Generic Ontologies and Generic Ontology Design Patterns

Bernd Krieg-Brückner and Till Mossakowski

German Research Center for Artificial Intelligence (DFKI), CPS, BAALL, and
Universität Bremen, FB3 Mathematik und Informatik, Bremen, Germany
Bernd.Krieg-Brueckner@dfki.de
Institute for Intelligent Cooperating Systems, Faculty of Computer Science,
Otto-von-Guericke-Universität Magdeburg, Germany
till@iks.cs.ovgu.de

Abstract. *Generic Ontologies* are introduced in *GDOL*, an extension of *DOL*, the *Distributed Ontology, Modeling and Specification Language*, and implemented using *Hets*, the *Heterogeneous Tool Set* framework. Parameters such as classes, properties, individuals, or whole ontologies may be instantiated with arguments in a host ontology. A *Generic Ontology Design Pattern*, GODP, defined in GDOL, embodies an ontology development operation and serves as a methodological tool for ontology development. Some non-trivial GODPs are presented for illustration, e.g. for *qualitatively graded relations*.

1 Introduction

Structuring Ontologies In-The-Large is absolutely indispensable to retain an overview and allow separate developments ("divide and conquer"), when we consider large ontologies. For example, the "hyper-ontology" [6, 13] in [12] with more than 25.000 definitions is a network of ontologies, which built on top of each other: foundational so-called "Upper Ontologies", ontologies in a variety of general domains, and finally ontologies in application domains.

Standard OWL [2] allows such a breakdown into separate ontologies with import, although transitive import is not always properly supported by tools. Sometimes, however, we would like to import classes, relations etc. with renaming. We will look at support for this by DOL (Sect. 2.1) and Hets (Sect. 2.2).

We extend ontologies to *Generic Ontologies*, and DOL to Generic DOL (Sect. 3), to allow easy *reuse* of ontologies without cumbersome renaming.

Structuring Ontologies In-The-Small is required for sustained confidence in (or better: correctness of) intricate local, reusable development tasks.

We take generic ontologies as the basis for *Generic Ontology Design Patterns* (Sect. 4) embodying development operations. We show how a complex pattern can be structured into sub-patterns, and how iteration can be achieved. This is demonstrated by several non-trivial examples.

```
ontology VSet2 =
Class: Val
Class: Val0 SubClassOf: Val
 EquivalentTo: {VAL_Val0}
Class: Val1 SubClassOf: Val
 EquivalentTo: {VAL_Val1}
ObjectProperty: greater_Val
 Characteristics: Transitive
 Domain: Val Range: Val
Individual: VAL_Val0 Types: Val0
Individual: VAL_Val1 Types: Val1
 Facts: greater_Val VAL_Val0
end

ontology VSet3 = VSet2 and
  VSet2 with Val0 |-> Val1,
             Val1 |-> Val2,
         VAL_Val0 |-> VAL_Val1,
         VAL_Val1 |-> VAL_Val2
end
```

```
ontology VSet3Exp =
 Class: Val
 Class: Val0 SubClassOf: Val
  EquivalentTo: {VAL_Val0}
 Class: Val1 SubClassOf: Val
  EquivalentTo: {VAL_Val1}
 Class: Val2 SubClassOf: Val
  EquivalentTo: {VAL_Val2}
 ObjectProperty: greater_Val
  Characteristics: Transitive
  Domain: Val Range: Val
 Individual: VAL_Val0 Types: Val0
 Individual: VAL_Val1 Types: Val1
  Facts: greater_Val VAL_Val0
 Individual: VAL_Val2 Types: Val2
  Facts: greater_Val VAL_Val1
end
```

Fig. 1. VSet2, VSet3 as renaming of VSet2, and VSet3Exp

2 Structured Semantic Modelling

2.1 Structuring with DOL

A repository of general and specific domain ontologies may be utilized very well indeed for a variety of applications. In a special application, these ontologies are often only needed selectively, in specific "views".

The *Distributed Ontology, Modeling and Specification Language*, DOL, an OMG standard [19, 16, 14], allows, in particular, the definition of a selective view to an ontology upon import, and renaming of entities.

Such a "fine" structuring is a form of *structured reuse*. It avoids multiple developments and error-prone copies: later changes to an original are often mistakenly omitted in copies, or erroneous when replaying a slight adaptation.

The "Same Name—Same Thing" Principle is fundamental in DOL, as it is in OWL. Thus a definition may be repeated for a class, say, when adding an additional axiom. Note, however, the use of separate name spaces, see Sect. 3.2.

The Value Sets VSet2, VSet3, see Fig. 1, are examples of small ontologies that can be re-used as building blocks in other ontologies.

VSet2 contains a class Val with two subclasses Val0 and Val1, with one individual each. A transitive order relation greater_Val is defined on the individuals.

Now VSet3 is formulated in DOL as VSet2 "**and**" (i.e. united with) a copy of VSet2 "**with**" renamings; thus it contains 3 classes and 3 values.

