[v0;v1::valS]
```

```
ontology GradedRelsSub_Significance =
 GradedRels_Significance then
 GradedRelsSub[PhysicalObject; PhysicalObject; hasIngredient;
  Significance;[0Insignificant,1Subordinate,2Essential,3Dominant]]
```

Figure 8. `GradedRelsSub` and instantiation of `GradedRelsSub_Significance`.

- a documentation part informing the user about the functionality of the pattern,
- an instantiation example.

Ideally, working with GODPs will be done via a GUI that hides the body of the pattern from the ontology developer (providing an appropriate documentation) and makes only those parameters visible that have to be instantiated.

Hets support for the Generic DOL language extensions introduced in this paper is currently in progress.

The GODPs presented in this paper are extensively used in an ontology on food, cooking and dietary restrictions (e.g. about 20 instantiations of `GradedRels` etc. in ca. 25,000 OWL axioms, cf. [6]); non-trivial GODPs are presently being developed for the robotics domain.

References

[1] 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.

[2] Pascal Hitzler, Aldo Gangemi, Krzysztof Janowicz, Adila Alfa Krisnadhi, and Valentina Presutti. Towards a Simple but Useful Ontology Design Pattern Representation Language. 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.

[3] Giancarlo Guizzardi. *Ontological Foundations for Structural Conceptual Models*. PhD thesis, University of Twente, 2005.

[4] Eduardo Zambon and Giancarlo Guizzardi. Formal Definition of a General Ontology Pattern Language using a Graph Grammar. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017, Prague, Czech Republic, September 3-6, 2017.*, pages 1–10, 2017.

[5] Martin G. Skjæveland, Leif Harald Karlsen, and Daniel P. Lupp. Practical Ontology Pattern Instantiation, Discovery, and Maintenance with Reasonable Ontology Templates - demo paper. In Marieke van Erp, Medha Atre, Vanessa López, Kavitha Srinivas, and Carolina Fortuna, editors, *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018.*, volume 2180 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[6] 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) 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.

[7] Till Mossakowski, Mihai Codescu, Fabian Neuhaus, and Oliver Kutz. The Distributed Ontology, Modeling and Specification Language – DOL. In Arnold Koslow and Arthur Buchsbaum, editors, *The Road to Universal Logic*, volume 2, pages 489–520. Birkhäuser, 2015.

[8] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set, Hets. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer, 2007.

[9] Bernd Krieg-Brückner, Till Mossakowski, and Fabian Neuhaus. Generic Ontology Design Patterns at Work. In Adrien Barton, Sejla Seppälä, and Daniele Porello, editors, *JOWO 2019. Joint Ontology Workshops (JOWO-2019), September 23-25, Graz, Austria*. CEUR Workshop Proceedings, CEUR-WS.org, 2019.

[10] Peter D. Mosses, editor. *CASL Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2004.

[11] Michael Schneider, Sebastian Rudolph, and Geoff Sutcliffe. Modeling in OWL 2 without Restrictions. In Mariano Rodriguez-Muro, Simon Jupp, and Kavitha Srinivas, editors, *Proceedings of the 10th International Workshop on OWL: Experiences and Directions (OWLED 2013) co-located with 10th Extended Semantic Web Conference (ESWC 2013), Montpellier, France, May 26-27, 2013.*, volume 1080 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[12] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[13] Bernd Krieg-Brückner. Generic Ontology Design Patterns: Qualitatively Graded Configuration. In Franz Lehner and Nora Fteimi, editors, *Knowledge Science, Engineering and Management - 9th International Conference, KSEM 2016, Passau, Germany, October 5-7, 2016, Proceedings*, volume 9983 of *Lecture Notes in Computer Science*, pages 580–595, 2016.