The Distributed Ontology, Model and Specification Language (DOL) Day 3: Structured OMS

Oliver Kutz[1]
Till Mossakowski[2]

[1]Free University of Bozen-Bolzano, Italy
[2]University of Magdeburg, Germany

Tutorial at ESSLLI 2016, Bozen-Bolzano, August 15 – 19

Summary of Day 2

On Day 2 we have looked at:

- intended consequences (competency questions)
- model finding and refutation of lemmas
- extensions and conservative extensions
- signature morphisms and the satisfaction condition
- refinements / theory interpretations

Today

We will focus today on structured OMS:

- Assembling OMS from pieces:
  Basic OMS, union, translation
- Making a large OMS smaller:
  module extraction, approximation, reduction, filtering
- Non-monotonic reasoning through employing
  a closed-world assumption:
  minimization, maximization, freeness, cofreeness

Assembling OMS from Pieces

Unions

O_1 **and** O_2: union of two stand-alone OMS

- Signatures (and axioms) are united
- model classes are intersected
- difference to extensions: there, O_2 needs to be basic

```
logic CASL.FOL=
spec Magma =
  sort Elem;  ops 0:Elem; __+__:Elem*Elem->Elem   end
spec CommutativeMagma = Magma then
  forall x,y:Elem . x+y=y+x                        end
spec Monoid = Magma then
  forall x,y,z:Elem . x+0=x
                    . x+(y+z) = (x+y)+z            end
spec CommutativeMonoid =
  CommutativeMagma and Monoid                      end
```

Competency Questions Revisited

Competency Questions – Simplified Summary

- Let O be an ontology
- Capture requirements for O as pairs of scenarios and competency questions
- For each scenario competency question pair S, Q:
  - Formalize S, resulting in theory Γ
  - Formalize Q, resulting in formula φ
  - Check with theorem prover whether $O \cup \Gamma \models \varphi$
- When all proofs are successful, your ontology meets the requirements.

Competency Questions Revisited

- CQ most successful idea for ontology evaluation
- Technically, CQ = proof obligations
- Language for expressing proof obligations?
- Ad hoc handling of CQs

We asked:

- How do we keep track of scenarios and competency questions in a systematic way?

Answer: The DOL constructs of `and` (union) and `%implies`

Competency Questions Workflow

1. The use cases for the ontology are captured in form of scenarios. Each scenario describes a possible state of the world and raises a set of competency questions. The answers to these competency questions should follow logically from the scenario – provided the knowledge that is supposed to be represented in the ontology.
2. A scenario and its competency questions are formalized or an existing formalization is refined.
3. The ontology is (further) developed.
4. An automatic theorem prover is used to check whether the competency questions logically follow from the scenario and the ontology.
5. Steps (2-4) are repeated until all competency questions can be proven from the combination of the ontology and their respective scenarios.

CQ Example: Family Relations

Ontohub enables the representation and execution of competency questions with the help of DOL files.

The use case is to enable semantically enhanced searches for a database, which contains names of people, their gender, and information about parenthood. Assuming the database contains the following information:

- *Amy is female and a parent of Berta and Chris.*
- *Berta is female.*
- *Chris is male and a parent of Dora.*
- *Dora is female.*

CQ Example: Family Relations (continued)

In this case the system should be able to answer the following questions:

- *Is Chris a father? (expected: yes)*
- *Is Dora a child of Chris (expected: yes)*
- *Is Chris female? (expected: no)*
- *Is Amy older than Dora? (expected: yes)*
- *Is Berta older than Chris (expected: unknown)*

CQ Example: Input Ontology

The ontology just discussed could be represented as follows.

```
logic OWL

ontology genealogy =
  Class: Male
  Class: Female

  ObjectProperty: parent_of
  Characteristics:  Irreflexive,  Asymmetric
  SubPropertyOf: older_than

  Class: Father
  EquivalentTo: parent_of some owl:Thing and Male

  ObjectProperty: child_of
  InverseOf: parent_of

  DisjointClasses: Male, Female

  ObjectProperty: older_than
  Characteristics:  Transitive
end
```