Renamings take the form of a sequence of "mapping items" (using the symbol "|->") from old to new names, e.g. Val0 is renamed as Val1, Val1 as Val2, in the copy. Note that mere textual substitution would be error-prone: the replacements in Fig. 1 would only work in reverse order in this particular case. The expanded result is shown as VSet3Exp.

While being more concise, even with this DOL notation, the example will become cumbersome when constructing a value set of 4 or more values, because each time we have to use renaming for yet another copy with distinct names. So we are looking for a means to make a general construction reusable, and to structure it in such a way that iteration becomes easy and obvious to do, i.e. a generic solution (see Sect. 3.2, Fig. 3).

2.2 Heterogeneous Ontologies

DOL has been devised for the *heterogeneous* combination of theories in a variety of logics, dealing with logic morphisms, i.e. functions mapping a theory from one logic into another, and theory morphisms, i.e. functions mapping a specification (e.g. an ontology) into another in the same logic.

Actually, the axioms for a total order cannot be defined in OWL-DL, but could be defined in another logic, say FOL or the *Common Algebraic Specification Language*, CASL [4, 18, 7]. Such a logically complete formulation for a total (or any other desired) order would then accompany the DOL definition (see section 3.2.5 in [14]), to be used for verification purposes with Hets.

The *Heterogeneous Tool Set*, Hets [17], is the implementation basis for DOL and provides and implementation of its semantics. It takes care of proper replacements; e.g. it generates VSet3Exp.

3 Generic Ontologies in GDOL

We propose the language *Generic DOL*, GDOL, which extends DOL by a parameterization mechanism for ontologies. Generics in GDOL borrow their semantics from CASL's *generic specification* mechanism. The syntax for OWL in GDOL is presently Manchester Syntax, extended by Parameterized Names (cf. Sect. 3.2).

3.1 Generic Ontologies

Generics (first introduced in ADA [10]) are not just macros. Their most important aspect is that all parameters are fully typed items, and argument types are checked against parameter types. Moreover, generics are closed: all free names in the body come from imports, or are bound as parameters. When parameter to argument substitutions are interpreted in the context of the instantiation, frustrating name clashes in the context of the instantiation are thus avoided.

Parameters in CASL or GDOL may be any items in the signature of a theory, or whole theories. Thus the parameter of a generic ontology may not only be a class, relation, or individual, it may be a whole *ontology* with specific abstract

```
ontology List [Class: Item] =
 Class: EmptyList[Item]
  SubClassOf: List[Item]
 Class: NEList[Item]
  SubClassOf: List[Item]
 Class: List[Item]
  DisjointUnionOf:
   EmptyList[Item], NEList[Item]
 ObjectProperty: first[Item]
  Characteristics: Functional
  Domain: NEList[Item] Range: Item
 ObjectProperty: restList[Item]
  Characteristics: Functional
  Domain: NEList[Item]
  Range: List[Item]
end

ontology: List_Num = List[Class: Num] end
```

```
ontology List_NumExp =
 Class: Num
 Class: EmptyList_Num
  SubClassOf: List_Num
 Class: NEList_Num
  SubClassOf: List_Num
 Class: List_Num
  DisjointUnionOf:
   EmptyList_Num, NEList_Num
 ObjectProperty: first_Num
  Characteristics: Functional
  Domain: NEList_Num Range: Num
 ObjectProperty: restList_Num
  Characteristics: Functional
  Domain: NEList_Num
  Range: List_Num
end
```

Fig. 2. Generic List, Instantiation List_Num, and Expansion List_NumExp

properties expressed by axioms. An argument ontology must conform to such a parameter ontology, i.e. the required properties must be satisfied.

This concept makes structuring of ontologies in-the-large and in-the-small extremely powerful to capture *separate semantic concepts* independently, beyond mere modularization and "object-oriented" inheritance.

Generic Lists. Lists, sequences, or sets are classic examples of parameterized data structures. Such "containers" are also quite useful in ontologies, likely to be instantiated many times. A list is cast into a generic OWL ontology List in Fig. 2: the class Item is a parameter.

In the ontology List_Num, List is then instantiated, where an argument class Num is provided for the parameter Item; cf. also the expansion List_NumExp. This way, different instantiations, say List_Num and List_Person, may coexist, since classes and relations such as first_Num and first_Person are differentiated.

3.2 Pragmatic Naming in GDOL

Inherited Name Spaces. DOL/GDOL supports declaration of prefixes that can be used for the abbreviation of URLs in much the same way as OWL does. Each ontology, and hence each argument of an instantiation of a generic ontology, carries its own set of prefixes, possibly distinct.

The default prefix of the body of a GODP is set by the instantiation. Sometimes an item in the body is closely related to a parameter and should in fact inherit the name space of the argument in an instantiation.

For example in TaxonomicExtension (Fig. 5), X is interpreted with the default prefix of the instantiation, whereas Y might be intended to reside in a different name space; this decision is made when the argument for Y is provided in the instantiation. Thus X gets the default prefix, and Y its own prefix, denoted Y:Y.

For better readability, we drop the definition of the default prefix, or other prefixes (given in Fig. 5), in the presentation of the other examples in this paper.

