A Proposal for a Glass-Box Approach for Subsumption Checking

Quentin Elhaik, Marie-Christine Rousset *, Marie-Claude Gaudel
L.R.I U.R.A C.N.R.S, University of Paris-Sud
Building 490,91405 Orsay Cedex, France
mcr@lri.lri.fr

1 Motivation

Description logics have good representational properties and their computational properties are well understood. However, one of their main drawbacks is that their current reasoning algorithms have an inherent black-box nature, i.e. they don’t make explicit how subsumption inferences are obtained.

This is an important drawback for providing explanation or debugging services. Description logics with a reasonable expressive power contain different constructors the combination of which can lead to numerous, complex and varied forms of subsumption relation. Consequently, some subsumption relations are complex and need to be explained to a user or to a terminology designer. The point is not to explain to users or designers how subsumption is computed but how a complex subsumption inference can be decomposed into different simpler inference steps. For KBS, rules have shown to be a formalism that is well appropriate to support trace capabilities for both explanation and debugging purposes.

More generally, as previously advocated by [1] and [6], specifying the subsumption relation with a set of inference rules would have many advantages for characterizing, customizing or optimizing implementations of subsumption algorithms. In addition, it could be the basis for defining different kinds of relevance (e.g., relevance of a fact to a query) and new ways of reasoning (e.g., abductive reasoning) in a description logics setting. The major difficulty is to obtain a complete set of inference rules characterizing the subsumption relation.

In this paper, we consider a rather expressive description logics language, including the two designated primitive concepts $\top$ (top) and $\bot$ (bottom), and the constructors $\cap$ (conjunction of concepts), $\neg$ (negation, for primitive concepts only), $\forall$ (universal role quantification), $\exists$ (existential role quantification) and qualified number restrictions. The possibility to express qualified number restrictions (denoted $(\geq nRC)$ and $(\leq nRC)$) has been very useful for a real application of modeling components of nuclear plants ([5]). Note that qualified number restrictions are more expressive than simple number restrictions.

In section 2, we summarize the glass-box consistency checking method that we proposed in [5], in order to provide a trace of some of the subsumption steps leading to an inconsistency as a trace-tree. In the sections 3 and 4 we describe a recent work [2], based on establishing a correspondence between complex concepts in our language and record types, which enabled us to obtain a sound and complete set of inference rules to characterize subsumption in our language.

2 A glass-box consistency checking method based on rules

The glass-box approach that we proposed in [5] is quite similar in spirit to [3] providing explanations for inferences in CLASSIC. Our approach can be used as a complement of existing subsumption algorithms, both to possibly complete existing subsumption algorithms and to provide traces for the most subtle subsumption inferences. The basic assumption of our approach is that the subsumption relations that are the most difficult to understand for a terminology designer are the subsumption relations that are not structural, in particular those stating that a conjunction of concepts is subsumed by a concept that does not not subsume any of the

---

*correspondence has to be sent to this author
conjects. Those schemas of subsumption relations are made explicit as trace rules. Each trace rule has pattern conditions and interpreted conditions. The pattern conditions of the rule represent the simple conjuncts of a complex concept, and its conclusion represents a subsuming simple concept. Its interpreted conditions express constraints that have to be satisfied for the schema subsumption to hold.

A trace rule represents a one-step not structural subsumption. A trace tree summarizes different steps of subsumption, which, when they are chained, account for a complex subsumption. Given a set of trace rules, and a subsumption query $C_1 \sqcap \ldots \sqcap C_k \preceq C$, building a trace tree can be done by applying any standard (forward chaining or backward chaining) inference procedure for rules.

We have implemented a forward chaining procedure in Prolog. Currently, we have a set of trace rules that contains about 15 rules with variables. Though they are not complete, they cover many cases of not structural subsumption relations. In particular, they were very useful to detect complex and hidden inconsistencies of an application on model-based inference rules. For instance, $(x : Int \times y : Bool)$ is a record type with two fields which are named $x$ and $y$ and whose types are integer (Int) and boolean (Bool) respectively.