CQ Example: Scenario Formalisation

```
ontology  scenario =
  Class: Male
  Class: Female
  ObjectProperty: parent_of

  Individual: Amy
  Types: Female
  Facts: parent_of  Berta
  Facts: parent_of  Chris

  Individual: Berta
  Types: Female

  Individual: Chris
  Types: Male
  Facts: parent_of Dora

  Individual: Dora
  Types: Female
end
```

CQ Example: Competency Questions Formalisation

```
ontology CCbase = genealogy  and scenario
%% Is Chris a father? (expected: yes)
ontology CC1 =  CCbase then %implies
  { Individual: Chris
    Types: Father }
%% Is Dora a child of Chris (expected: yes)
ontology CC2 =  CCbase then %implies
  { Individual: Dora
    Facts: child_of Chris }
%% Is Chris female? (expected: no)
%% reformulated: Is Chris not female? (expected: yes)
ontology CC3 =  CCbase then %implies
  { Individual: Chris
    Types: not Female }
%% Is Amy older than Dora? (expected: yes)
ontology CC4 =  CCbase then %implies
  { Individual: Amy
    Facts: older_than Dora }
%% Is Berta older than Chris (expected: unknown)
ontology CC5 =  CCbase then %satisfiable
  { Individual: Berta
    Facts: older_than Chris }
```

CQ approach applied to machine diagnosis

Suppose the engine of a car does not perform properly. We want to decide whether we should

- repair the engine,
- replace the engine, or
- replace auxiliary equipment.

Some Rules for Machine Diagnosis

The following facts relate symptoms to diagnoses:

(i) If the engine overheats and the ignition is correct, then the radiator is clogged.

(ii) If the engine emits a pinging sound under load and the ignition timing is correct, then the cylinders have carbon deposits.

(iii) If power output is low and the ignition timing is correct, then the piston rings are worn, or the carburetor is defective, or the air filter is clogged.

(iv) If the exhaust fumes are black, then the carburetor is defective, or the air filter is clogged.

(v) If the exhaust fumes are blue, then the piston rings are worn, or the valve seals are worn.

(vi) The compression is low if and only if the piston rings are worn.

Some Rules for Machine Diagnosis

The following facts relate diagnoses to repair decisions:

(i) If the piston rings are worn, then the engine should be replaced.

(ii) If carbon deposits are present in the cylinders or the carburetor is defective or valve seals are worn, then the engine should be repaired.

(iii) If the air filter or radiator is clogged, then that equipment should be replaced.

Machine Diagnosis: Input Specification

```
logic Propositional

%% possible symptoms of an engine that is malfunctioning
spec EngineSymptoms =
  props black_exhaust, blue_exhaust, low_power, overheat,
        ping, incorrect_timing, low_compression
end

%% diagnosis derived from symptoms
spec EngineDiagnosis = EngineSymptoms then %cons
  props carbon_deposits, clogged_filter, clogged_radiator,
        defective_carburetor, worn_rings, worn_seals
   . overheat /\ not incorrect_timing => clogged_radiator  %(diagnosis1)%
   . ping /\ not incorrect_timing => carbon_deposits       %(diagnosis2)%
   . low_power /\ not incorrect_timing =>
                worn_rings \/ defective_carburetor \/ clogged_filter
                         %(diagnosis3)%
   . black_exhaust => defective_carburetor \/ clogged_filter %(diagnosis4)%
   . blue_exhaust => worn_rings \/ worn_seals                %(diagnosis5)%
   . low_compression <=> worn_rings                          %(diagnosis6)%
end
```

