An Extension to Structural Subsumption Algorithm

Background

The present description logic classification algorithm is based on tableau algorithm, which can provide sound and complete reasoning on very expressive language. This approach solves the subsumption problem by reducing to satisfiablilty test and the worst case computational complexity is nexp-time. Furthermore, it only focuses on the tractablility of a single subsumption test instead of construction taxonomy. However, in most real applications, ontologies do not take full advantage of expression power of OWL and the most important action is to build up taxonomic classification.

Therefore what we need is an algorithm with high efficiency but less expressive to support large scale of taxonomic classification. The requirements are:

- Efficiency enough to support large tbox
- Support language which is expressive enough for practically useful ontology

With above requirements, a potential solution is Structural Subsumption algorithm, which is proved to be an efficient technique for taxonomic classification (p-time). The basic idea of the algorithm is to reuse redundant information of concepts definition to conduct subsumption inference. It has traditionally been limited due to its inability to provide complete reasoning for expressive languages. It is because in expressive language, the combinations of DL constructors or some axioms will make it difficult to setup appropriate comparison rules to support complete inference. However, we could possibly extend the Structural Subsumption Algorithm to achieve a balance between effiiency and expressiveness by adding more comparison rules, making some restrictions on concept definition and axioms.

Preliminary – Structural Subsumption Algorithm for *FL*

$$
\begin{aligned}
C,D \quad \rightarrow \quad & A \mid \text{(atomic concept)} \\
& \top \mid \text{(universal concept)} \\
& \bot \mid \text{(bottom concept)} \\
& C \sqcap D \mid \text{(intersection)} \\
& \forall R.C \mid \text{(value restriction)}
\end{aligned}
$$

The algorithm proceeds in two phases:

1. Normalize

   An *FL*0-concept description is in *normal form* iff it is of the form

$$A_1 \sqcap \ldots \sqcap A_m \sqcap \forall R_1.C_1 \sqcap \ldots \sqcap \forall R_n.C_n$$

2. Proposition: Let

$$A_1 \sqcap \ldots \sqcap A_m \sqcap \forall R_1.C_1 \sqcap \ldots \sqcap \forall R_n.C_n$$

be the normal form of the *FL*0-concept description C, and

$$B_1 \sqcap \ldots \sqcap B_k \sqcap \forall S_1.D_1 \sqcap \ldots \sqcap \forall S_l.D_l$$

the normal form of the *FL*0-concept description D, then $C \sqsubseteq D$ iff the following conditions holds:

i) for all i, $1 \leqslant i \leqslant k$, there exists j, $1 \leqslant j \leqslant m$, such that $B_i = A_j$
ii) for all i, $1 \leqslant i \leqslant l$, there exists j, $1 \leqslant j \leqslant m$, such that $S_i = R_j$ and $C_i \sqsubseteq D_j$

The structural algorithm has been extended to ALN with soundness & completeness. Please refer the attachment for detail algorithm.

Reference
1. *The simplest Structural Description Logic*
2. *Structural Subsumption Considered from an Automata Theoretic Point of View - Baader*

Extended Structural Subsumption Algorithm

Language definition currently supported

Concepts: (Cyclic concept definitions are not supported)

C, D → A (atomic concept),
$\top$ (universal concept),
$\bot$ (bottom concept),
$C \sqcap D$ (intersection),
$C \sqcup D$ (union),
$\exists R.C$ (some value from restriction)
$\forall R.C$ (all value from restriction),
$\exists R.\{x\}|$ (hasValue)

Axioms:

Axioms → $C \sqsubseteq D$ (concept inclusion),
$R \sqsubseteq S$ (role inclusion)

Basic Idea

From the Structural Subsumption Algorithm for *FL* we can see that there two major steps to compare two concepts. The first step is to transform concepts into standard forms to ensure structural comparisons can be preceded in a standard way. The second one is recursively compared sub-constitutes of each concept.

Following the basic idea, we can extend the structural subsumption algorithm for more expressive language by defining more comparison criteria. For an ontology without acyclic definition, every defined concept can be deemed as restrictions on some properties (atom concepts can be seem as a kind of "special" restriction). For example, $C \equiv A \sqcap B \sqcap \forall R.(\forall S.D) \sqcap \forall R.C$ can be seem as a concept with restriction on R_A, R_B , R (R_A, R_B is special restriction brought by A, B).