Parameterized Names and Stratified Names. Names of new items are often related to ("dependent on") names of parameters. To avoid renaming and extra parameters, we use the construct of compound names in CASL to introduce *Parameterized Names* for use in ontologies.

As in the CASL notation, Parameterized Names use brackets "[" and "]" around constituent names, and comma "," as separator.

When parameters are substituted by arguments in the expansion of an instantiation, the constituent names are substituted by the corresponding argument names, and "[" and "," are replaced by "_" (the trailing "]" is dropped). The resulting *Stratified Names* are thus expressed in legal OWL notation, and all OWL-related tools can be used on instantiated generics.

Parameterized Names in List allow a compact notation (Fig. 2): just the parameter class Item needs to be instantiated with an argument. The new classes, relations etc. are expressed as parameterized names such as List[Item], first[Item], and corresponding stratified names such as List_Num, first_Num are generated by Hets, supplying a fresh name for each expanded instantiation.

Thus the required parameter definitions and instantiations are reduced considerably; if parameterized names were not available, all these names would have to be supplied as extra parameters (and arguments!), and generic instantiation would be quite cumbersome and not very different from renaming.

Generic Value Sets. While VSet2 (see Sect. 2.1) is not per se generic, it is made to be so as ValSet4, see Fig. 3, to provide new names for parameters Val, Val0, Val1, etc., such that a new instance with different names is created for each instantiation.

Note that we use multiple parameters, each consisting of a single class. The advantage is that we can write instantiations similarly, with one argument each, avoiding the need to rename, i.e. explicitly give a fitting morphism from parameter to argument.

ValSet4 is structured: ValSetInitial is the base for a generic extension, ValSetStep, which adds another value class and individual to the set, and the order. This extension may be iterated: in ValSet4, ValSetStep is instantiated 3 times, providing a total of 4 values Val0 to Val3. In a final phase, a disjointness axiom completes ValSet4. Note that the "Same Name—Same Thing" principle ensures that repeated instantiations may speak about the same class (e.g. Val2). In the extension of the instantiation for Significance, duplicates of class definitions, e.g. for 2-Essential, are removed by Hets, cf. Fig. 3.

```
ontology ValSet4
 [Class: Val]
 [Class: Val0][Class: Val1]
 [Class: Val2][Class: Val3]
= ValSetInitial
   [Class: Val][Class: Val0]
and ValSetStep[Class: Val]
   [Class: Val0][Class: Val1]
and ValSetStep[Class: Val]
   [Class: Val1][Class: Val2]
and ValSetStep[Class: Val]
   [Class: Val2][Class: Val3]
and Class: Val
     DisjointUnionOf:
      Val0, Val1, Val2, Val3
end
```

```
ontology ValSetInitial
  [Class: Val] [Class: Val0]
= Class: Val
   Class: Val0 SubClassOf: Val
    EquivalentTo: {VAL[Val0]}
   Individual: VAL[Val0]
    Types: Val0
   ObjectProperty: greater[Val]
    Characteristics: Transitive
    Domain: Val Range: Val
end
```

```
ontology ValSetStep
  [Class: Val]
 [Class: PrevVal]
  [Class: NextVal]
= ValSetInitial
    [ Class: Val][ Class: PrevVal]
and
  Class: NextVal SubClassOf: Val
   EquivalentTo: {VAL[NextVal]}
  Individual: VAL[NextVal]
   Types: NextVal
   Facts: greater[Val] VAL[PrevVal]
end
```

```
ontology Significance
= ValSet4
 [Class: Significance]
 [Class: 0-Insignificant]
 [Class: 1-Subordinate]
 [Class: 2-Essential]
 [Class: 3-Dominant]
end
```

```
ontology GradedRelations4Exp =
Class: 0-Insignificant  SubClassOf: Significance
 EquivalentTo: {VAL_0-Insignificant}
Class: 1-Subordinate    SubClassOf: Significance
 EquivalentTo: {VAL_1-Subordinate}
Class: 2-Essential      SubClassOf: Significance
 EquivalentTo: {VAL_2-Essential}
Class: 3-Dominant       SubClassOf: Significance
 EquivalentTo: {VAL_3-Dominant}
Class: Significance
 DisjointUnionOf: 0-Insignificant, 1-Subordinate, 2-Essential, 3-Dominant
ObjectProperty: greater_Significance  Range:  Significance
 Characteristics: Transitive          Domain: Significance
Individual: VAL_0-Insignificant       Types:  0-Insignificant
Individual: VAL_1-Subordinate         Types:  1-Subordinate
 Facts: greater_Significance VAL_0-Insignificant
Individual: VAL_2-Essential           Types:  2-Essential
 Facts: greater_Significance VAL_1-Subordinate
Individual: VAL_3-Dominant            Types:  3-Dominant
 Facts: greater_Significance VAL_2-Essential
end
```