Though many differences exist between types and concepts, there is a similarity between a conjunction of complex concepts and a record type. Syntactically speaking, complex concept expressions are mainly defined by compositions of role names each of them coming with a concept expression (and a restriction on that concept expression). If we identify role names with field names and concept names with types, this has a flavor of records. However, we have to distinguish the different restrictions which can be associated with each concept expression. For doing so, we introduced quantified types in our universe of types. Our universe of types is then defined by associating:

- a basic type $T_C$ with every primitive concept $C$; we will denote $\bot$ and $\top$ the two special types associated with the bottom and top concepts.
- a record type with every non primitive concept. In our description logics language, every non primitive concept can be seen as a conjunction of concept expressions, which are of the form $(\forall R.C)$, $(\geq n R.C)$, or $(\leq n R.C)$, where $R$ is a role name and $C$ is a concept expression.\footnote{We introduce special role $\text{Id}$ (whose the semantics is the identity relation) in order to make any primitive concept $C$ equivalent to $(\forall \text{Id}.C) \cap (\geq 1 \text{Id}.C)$} We will say that $C$ is the concept expression and that $\forall$ (respectively $(\geq n)$, $(\leq n)$) is the restriction, which are associated with $R$ in $(\forall R.C)$ (respectively $(\geq n R.C)$, $(\leq n R.C)$).

The type associated with a non primitive concept is a record type where

- the field identifiers are the role names,
- the type of a field $R$ is a quantified type, which is obtained by tagging the type of the concept expression associated with $R$ by its restriction,

$$t_1 \preceq t_2$$ is inferred if a proof of $e : t$ (respectively, $t_1 \ll t_2$) can be built from the set of inference rules and axioms. The classical subtyping rule for record types is given in Appendix. Let us recall that a record type is a collection of typed fields identifiers. For instance, $(x : Int \times y : Bool)$ is a record type with two fields which are named $x$ and $y$ and whose types are integer (Int) and boolean (Bool) respectively.

### 3 Characterizing the subsumption relation by a set of inference rules

Our approach is based on defining a type system and its associated typing rules for our description logics language. The type system of a programming language is based on a set of predefined types (e.g. booleans, integers) and some type constructors (e.g. records) which are used to combine types into more complex ones. Some type systems include a subtyping relation as a partial order on types, satisfying the so-called "substitution property".

Traditionally, in type systems, typing and subtyping are defined through a set of inference rules and axioms. They express predefined typing and subtyping statements and basic typing and subtyping inferences. The type $t$ of a complex expression $e$ (respectively a subtyping relation between 2 types $t_1$ and $t_2$) is inferred if a proof of $e : t$ (respectively, $t_1 \ll t_2$) can be built from the set of inference rules and axioms. The classical subtyping rule for record types is given in Appendix. Let us recall that a record type is a collection of typed fields identifiers. For instance, $(x : Int \times y : Bool)$ is a record type with two fields which are named $x$ and $y$ and whose types are integer (Int) and boolean (Bool) respectively.

### 3 Characterizing the subsumption relation by a set of inference rules

Our approach is based on defining a type system and its associated typing rules for our description logics language. The type system of a programming language is based on a set of predefined types (e.g. booleans, integers) and some type constructors (e.g. records) which are used to combine types into more complex ones. Some type systems include a subtyping relation as a partial order on types, satisfying the so-called "substitution property".

Traditionally, in type systems, typing and subtyping are defined through a set of inference rules and axioms. They express predefined typing and subtyping statements and basic typing and subtyping inferences. The type $t$ of a complex expression $e$ (respectively a subtyping relation between 2 types $t_1$ and $t_2$) is inferred if a proof of $e : t$ (respectively, $t_1 \ll t_2$) can be built from the set of inference rules and axioms. The classical subtyping rule for record types is given in Appendix. Let us recall that a record type is a collection of typed fields identifiers. For instance, $(x : Int \times y : Bool)$ is a record type with two fields which are named $x$ and $y$ and whose types are integer (Int) and boolean (Bool) respectively.
when there are several occurrences of the same role $R$, there is only one field $R$, whose type is the intersection of the associated quantified types.