Restrictions provide comparison criteria for subsumption test. Restrictions on same property or its sub-property are basic components that can be compared. Table 1 lists the basic comparison rules for restrictions.

Concept A	Concept B	Condition of $A \sqsubseteq B$
$\exists R.C$	$\exists S.D$	Iff $R \preceq S$ and $C \sqsubseteq D$
$\forall R.C$	$\forall S.D$	Iff $S \preceq R$ and $C \sqsubseteq D$
$\geqslant nR.C$	$\geqslant mS.D$	Iff $R \preceq S$ and $C \sqsubseteq D$ and $n \leqslant m$
$\leqslant nR.C$	$\leqslant mS.D$	Iff $S \preceq R$ and $D \sqsubseteq C$ and $n \geqslant m$

Table 1 Basic Comparison Rules for Restrictions

To decide whether two concepts are subsumed by each other, we can first normalize them into comparable restrictions and then compare their sub constitutes recursively. Table 2 lists a set of recursive rules for comparing two concepts. Figure 1 shows an example of how such rules works.

Concept A	Concept B	Condition of $A \sqsubseteq B$
Primitive	Primitive	Iff B is in the $\sqsubseteq$ transitive closure of A
Primitive	Conjunctive	Iff for **every** constitute B_i of B, $A \sqsubseteq B_i$
Primitive	Disjunctive	Iff for **some** constitute B_i of B, $A \sqsubseteq B_i$
Conjunctive	Primitive	Iff for **some** constitute A_i of A, $A_i \sqsubseteq B$
Conjunctive	Conjunctive	Iff for **every** constitute B_i of B , there is **some** A_j of A, $A_j \sqsubseteq B_i$
Conjunctive	Disjunctive	Iff for **some** constitute of A_i of A, there is **some** constitute B_i of B, $A_j \sqsubseteq B_i$
Disjunctive	Primitive	Iff for **every** constitute A_i of A, $A_i \sqsubseteq B$
Disjunctive	Conjunctive	Iff for **every** constitute A_i of A, and **every** constitute B_i of B, $A_j \sqsubseteq B_i$
Disjunctive	Disjunctive	Iff for **every** constitute A_i of A, there is **some** constitute B_i of B, $A_j \sqsubseteq B_i$

Table 2 Recursive Comparison Rules

Given $A \equiv \exists R1.C \sqcap \exists R2.(\forall R4.D) \sqcap \forall R3.E$, $B \equiv \exists R2.(\forall R4.F) \sqcap \forall R3.G$, $D \sqsubseteq F$, $E \sqsubseteq G$

It can be concluded that $A \sqsubseteq B$

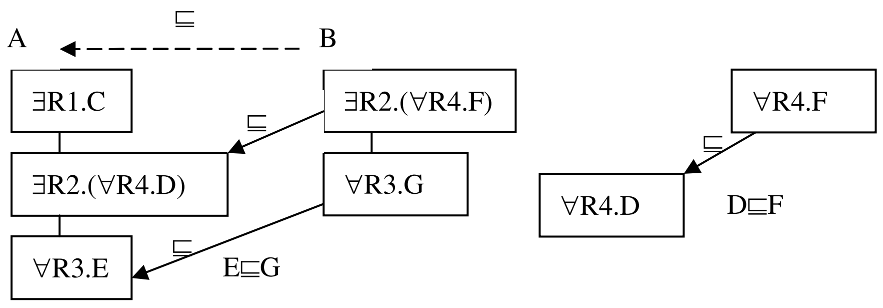


Figure 1 An Example

Interrelations among restrictions

The basic comparison rules and the recursive comparison rules work well when restrictions in a concept are independent. But in many cases restrictions on the same property are interrelated with each other. In such situation the comparison rules will be invalid. Figure 2 shows such an example.