Fig. 3. ValSet4, ValSetInitial, ValSetStep, instantiation as Significance and expansion

```
ontology OrderRelationExtension
 [ OrderRelation] =
ObjectProperty: greater_Val
 InverseOf: less_Val
 SubPropertyOf: greaterOrEqual_Val
ObjectProperty: less_Val
 InverseOf: greater_Val
 SubPropertyOf: lessOrEqual_Val
 Characteristics: Transitive
 Domain: Val  Range: Val
ObjectProperty: greaterOrEqual_Val
 InverseOf: lessOrEqual_Val
 Characteristics:
  Transitive, Reflexive
 Domain: Val  Range: Val
ObjectProperty: lessOrEqual_Val
 InverseOf: greaterOrEqual_Val
 Characteristics:
  Transitive, Reflexive
 Domain: Val  Range: Val
end
```

```
ontology OrderRelation =
 Class: Val
 ObjectProperty: greater_Val
  Characteristics: Transitive
  Domain: Val  Range: Val
end
```

```
ontology ValSet4WithOrderRelations
 [ Class: Val]
 [ Class: Val0][ Class: Val1]
 [ Class: Val2][ Class: Val3]
= OrderRelationExtension
    [  ValSet4 [ Class: Val]
      [ Class: Val0][ Class: Val1]
      [ Class: Val2][ Class: Val3]]
end
```

Fig. 4. OrderRelationExtension; composition with ValSet4 to ValSet4WithOrderRelations

3.3 Pre-conditions in Ontology Parameters

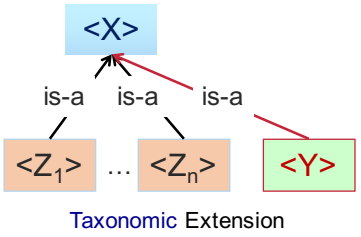
Pre-conditions to an instantiation may be stated by axioms in parameters expressed as whole ontologies, a powerful way to *ensure semantic consistency*.

OrderRelationExtension in Fig. 4 is the extension of greater_Val, an order relation, to its inverse less_Val, the reflexive version greaterOrEqual_Val, and its inverse lessOrEqual_Val. The parameters Val and greater_Val are formulated together as an *ontology parameter* OrderRelation. This way, the *precondition* that greater_Val should be transitive, is formulated in it, and will be checked for the argument (for a complete definition of an order relation in DOL cf. Sect. 2.2).

In ValSet4WithOrderRelations, the instantiation of OrderRelationExtension receives both the value set Val to be extended, and the order relation greater_Val, i.e. a proper OrderRelation, from nested instantiation of ValSet4. The resulting ontology is, as always, a union of the argument(s) and the (translated) body, thus contains the instantiated ValSet4 plus the extensions.

4 Generic Ontology Design Patterns

Most of the existing patterns in the repository for *ontology patterns* [1] treat Upper Ontologies, or design issues and structuring in specific application domains (cf. Content Design Patterns in [9, 20]). *Generic Ontology Design Patterns*,




```
%prefix( : <http://X.owl#>
         Y: <http://Y.owl#> )%
ontology TaxonomicExtension
 [Class: X] [Class: Y:Y]
= Class: Y:Y SubClassOf: X
end
```

```
ontology IndividualExtension
 [Class: X]
= Individual: VAL[X] Types: X
end

ontology DisjointUnionExtension
 [Class: X
  Class: Z1 SubClassOf: X
  Class: Zn SubClassOf: X]
 [Class: Y]
= Class: Y SubClassOf: X
  Class: X DisjointUnionOf: Y,Z1,Zn
   reject DisjointUnionOf: Z1,Zn
end
```

Fig. 5. IndividualExtension, TaxonomicExtension and DisjointUnionExtension

GODPs (cf. [12] for a first sketch), are mostly *domain independent*, universal *Ontology Design Patterns* according to the classification of [8]. They abstract away from application domains to a generic methodological level, and are then instantiated to particular domains as part of an ontology engineering process. A GODP may be a large ontology to be reused, but also a tiny ontology fragment to be composed with others, thus structuring ontologies in-the-large or in-the-small.

GODPs now receive their semantic foundation from *generic ontologies* (cf. Sect. 3) and implementation basis from Hets.

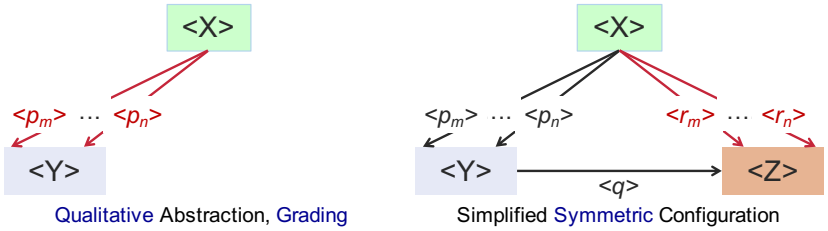
Examples for GODPs were introduced informally by diagrams in [12]. We formalize some of them here as examples for GODPs in GDOL/Hets.

Taxonomic Extension. The diagram for a TaxonomicExtension pattern in Fig. 5 illustrates the introduction of a new class $<Y>$ as a subclass of $<X>$.