Machine Diagnosis: Input Specification (cont'd)

```
%% needed repair, derived from diagnosis
spec EngineRepair = EngineDiagnosis
then %cons
  props replace_auxiliary,
        repair_engine,
        replace_engine
  . worn_rings => replace_engine            %(rule_replace_engine)%
  . carbon_deposits \/ defective_carburetor \/ worn_seals => repair_engine
                                            %(rule_repair_engine)%
  . clogged_filter \/ clogged_radiator => replace_auxiliary
                                            %(rule_replace_auxiliary)%
end
```

Machine Diagnosis: Scenario Formalisation

Suppose the car owner complains that the engine overheats. Due to a recent engine check, it is known that the ignition timing is correct. What should be done to eliminate the problem?

```
spec MyObservedSymptoms =
  EngineSymptoms
then
  . overheat                  %(symptom_overheat)%
  . not incorrect_timing      %(symptom_not_incorrect_timing)%
end
```

Diagnosis Question Formalisation

```
spec MyRepair =
  EngineRepair and MyObservedSymptoms
end

spec Repair =
  prop repair
  . repair
end

interpretation repair1 : Repair to MyRepair = %cons
  repair |-> replace_engine end
interpretation repair2 : Repair to MyRepair = %cons
  repair |-> repair_engine end
interpretation repair3 : Repair to MyRepair = %cons
  repair |-> replace_auxiliary end
%% only repair3 is a valid interpretation. That is, 'replace_auxiliary'
%% is the required action
```

Translations

A translation *O* **with** σ renames *O* along σ

- σ is a signature morphism
- in practice, σ is a symbol map, from which one can compute a signature morphism

```
ontology BankOntology =
  Class: Bank  Class: Account ...      end
ontology RiverOntology =
  Class: River  Class: Bank ...        end
ontology Combined =
  BankOntology with Bank |-> FinancialBank
 and
  RiverOntology with Bank |-> RiverBank
    %% necessary disambiguation when uniting OMS
end
```

Making large OMS smaller

Making a large OMS smaller

General problem:

> *you have an OMS over a large signature Σ and want to make it smaller. Say, it should be restricted to $\Sigma' \subseteq \Sigma$.*

DOL provides four options:

- Module extraction
- Approximation
- Reduction
- Filtering

We will discuss these options for two examples:

- the medical ontology SNOMED
- the specification of groups

Module Extraction applied to SNOMED

Question: What does SNOMED say about hearts and heart attacks?

Answer 1:

```
SNOMED extract Heart, HeartAttack
```

extract:

- SNOMED module (sub-ontology of SNOMED)
- capturing the same facts about hearts and heart attacks as SNOMED itself (SNOMED is a conservative extension of the module)
- signature of the module may contain more than heart and heart attack

Dual operation: **`remove`** (lists the symbols to remove)

Approximation applied to SNOMED

Question: What does SNOMED say about hearts and heart attacks?

Answer 2:

```
SNOMED keep Heart, HeartAttack
```

keep:

- captures all logical consequences involving `Heart(Attack)`
- not necessarily a sub-OMS
- may involve new axioms in order to capture the SNOMED facts about hearts and heart attacks
- resulting OMS features exactly the two specified entities, heart and heart attack
- finite axiomatization may be hard to compute, if it exists at all

Dual operation: **`forget`** (lists the symbols to remove)

Reduction applied to SNOMED

Question: What does SNOMED say about hearts and heart attacks?

Answer 3:

```
SNOMED reveal Heart, HeartAttack
```

reveal:

- essentially keeps the whole of SNOMED
- provides some export interface consisting of heart and heart attack only
- while symbols are hidden, the semantic effect of sentences (also those involving these symbols) is kept
- useful when interfacing SNOMED with other ontologies, e.g. in an interpretation.

Dual operation: **hide** (lists the symbols to remove)

Filtering applied to SNOMED

Question: What does SNOMED say about hearts and heart attacks?

Answer 4:

```
SNOMED select Heart, HeartAttack
```

select:

- simply removes all SNOMED axioms that involve other symbols then heart and heart attack
- can be computed easily
- might lead to poor ontology, capturing only a small fraction and only the basic facts of SNOMED's knowledge about hearts and heart attacks.