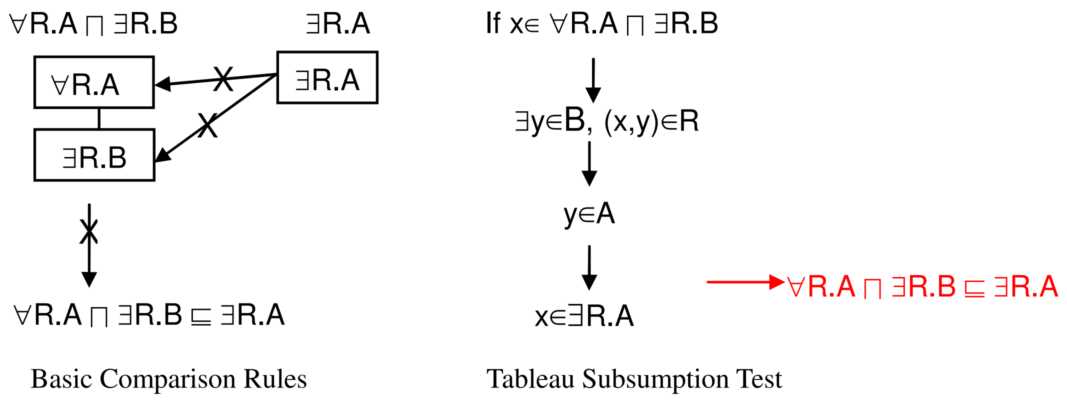

$\forall R.A \sqcap \exists R.B$
$\exists R.A$
$\forall R.A$
$\exists R.A$
$\exists R.B$
$\forall R.A \sqcap \exists R.B \sqsubseteq \exists R.A$
Basic Comparison Rules
If $x \in \forall R.A \sqcap \exists R.B$
$\exists y \in B, (x,y) \in R$
$y \in A$
$x \in \exists R.A$
$\forall R.A \sqcap \exists R.B \sqsubseteq \exists R.A$
Tableau Subsumption Test

Figure 2

In the example, "$\exists$R.B" in "$\forall$R.A$\sqcap\exists$R.B" interacts with "$\forall$R.A", so that "$\forall$R.A$\sqcap\exists$R.B" can be concluded as sub class of "$\exists$R.A". But only by the basic comparison rules and the recursive comparison rules we can not get such conclusion. To solve the problem, we define four transformation rules for normalization.

- R1: $\forall R.A \sqcap \forall S.B \equiv \forall R.A \sqcap \forall S.(A \sqcap B)$ $\quad S \preceq R$
- R2: $\exists R.A \sqcup \exists S.B \equiv \exists R.A \sqcup \exists S.(A \sqcup B)$ $\quad S \preceq R$
- R3: $\forall R.A \sqcap \exists S.B \equiv \forall R.A \sqcap \exists S.(A \sqcap B)$ $\quad S \preceq R$
- R4: $\exists R.A \sqcup \forall S.B \equiv \exists R.A \sqcup \forall S.(A \sqcup B)$ $\quad S \preceq R$

Therefore, in the example, "$\forall$R.A$\sqcap\exists$R.B" will be transformed to "$\forall$R.A$\sqcap\exists$R.(A$\sqcap$B)" after normalization. And then we can apply comparison rules to draw correct conclusion.

When performing a subsumption test, we use a classification tree to cache the inter-result for convenience. If we step further, we can leverage the information of the classifcation tree to design an algorithm for taxonomic classification. Therefore, the steps to decide whether class A is subsumed by class B will become:

1. classify A, insert A and all its sub constitutes into classification tree
2. classify B, insert B and all its sub constitutes into classification tree
3. decide whether A is subsumed by B according to the classification tree

Algorithm Description

buildClassificationTree

```
1.  Building Properties Hierarchy
2.  Insert ⊥, ⊤ into classification Tree(CT)
3.  for each axiom $C \sqsubseteq D$ {
        classify(C)
        classify(D)
```

```
    addSubsumptionLink(C, D) // it will remove outdated links
  }
```