Like the generation of a standard individual in IndividualExtension, this appears to be an extremely simple pattern, whose effect might just as well be achieved by free editing. However, it stands for a set of GODPs corresponding to development operations that rope in free editing and capture it under semantic control in an application context (cf. Sect. 5).

Disjoint Union Extension. Moreover, if previously the subclasses $<Z_1>$, $<Z_n>$ of $<X>$ were stated to be disjoint, we would expect the disjoint union axiom to be extended as well. When adding another $<Y>$ that should be included in the union, we have to "repair" the disjoint union axiom to include $<Y>$ by rejecting the old axiom and introducing the extended one (cf. [15]).

The need for an extension of a previous disjoint union (with associated deletion) is circumvented in the example ValSet4 (Fig. 3) by introducing an explicit completion with a disjoint union of all constituent value classes as a final phase.




```
ontology GradedRelations4
 [Class: S][Class: T][Class: Val]
 [Class: Val0][Class: Val1]
 [Class: Val2][Class: Val3] =
 ObjectProperty: has[T,Val,Val0]
  Domain: S Range: T
 ObjectProperty: has[T,Val,Val1]
  Domain: S Range: T
 ObjectProperty: has[T,Val,Val2]
  Domain: S Range: T
 ObjectProperty: has[T,Val,Val3]
  Domain: S Range: T
 ObjectProperty: has[Val]
  Domain: T Range: Val
end
```

```
ontology GradedRels4_Significance
= GradedRelations4
  [Class: Ingredient]
  [Class: Ingredient]
  [Class: Significance]
  [Class: 0-Insignificant]
  [Class: 1-Subordinate]
  [Class: 2-Essential]
  [Class: 3-Dominant]
end
```

Fig. 6. Qualitative Grading and Configuration patterns, and GradedRelations4

Qualitative Abstraction. To achieve a graded valuation according to some qualitative abstraction of a semantic concept, we might introduce extra valuation domains, e.g. Significance (see Fig. 3), with values such as Insignificant, Subordinate, Essential, Dominant, or some other (arbitrarily fine) qualitative metrics; the number of levels depends on the application.

To combine a class, say C, with its valuations, resp., we could construct extra combined domains, say C-Insignificant, ... , C-Dominant, one for each domain, or standard individuals for valuation combinations. The general approach in this vein is *reification* to realize a ternary relation by binary relations.

These approaches involve a considerable overhead of introducing extra classes and relations, and are clumsy, barely readable, and ultimately error-prone.

Qualitatively Graded Relations have been proposed instead in [12] to encode such valuations. Formally, we use an ordered set (Sect. 3.2) as grading domain. A ternary relation hasTarget with grade value Val1 as third argument is represented by a qualitatively graded relation hasTargetWithGrade_Val1 such that

hasTarget(?s,?t,VAL_Val1) ≡ hasTargetWithGrade_Val1(?s,?t)

This axiom cannot be expressed in OWL, but defines the proper semantic relationships that may be used for verification in Hets.

```
ontology GESubsumption4
 [Class: S][Class: T][Class: Val]
 [Class: Val0][Class: Val1]
 [Class: Val2][Class: Val3]
= GradedRelations4 [Class: S]
    [Class: T][Class: Val]
    [Class: Val0][Class: Val1]
    [Class: Val2][Class: Val3]
and
GESubStep
 [Class: S][Class: T][Class: Val]
 [Class: Val0][Class: Val1]
 [ObjectProperty: has[T,Val,Val0]]
and
GESubStep
 [Class: S][Class: T][Class: Val]
 [Class: Val1][Class: Val2]
 [ObjectProperty: has[T,Val,Val1]]
and
GESubFinal
 [Class: S][Class: T][Class: Val]
 [Class: Val2][Class: Val3]
 [ObjectProperty: has[T,Val,Val2]]
 [ObjectProperty: has[T,Val,Val3]]
end
```

```
ontology GESubStep
 [Class: S][Class: T][Class: Val]
 [Class: Vali][Class: Valj]
 [ObjectProperty: has-T-Val-Vali]
= ObjectProperty: hasGE[T,Val,Vali]
    Domain: S  Range: T
  ObjectProperty: has-T-Val-Vali
    SubPropertyOf: hasGE[T,Val,Vali]
  ObjectProperty: hasGE[T,Val,Valj]
    SubPropertyOf: hasGE[T,Val,Vali]
    Domain: S  Range: T
end

ontology GESubFinal
 [Class: S][Class: T][Class: Val]
 [Class: Vali][Class: Valj]
 [ObjectProperty: has-T-Val-Vali]
 [ObjectProperty: has-T-Val-Valj]
= ObjectProperty: hasGE[T,Val,Vali]
    Domain: S  Range: T
  ObjectProperty: has-T-Val-Vali
    SubPropertyOf: hasGE[T,Val,Vali]
  ObjectProperty: has-T-Val-Valj
    SubPropertyOf: hasGE[T,Val,Vali]
end
```

```
ontology GESubsumption4_SigExp =
 Class: Ingredient  Class: Significance
 ObjectProperty: hasGE_Ingredient_Significance_0-Insignificant
  Domain: Ingredient Range: Ingredient
 ObjectProperty: hasGE_Ingredient_Significance_1-Subordinate
  SubPropertyOf: hasGE_Ingredient_Significance_0-Insignificant
  Domain: Ingredient Range: Ingredient
 ObjectProperty: hasGE_Ingredient_Significance_2-Essential
  SubPropertyOf: hasGE_Ingredient_Significance_1-Subordinate
  Domain: Ingredient Range: Ingredient
 ObjectProperty: has_Ingredient_Significance_0-Insignificant
  SubPropertyOf: hasGE_Ingredient_Significance_0-Insignificant
  Domain: Ingredient Range: Ingredient
 ObjectProperty: has_Ingredient_Significance_1-Subordinate
  SubPropertyOf: hasGE_Ingredient_Significance_1-Subordinate
  Domain: Ingredient Range: Ingredient
 ObjectProperty: has_Ingredient_Significance_2-Essential
  SubPropertyOf: hasGE_Ingredient_Significance_2-Essential
  Domain: Ingredient Range: Ingredient
 ObjectProperty: has_Ingredient_Significance_3-Dominant
  SubPropertyOf: hasGE_Ingredient_Significance_2-Essential
  Domain: Ingredient Range: Ingredient
 ObjectProperty: has_Val  Domain: Ingredient Range: Significance
```