Dual operation: **`reject`** (lists the symbols to remove)

Module Extraction applied to Groups (1)

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem; inv:Elem->Elem
forall x,y,z:elem . x+0=x
                  . x+(y+z) = (x+y)+z
                  . x+inv(x) = 0
remove inv
```

The semantics returns the following theory:

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem; inv:Elem->Elem
forall x,y,z:elem . x+0=x
                  . x+(y+z) = (x+y)+z
                  . x+inv(x) = 0
```

The module needs to be enlarged to the whole OMS.

Module Extraction applied to Groups (2)

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem; inv:Elem->Elem
forall x,y,z:elem . x+0=x
                  . x+(y+z) = (x+y)+z
                  . x+inv(x) = 0
                  . exists y:Elem . x+y=0
remove inv
```

The semantics returns the following theory:

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem
forall x,y,z:elem . x+0=x
                  . x+(y+z) = (x+y)+z
                  . exists y:Elem . x+y=0
```

Here, adding `inv` is conservative.

Approximation applied to Groups

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem; inv:Elem->Elem
forall x,y,z:elem . x+0=x
                  . x+(y+z) = (x+y)+z
                  . x+inv(x) = 0
forget inv
```

The semantics returns the following theory:

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem
forall x,y,z:elem . x+0=x
                  . x+(y+z) = (x+y)+z
                  . exists y:Elem . x+y=0
```

Computing finite interpolants can be hard, even undecidable.

Reduction applied to Groups

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem; inv:Elem->Elem
forall x,y,z:elem . x+0=x        . x+(y+z) = (x+y)+z
                  . x+inv(x)=0
hide inv
```

Semantics: class of all monoids that can be extended with an inverse, i.e. class of all groups. The effect is second-order quantification:

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem;
exists inv:Elem->Elem .
  forall x,y,z:elem . x+0=x
                    /\ x+(y+z) = (x+y)+z
                    /\ x+inv(x)=0
```

Filtering applied to Groups

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem; inv:Elem->Elem
forall x,y,z:elem . x+0=x
                  . x+(y+z) = (x+y)+z
                  . x+inv(x) = 0
reject inv
```

The semantics returns the following theory:

```
sort Elem
ops 0:Elem; __+__:Elem*Elem->Elem
forall x,y,z:elem . x+0=x
                  . x+(y+z) = (x+y)+z
```

Hide – Extract – Forget – Select

	hide/reveal	remove/extract	forget/keep	select/reject
semantic background	model reduct	conservative extension	uniform interpolation	theory filtering
relation to original	interpretable	subtheory	interpretable	subtheory
approach	model level	theory level	theory level	theory level
type of OMS	elusive	flattenable	flattenable	flattenable
signature of result	$= \Sigma$	$\geq \Sigma$	$= \Sigma$	$\geq \Sigma$
change of logic	possible	not possible	possible	not possible
application	specification	ontologies	ontologies	blending

Pros and Cons

	hide/reveal	remove/extract	forget/keep	select/reject
information loss	none	none	minimal	large
computability	depends	good/depends	depends	easy
signature of result	$=\Sigma$	$\geq\Sigma$	$=\Sigma$	$=\Sigma$
conceptual simplicity	simple (but unintuitive)	complex	farily simple	simple

Example for hiding: sorting

Informal specification:

To sort a list means to find a list with the same elements, which is in ascending order.

Formal requirements specification:

```
%right_assoc( __::__ )%
logic CASL.FOL=
spec PartialOrder =
  sort Elem
  pred __leq__ : Elem * Elem
  . forall x : Elem . x leq x %(refl)%
  . forall x, y : Elem . x leq y /\ y leq x => x = y %(antisym)%
  . forall x, y, z : Elem . x leq y /\ y leq z => x leq z %(trans)%
end
spec List =  PartialOrder then
  free type List ::= [] | __::__(Elem; List)
  pred __elem__ : Elem * List
  forall x,y:Elem; L,L1,L2:List
  . not x elem []
  . x elem (y :: L) <=>  x=y \/ x elem L
end
```

Sorting (cont'd)