classify(C)

```
1. Normalize(C) // normalize C to a standard form to perform classify
2. if C is already in CT, return
3. classify(Ci) //Add sub constitute Ci of C to CT
4. // Determine the position of C in Classification Tree
   mss <- Most specific subsumer of C according to current Classification Tree
   mgs <- Most general subsumee of C according to current Classification Tree
5. //link each p in mss as directly father; each p in mgs as directly son
   foreach (v in mss){
     addSubsumptionLink (v, C); // it will remove outdated links
   }
   foreach(v in mgs){
     addSubsumptionLink (C, v); // it will remove outdated links
   }
```

getMSS(C) // getMGS is similar

```
1. C in form of $C1 \sqcup \ldots \sqcup Cn$    // $C1 \sqcap \ldots \sqcap Cn$ is similar
   a) superClassList ← all nodes D which is a common super class of C1… Cn
   b) If A, B are in superClassList, and $A \sqsupseteq B$, remove A from superClassList
2. C in form of $\exists R.D$    // $\forall R.D$ is similar
   a) superClassList ←all nodes that satisfied with the form $\exists R.E$, $E \sqsupseteq D$
   b) If A, B are in superClassList, and $A \sqsupseteq B$, remove A from superClassList
```

As we can see from the pseudo codes, the algorithm will start with building the classification tree. Given an axiom $C \sqsubseteq D$, the algorithm will first recursively classify C, D and all the sub constitutes until all of them have been correctly linked into the classification tree. And then it will add the subsumption link between C and D which will automatically remove outdated links. Figure 3 shows an example of classifying an ontology which contains the following definitions:

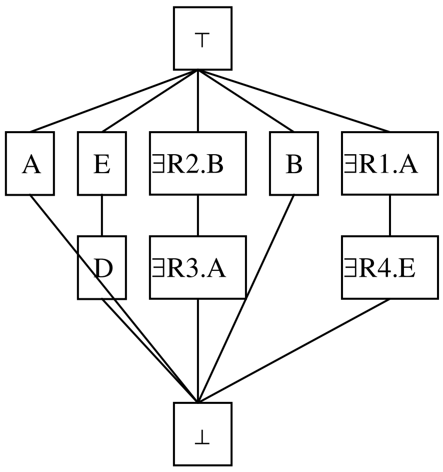
$C \equiv \exists R3.(A \sqcap B) \sqcap \exists R4.D$