Fig. 7. GESubsumption4, GESubStep, GESubFinal and expansion of instantiation

Generic GradedRelations. With this encoding we obtain a sheaf of relations $<p_i>$, one for each qualitative grade i, cf. Fig. 4. Note that GradedRelations4 is appropriate for a grade domain with 4 values; we have to provide an analogous GODP for each number by adding further relations.

Graded Configuration can be defined in a similar way (cf. [12]), taking two sheafs of graded relations as parameters.

Graded Subsumption. A grading level often corresponds to an appropriate interval of quantitative properties. When the levels are ordered (the intervals are consecutive), it makes sense to also introduce associated relations corresponding to a greater[Val] ordering (actually a greaterOrEqual[Val] ordering; ValSet4WithOrderRelations could be added if verification is intended).

In GESubsumption4, see Fig. 7, this is expressed by subsumption of relations, where a given sheaf of graded relations is a parameter. Whereas an instantiation of GradedRelations4 (Fig. 6) only generates relations such as has_Ingredient_Significance_1-Subordinate, has_Ingredient_Significance_2-Essential, etc., an instantiation of GESubsumption4 additionally yields e.g. a relation hasGE_Ingredient_Significance_1-Subordinate, which subsumes grades 1-Subordinate, 2-Essential, and 3-Dominant. We obtain the following relation subsumption hierarchy:

hasGE_Ingredient_Significance_0-Insignificant
  has_Ingredient_Significance_0-Insignificant
  hasGE_Ingredient_Significance_1-Subordinate
    has_Ingredient_Significance_1-Subordinate
    hasGE_Ingredient_Significance_2-Essential
      has_Ingredient_Significance_2-Essential
      has_Ingredient_Significance_3-Dominant

To create this hierarchy in GESubStep, a new GE-relation hasGE[T,Val,Valj] is defined as a subrelation of the previous GE-relation hasGE[T,Val,Vali], and has-T-Val-Vali, passed as a parameter, is also made a subrelation of the previous GE-relation hasGE[T,Val,Vali] (the instantiations are chosen in such a way that j denotes i+1).

5 Conclusion

Development Operations with Constrained Focus. A GODP abstracts away from a particular development fragment. As such it embodies an *ontology development operation*, to be re-used repeatedly in a variety of contexts. While a free development, e.g. in Protégé, allows changes everywhere and cannot guard against erroneous changes in a different part of the hyper-ontology, a GODP provides a constrained focus, a kind of "straightjacket" to allow only this particular development operation. Changes are confined to the effects of an instantiation, no other part of the host ontology is accidentally affected, it is "frozen".

The developers attention is focused on providing appropriate arguments, which are checked for violation of semantic constraints.

Composition of Development Operations is a powerful way to create new GODPs from the repository of generally applicable building blocks, viz. GODPs. We have seen in Fig. 4 that OrderRelationExtension may be instantiated on ValSet4, yielding a new GODP ValSet4WithOrderRelations.

Similarly, a combination of initial, iteration step and final sub-GODPs as in ValSet4 (Fig. 3), or GESubsumption4 (Fig. 7) seems to be quite typical for the compilation of larger GODPs from small building blocks.

Consistency. A GODP is separately defined from its applications; it can be tested to comply with the intentions of the developer once and for all (i.e. for all instantiations). The semantics of its body is defined w.r.t. its parameters; *semantic pre-conditions* may be stated by axioms in parameter ontologies.

Structural consistency is then ensured in each instantiation by checking structural conformance of arguments, e.g. that an argument of appropriate kind with the resp. name exists (subject to renaming). *Semantic consistency* is ensured by proving each ontology argument to comply with its parameter, if it contains axioms. Such a proof obligation is generated and managed by Hets. It may be discharged automatically, if it can be reduced to Description Logic, DL, or some other logic for which automatic reasoning is provided by an appropriate reasoner.