```
spec AbstractSort =
  List
then %def
  preds is_ordered : List;
        permutation : List * List
  op sorter : List->List
  forall x,y:Elem; L,L1,L2:List
  . is_ordered([])
  . is_ordered(x::[])
  . is_ordered(x::y::L) <=> x leq y /\ is_ordered(y::L)
  . permutation(L1,L2) <=>
            (forall x:Elem . x elem L1 <=> x elem L2)
  . is_ordered(sorter(L))
  . permutation(L,sorter(L))
end
```

Sorting (cont'd)

We want to show insert sort to enjoy these properties.
Formal design specification:

```
spec InsertSort =  List then
  ops insert : Elem*List -> List;
      insert_sort : List->List
  vars x,y:Elem; L:List
  . insert(x,[]) = x::[]
  . x leq y => insert(x,y::L) = x::insert(y,L)
  . not x leq y => insert(x,y::L) = y::insert(x,L)
  . insert_sort([]) = []
  . insert_sort(x::L) = insert(x,insert_sort(L))
end
```

Correctness

Is insert sort correct w.r.t. the sorting specification?

```
interpretation correctness :
     { AbstractSort hide is_ordered, permutation }
  to { InsertSort hide insert }
end
```

Non-monotonicity

Non-monotonic Reasoning

Non-monotonic reasoning =
more premises may lead to fewer conclusions:
If b is a bird, it can fly.
But if b is a bird and a penguin, it cannot fly.

Non-monotonic reasoning is used in defeasible reasoning, default reasoning, abductive reasoning, belief revision, reasoning about subjective probabilities, ...

BUT: logical consequence $\Gamma \models_\Sigma \varphi$ is monotonic!

DOL's way of supporting non-monotonic reasoning:
closed-world assumptions

Closed-World Assumption

- Prop, FOL and OWL employ an open-world semantics
  1. predicates may hold for more individuals than specified in the theory
  2. a model may have more individuals than specified in the theory
  3. more equations than specified in the theory may hold between individuals
- sometimes, a closed-world semantics is useful
  1. predicates only hold for individuals if specified in the theory
  2. a model has only those individuals specified in the theory
  3. only equations specified in the theory hold between individuals
- Minimization (circumscription) addresses 1
- Freeness addresses 1-3
- Both are non-monotonic operations

Minimizations (circumscription)

- O_1 **then minimize {** O_2 **}**
- forces minimal interpretation of non-logical symbols in O_2

```
  Class: Block
  Individual: B1 Types: Block
  Individual: B2 Types: Block DifferentFrom: B1
then minimize {
        Class: Abnormal
        Individual: B1 Types: Abnormal }
then
  Class: Ontable
  Class: BlockNotAbnormal EquivalentTo:
    Block and not Abnormal SubClassOf: Ontable
then %implied
  Individual: B2 Types: Ontable
```

Minimizations

- O_1 **then minimize {** O_2 **}**
- forces minimal interpretation of non-logical symbols in O_2

```
  Class: Block
  Individual: B1 Types: Block
  Individual: B2 Types: Block DifferentFrom: B1
then minimize {
        Class: Normal
        Individual: B2 Types: Normal }
then
  Class: Ontable SubClassOf: Block and Normal
then %implied
  Individual: B1 Types: not Ontable
```

Freeness

- free { O }
- O_1 then free { O }
- forces closed-world conditions 1-3

```
logic OWL
ontology Family_closed =
 free {
   Class: Person        Class: Male < Person
   Individual: john Types: Male
   Individual: mary Types: Person
  }
```

There is only one model
(up to isomorphism):