As an example, let us consider the concept:

\[(\geq 2 \text{ associate Company}) \cap (\geq 1 \text{ employee FullTime}) \cap (\leq 10 \text{ employee FullTime})].\]

Its type is:

\[
\begin{align*}
\text{associate} : & \ (\geq 2)T_{\text{Company}} \\
\times \text{employee} : & \ (\geq 1)T_{\text{FullTime}} \cap (\leq 10)T_{\text{FullTime}}.
\end{align*}
\]

Note that the two occurrences of the role employee in the concept expression were translated in a unique field name in the record, associated with a conjunction of two quantified types.

The set of inference rules that defines typing and subtyping in our universe of types is given in [2]. It is composed of 21 rules for defining type equivalence and 18 rules for defining subtyping (including 3 rules for defining type incompatibility). It is also proved in [2] that the subtyping relation of our type system completely characterizes the subsumption relation of our description logics language. The compactness of our set of inference rules relies on two main points. First, the structural subtyping is expressed by a unique rule corresponding to the standard subtyping rule on records (recalled in appendix). Second, we capture the non structural subsumptions relations caused by combinations of qualified number restrictions by two rules (see Rule (a) and Rule (b) in the appendix). On one hand, those two rules express that conjunctions of number restrictions of a same role $R$ (i.e. $(\geq n_1 R C_1) \cap \ldots \cap (\geq n_m R C_m) \cap (\leq p_1 R D_1) \cap \ldots \cap (\leq p_k R D_k) \leq (\leq p R D)$) are subsumed by number restrictions of the form $(\geq n R C)$ and $(\leq p R C)$. On the other hand, the minimal number $n$ and the maximal number $p$, which characterize them, depend on the relations of subsumption and of disjointness existing between the different combinations of subconcepts involved in the conjunction of number restrictions. In order to avoid to duplicate the number of inference rules dealing with all those different cases (that would vary on the different conditions for applying the rules), we chose to determine the minimal number $n$ and the maximal number $p$ by two algorithms. They are summarized in the following section 4 (see [2] for more details). The general idea is to consider the restriction numbers $n_1, \ldots, n_m, p_1, \ldots, p_k$ as cardinality constraints to be satisfied by distributions of objects among sets.

We are currently investigating how this approach can be extended to handle Abox reasoning as well.

4 The algorithms of Minimization and Maximization

Let consider a concept expression $E$ of the form:

\[(\geq n_1 R C_1) \cap \ldots \cap (\geq n_m R C_m) \cap (\leq p_1 R D_1) \cap \ldots \cap (\leq p_k R D_k).\]

We will call its at-least list (respectively, its at-most list), the list of pairs \{(C_1, n_1), \ldots, (C_m, n_m)\} (respectively the list of pairs \{\{D_1, p_1\}, \ldots, \{D_k, p_k\}\}).

Let denote $n_{\min}$ the minimal number $n$ such that $E \leq (\geq n R C)$ (if $C$ is a common subsumer of $C_1 \ldots C_m$, and $p_{\max}$ the maximal number $p$ such that $E \leq (\leq p R D)$) if $D \leq D_{i_1} \cup \ldots \cup D_{i_j}$ 2). Those two numbers depend on the disjointness relations existing between the different concepts $C_i$ and $D_j$.

A simple case for determining $n_{\min}$ (resp. $p_{\max}$) corresponds to the case where all the concepts $C_i$ are pairwise disjoint and $k = 0$ (resp. all the concepts $D_j$ are pairwise disjoint and $m = 0$). In that case, $n_{\min} = n_1 + \ldots + n_m$ (resp. $p_{\max} = p_1 + \ldots + p_k$).

However, determining $n_{\min}$ and $p_{\max}$ cannot be reduced to those simple cases, even when $k = 0$ or $m = 0$. Let us illustrate it by two examples corresponding to the case of concepts $C_1 \ldots C_m$ which are not pairwise disjoint but which are disjoint altogether.

Example 4.1: First, let us consider the three concepts $A$, $B$ and $C$ defined as follows.