In fact, every (generic) ontology pattern, e.g. for Order (cf. Sect. 2.2), should be accompanied by a formal definition in CASL, say, if DL is not sufficient, to properly define its semantics and to support a potential verification process.

Towards Safer Ontology Development. It cannot be expected that ontologies are ever "complete"; thus the development process and the correctness of changes on the resulting ontologies are a major concern.

A GODP, separately defined and verified, is *safe* insofar as *changes are confined* to the effects of pattern instantiation, checking arguments for violation of semantic constraints; no other part of the host ontology is accidentally affected.

The *constrained development focus* of "straightjacket" GODPs seems to have potential for supporting a *safe development process*.

6 Ongoing and Future Work

Development Support Tools. We are working on an environment for just such "straightjacket" GODPs focussing on one GODP as a development operation at a time while other changes are blocked. A planned user interface, as a Protégé [3] plugin, will take care of smooth interaction with Hets, tracks changes with change management, and provides development control (see below).The vision for the future is that all development will be done this way and free editing is banned.

This user interface will be for the *semantic modelling expert*, who will develop new general GODPs, and the *domain expert* selecting ready-made interfaces on top of GODPs for specific development tasks in an application domain.

The above experts are allowed to change the modelling level of an ontology; an application interface for an *end-user* will only be able to browse (with a frozen model), and add or change application-oriented data (i.e. individuals and relating facts), such as person profiles, new recipes, or shopping lists.

Compilation of Application-Oriented Patterns. We expect the *semantic modelling expert* to be able to easily compile specialized domain-oriented GODPs from the repository of GODPs. An extra user-interface might then be added for the *domain expert*, compiled from a repository of appropriate components, but it is mere "syntactic sugar" ("interface topping") with its semantics firmly and safely rooted in the GODP framework.

Generation of Ontology Components is usually quite cumbersome and error-prone, since code using the OWL-API has to be written and debugged.

With GODPs, this process is raised to an appropriate abstraction level (including potential verification). Only the triggering of the application of a GOPD has to be programmed (to be included in our support tools). The iteration of a GODP application is then easily achieved, possibly by development control (see below). As a simple example consider the generation of a standard individual IndividualExtension (Fig. 5), to be repeated for all classes in a particular context.

Development Control for complex development tasks, guiding through subtasks, will require further research. Examples are *collateral* or *intermediate abstraction* (cf. [12]), where an ongoing development is temporarily suspended, while another auxiliary development is pursued, and subsequently resumed; in fact, control is alternating between subtasks.

We envisage *development terms* with GODP instantiations as development atoms and operations such as subordinate or collateral composition, where subdevelopments are initiated while the parent development stays suspended. Similarly, iteration of developments is intended (cf. development functionals in [11]).

Development Ontology. Development operations, their parameter profiles and other details will be formalized in a Development Ontology, which is imported into an actual development, but does not affect its semantics; links into it act like formal annotations. Of particular interest is the interlinking of the application of a development operation, viz. instantiation of a GODP, to the GODP and the arguments in the instantiation.

Change Management. The definition of such a Development Ontology, which is the basis for change management, is fairly well advanced. On the basis of semantic change impact analysis [5], development operations will be treated as first class citizens; this will open the possibility for transformations of development terms such as reordering according to algebraic properties.

Not only GODPs but whole developments will be re-usable by replay, potentially with adaptation transformations.

GDOL extensions. GODPs are presently supported as generic CASL specifications in Hets. Generic ontologies are defined as a Generic extension of DOL to GDOL, to be proposed as an extension of the DOL standard (cf. Sect. 2.1).

GDOL examples in this paper (available at `https://ontohub.org/godp`) are confined to *extensions* of an ontology (actually the vast majority). Deletions of items in GDOL receive their semantics in [15].

In contrast to CASL, where generics are "first order", we are considering an extension, where separate parameters, which often occur in the examples above, are taken to augment the context one by one; in fact, new generics could be defined from existing ones by partial parameterization (whether one should go all the way to introduce higher-order generics is a different question). This seems to be natural and would simplify some of the examples: e.g., with

GESubStepVal = GESubStep[Class: S][Class: T][Class: Val]

as a local definition in GESubStep, these parameters are fixed, and the previous instantiations of GESubStep are then reduced to just the remaining arguments, e.g. GESubStepVal[Class: Val0][Class: Val1][ObjectProperty: has[T,Val,Val0]].

We are also contemplating an extension of the syntax for GODPs in GDOL that corresponds to iterations in patterns, denoted e.g. by some elliptical notation. Compare the initial, iteration step and final phases represented by extra component GODPs in Fig. 3 (Sect. 3.2), which suggest an ellipsis in the body of ValSet4 for instantiations of ValSetStep, accompanied by some ellipsis for extra parameters (similarly in Fig. 6 and Fig. 7).