$F \equiv \exists R1.A \sqcap \exists R2.B$

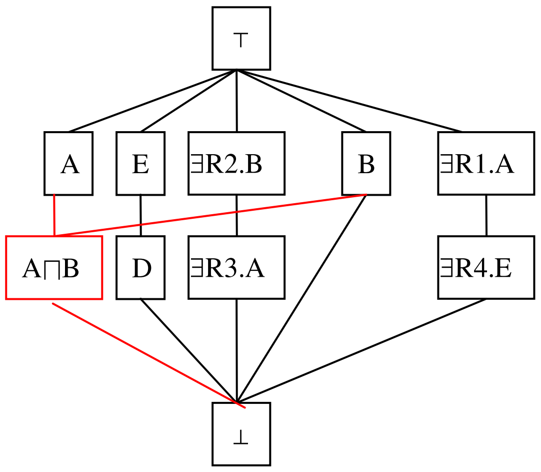
$D \sqsubseteq E$,

$\exists R3.A \sqsubseteq \exists R2.B$

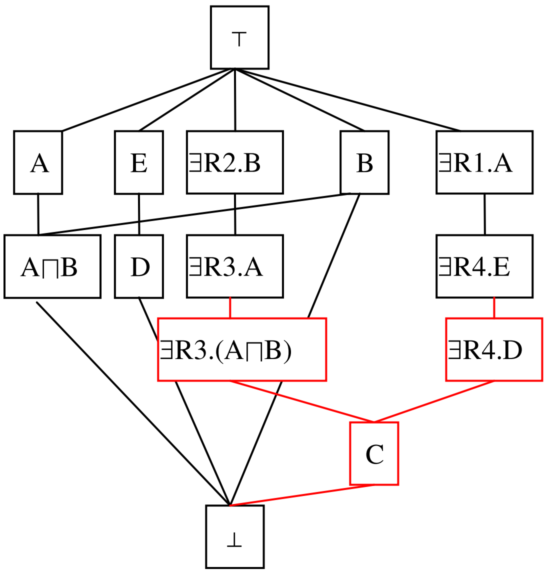
$\exists R4.E \sqsubseteq \exists R1.A$



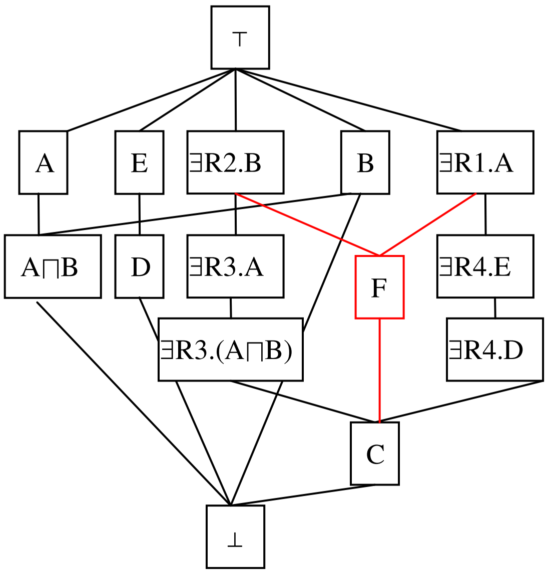
1. Initial classification tree, D ⊑ E, ∃R3.A ⊑ ∃R2.B, ∃R4.E ⊑ ∃R1.A



2. Recursively classify C: Adding A⊓B



3. Recursively classify C: Adding ∃R3.(A⊓B), ∃R4.D and C



4. Classify F

Figure 3 An example of building classification tree

Completeness & Soundness

It can be proved that the structural subsumption algorithm is sound but not complete. The incompleteness of the algorithm mainly comes from the TBox axioms. For example, we can use TBox axioms to define:

$C \sqsubseteq \exists R.B, \quad C \sqsubseteq \forall R.A$

We can see that there is no rules in algorithm to identify the relationship between C and ∃R.(A⊓B), which in fact can be proved that $\exists R.(A \sqcap B) \sqsupseteq (\exists R.B \sqcap \forall R.A) \sqsupseteq C$

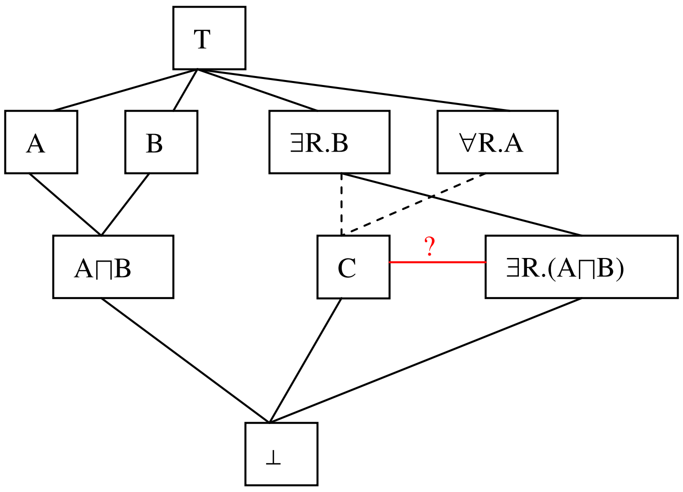


Figure.4 A Counter Example

Though the extended structural subsumption algorithm is not complete, but it can still achieve satisfactory results for subsumption reasoning in many real cases because:

- Many OWL constructs have not been used in real ontologies. 80% of ontologies only use 20% of OWL constructs. There are many of them only use atom class and subClassOf
- In many cases, even there are constructs that EODM reasoner can not support in ontologies, the extended structural subsumption algorithm can still get the correct answers because those constructs are usually not interact with others in concept definitions. In other words, such constructs act as primitive class during the component comparsion.
- The counter situations that the extended structural subsumption algorithm can not support are rare in real ontologies

Test of the Structural Subsumption Reasoner

Test Description

The goal of this test is to evaluate the efficiency and correctness of EODM Structural Subsumption Reasoner. The test can be divided into 2 kinds

- Download Ontologies Test
- Random Ontologies Test

Two measures will be considerate in evaluating the efficiency of the reasoner:

- Load time (time for classifying an ontology)
- Query time - Query time will be evaluated by calling **getDescendantClass** method for all named classes in an ontology (Import classes are also included in model, but will not be used for queries) and calculate the average responding time. This time will be deemed as a standard to evaluate efficiency of reasoner.