Moreover, it would be nice, if Hets would take the type of an argument (as required by the parameter) into account in instantiations; this would eliminate the need for stating such information in the argument and render instantiations considerably more compact: "**Class:**" could e.g. be deleted everywhere in the body of ValSet4 (Fig. 3), GradedRelations4 (Fig. 6), or GESubsumption4 (Fig. 7), and similarly in the associated instantiations.

Acknowledgements. We are very grateful to the reviewers and Serge Autexier, Jens Pelzetter, and Martin Rink for their suggestions and contributions.

References

1. Ontology Design Pattern Types, `ontologydesignpatterns.org/wiki/OPTypes`
2. OWL Web Ontology Language - Use Cases and Requirements - W3C Recommendation 10 February 2004, `www.w3.org/TR/2004/REC-webont-req-20040210/`
3. Protégé Desktop User Documentation, August 2016, `protegewiki.stanford.edu/wiki/ProtegeDesktopUserDocs`
4. Astesiano, E., Bidoit, M., Krieg-Brückner, B., Kirchner, H., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL - the Common Algebraic Specification Language. Theoretical Computer Science 286, 153–196 (2002), `http://www.cofi.info`
5. Autexier, S., Hutter, D., Mossakowski, T.: Change management for heterogeneous development graphs. In: Siegler, S., Wasser, N. (eds.) Verification, Induction, Termination Analysis, Festschrift in honor of Christoph Walther. LNCS, Springer (November 2010)

6. Bateman, J.A., Castro, A., Normann, I., Pera, O., Garcia, L., Villaveces, J.M.: OASIS Common hyper-ontological framework (COF). EU FP7 Project OASIS – Open architecture for Accessible Services Integration and Standardization Deliverable D1.2.1, Bremen University, Bremen, Germany (January 2010)
7. Bidoit, M., Mosses, P.D. (eds.): CASL User Manual, LNCS, vol. 2900. Springer, Berlin, Heidelberg (2004)
8. Blomqvist, E., Sandkuhl, K.: Patterns in ontology engineering: Classification of ontology patterns. In: Chen, C., Filipe, J., Seruca, I., Cordeiro, J. (eds.) ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems, Miami, USA, May 25-28, 2005. pp. 413–416 (2005)
9. Gangemi, A.: Ontology design patterns for semantic web content. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, Proceedings. LNCS, vol. 3729, pp. 262–276. Springer (2005)
10. Ichbiah, J., Krieg-Brückner, B., Wichmann, A., Ledgard, H., Heliard, J.C., Abrial, J., Barnes, J., Roubine, O.: Preliminary Ada Reference Manual. ACM SIGPLAN Notices (14:6 Part A) (1979)
11. Krieg-Brückner, B.: Transformational meta program development. In: Broy, M., Wirsing, M. (eds.) Methods of Programming. LNCS, vol. 544, pp. 19–33. Springer, Berlin, Heidelberg (1991)
12. Krieg-Brückner, B.: Generic Ontology Design Patterns: Qualitatively Graded Configuration. In: Lehner, F., Fteimi, N. (eds.) KSEM 2016, The 9th International Conference on Knowledge Science, Engineering and Management. Lecture Notes in Artificial Intelligence, vol. 9983, pp. 580–595. Springer International Publishing (2016)
13. Kutz, O., Mossakowski, T., Lücke, D.: Carnap, Goguen, and the Hyperontologies: Logical Pluralism and Heterogeneous Structuring in Ontology Design. Logica Universalis 4(2), 255–333 (2010), special Issue on 'Is Logic Universal?'
14. Mossakowski, T., Codescu, M., Neuhaus, F., Kutz, O.: The Distributed Ontology, Modeling and Specification Language – DOL. In: Koslow, A., Buchsbaum, A. (eds.) The Road to Universal Logic, vol. I, pp. 489–520. Birkhäuser (2015)
15. Mossakowski, T., Krieg-Brückner, B.: Partial Pushout Semantics of Generics in GDOL (submitted)
16. Mossakowski, T., Kutz, O., Codescu, M., Lange, C.: The distributed ontology, modeling and specification language. In: Vescovo, C.D., Hahmann, T., Pearce, D., Walther, D. (eds.) WoMo 2013. CEUR-WS online proceedings, vol. 1081 (2013)
17. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set, Hets. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Proceedings. LNCS, vol. 4424, pp. 519–522. Springer (2007)
18. Mosses, P.D. (ed.): CASL Reference Manual, LNCS, vol. 2960. Springer, Berlin, Heidelberg (2004)
19. Object Management Group: The distributed ontology, modeling, and specification language (DOL) (2016), OMG standard available at `omg.org/spec/DOL`. See also `dol-omg.org`
20. Presutti, V., Gangemi, A.: Content ontology design patterns as practical building blocks for web ontologies. In: Li, Q., Spaccapietra, S., Yu, E.S.K., Olivé, A. (eds.) Conceptual Modeling - ER 2008, 27th International Conference on Conceptual Modeling, Barcelona, Spain, October 20-24, 2008. Proceedings. LNCS, vol. 5231, pp. 128–141. Springer (2008)