Testing with Download Ontologies

In this kind of test, we download a number of ontologies from protégé ontology library. Based on these ontologies, a series of testing are performed.

	Num Of Classes	Eodm Load (ms)	Eodm Query Avg. (ms)	Query Completeness	Query Soundness
generations.owl An ontology about family relationships that demonstrates classification. DL Expressivity: ALCOIF	18	31	0	100%	100%
ka.owl — Defines concepts from academic research. DL Expressivity: AL(D)	96	31	0	100%	100%
not-galen.owl — A selective adaptation made in 1995 of an early prototype GALEN model; content is not related to or representative of any current or historical OpenGALEN release DL Expressivity: SHF	3097	61768	98	89.6%	100%
koala.owl — A simple ontology about humans and marsupials DL Expressivity: ALCON(D)	20	251	0	100%	100%

Note:
1) 0 means the responding time is less than 1ms

Testing with Random Ontologies

Random Ontology Generated Rules:
During this test, random ontologies will be generated according to the capability of EODM Reasoner. This test mainly evaluates the efficiency of EODM Reasoner. Random ontologies will be generated according to 7 parameters:

- Number of atoms (Atoms): Number of atom classes which will appear in generated ontology.
- Depth of Nesting Expression (Depth): The max depth of an owl class expression. The depth of an atom class is 0.
- Number of properties (Props): Number of properties which will appear in generated ontology. No domain and range will be claimed.
- Number of individuals (Inds): Number of individual which will appear in generated ontology. And they will randomly be claimed to belong to some atom class.
- Rate of subclasses (R_SUBC): the possibility that an owl class expression can have sub class with same depth. (No cyclic definitions)
- Rate of subProperties (R_SUBP): the possibility that an owl property can have sub

property. (No cyclic definitions)

- Rate of equivalent classes (R_EQ): the possibility that an owl class expression can have equivalent classes with same depth.

In an OWL Expression, 5 types of OWL constructor may possibly appear: intersection, union, someValueFrom, AllValueFrom and hasValue. The total number of owl classes in random ontology will be Atoms * Depth.

Test Results:

During the test, every group of tests will be performed 10 times. Below parameters will be fixed:

- R_SUBC = 1/3
- R_SUBP = 1/4
- R_EQ = 1/6
- Inds = 20

Scale: (ms)

Atoms/Props/Depth = 20/3/2   40 classes

	Loading (avg)	Loading (max)	Query (avg)	Query (max)	Query Soundness	Query Completeness
EODM Reasoner	79	281	0	16	100%	100%

Atoms/Props/Depth = 40/3/2

	Loading (avg)	Loading (max)	Query (avg.)	Query (max)	Query Soundness	Query Completeness
EODM Reasoner	64	94	0	16	100%	100%

Atoms/Props/Depth = 10/3/3   30 classes

	Loading (avg)	Loading (max)	Query (avg.)	Query (max)	Query Soundness	Query Completeness
EODM Reasoner	102	375	0	16	100%	100%

Atoms/Props/Depth = 20/3/3   60 classes

	Loading (avg)	Loading (max)	Query (avg.)	Query (max)	Query Soundness	Query Completeness
EODM Reasoner	221	797	0	16	100%	100%

Atoms/Props/Depth = 100/3/2   200 classes

	Loading (avg)	Loading (max)	Query (avg.)	Query (max)	Query Soundness	Query Completeness

EODM Reasoner	576	766	0	16	100%	100%

Atoms/Props/Depth = 200/20/5   1000 classes

	Loading (avg)	Loading (max)	Query (avg.)	Query (max)	Query Soundness	Query Completeness
EODM Reasoner	8826	31673	3	93	100%	100%

Atoms/Props/Depth = 1000/20/3   3000 classes

	Loading (avg)	Loading (max)	Query (avg.)	Query (max)	Query Soundness	Query Completeness
EODM Reasoner	22327	98431	3	125	100%	100%