- $A := (\geq 1 \text{ employee Fulltime}) \cap (\leq 10 \text{ employee Fulltime})$
- $B := (\geq 5 \text{ employee Fulltime}) \cap (\leq 20 \text{ employee Fulltime})$
- $C := (\geq 15 \text{ employee Fulltime}) \cap (\leq 50 \text{ employee Fulltime})$

They are not pairwise disjoint : $A \cap B$ and $B \cap C$ are non empty intersections, while $A \cap C$ is an empty intersection. It can be shown that the minimal $n$ such that $(\geq 2 R A) \cap (\geq 3 R B) \cap (\geq 4 R C) \leq (\geq n R T)$ is: $n_{\min} = 6$.

Now, let us consider $C_1, C_2$ and $C_3$ three concepts

\[\text{Example 4.1: First, let us consider the three concepts } A, B \text{ and } C \text{ defined as follows.}
\]

\[\begin{align*}
A & := (\geq 1 \text{ employee Fulltime}) \cap (\leq 10 \text{ employee Fulltime}) \\
B & := (\geq 5 \text{ employee Fulltime}) \cap (\leq 20 \text{ employee Fulltime}) \\
C & := (\geq 15 \text{ employee Fulltime}) \cap (\leq 50 \text{ employee Fulltime})
\end{align*}\]

They are not pairwise disjoint: $A \cap B$ and $B \cap C$ are non empty intersections, while $A \cap C$ is an empty intersection. It can be shown that the minimal $n$ such that $(\geq 2 R A) \cap (\geq 3 R B) \cap (\geq 4 R C) \leq (\geq n R T)$ is: $n_{\min} = 6$.

Now, let us consider $C_1, C_2$ and $C_3$ three concepts
such that \( C_1 \cap C_2 \), \( C_2 \cap C_3 \) and \( C_1 \cap C_3 \) are nonempty intersections, while \( C_1 \cap C_2 \cap C_3 \equiv \perp^3 \). It can be shown that the minimal \( n \) such that \( (\geq 2 R C_1) \cap (\geq 3 R C_2) \cap (\geq 4 R C_3) \leq (\geq n R \top) \) is: \( n_{\min} = 5 \).

That example shows that it is crucial to determine all the nonempty intersections among the concepts \( C_1 \ldots C_m \), in order to determine the parameter \( n_{\min} \). To do so, the inferences rules are used: for all the intersections \( C_{i_1} \cap \ldots \cap C_{i_t} \), it is checked whether the inference rules can infer that the concept \( C_{i_1} \cap \ldots \cap C_{i_t} \) is subsumed by the concept \( \perp \). In general, things can be made even more complicated when the at-most restrictions can interfere with determining the parameter \( n_{\min} \). It is illustrated in the following example.

**Example 4.2:** In addition to the three concepts \( A \), \( B \) and \( C \) of our previous example, let us consider the concepts \( D_1 \), defined as \((\geq 5 \text{ employee FullTime}) \cap (\leq 10 \text{ employee FullTime}) \cap (\leq 10 \text{ employee FullTime}) \cap (\geq 20 \text{ employee FullTime}) \). It can be shown that the minimal \( n \) such that \( (\geq 2 R A) \cap (\geq 3 R B) \cap (\geq 4 R C) \cap (\leq 1 R D_1) \cap (\leq 1 R D_2) \leq (\geq n R \top) \) is: \( n_{\min} = 7 \).

**The algorithms:**

The input of the two algorithms is an at-least list \( \{ (C_1, n_1), \ldots, (C_m, n_m) \} \) and an at-most list \( \{ (D_1, p_1), \ldots, (D_k, p_k) \} \). The general idea of our algorithms is to consider the at-least and at-most lists as constraints to be satisfied by distributions of objects among concepts.

The Minimization (resp. Maximization) algorithm has to minimize (resp. minimize) the number of objects to distribute among the concepts \( C_1 \ldots C_m \), while satisfying its input at-least and at-most constraints.

Our algorithms follow a *generate and test* schema.

- First, Minimization (respectively Maximization) generates all the distributions of \( r \) objects among the concepts \( C_1 \ldots C_m \) (respectively \( D_1 \ldots D_k \)), for all \( r \) such that \( \max \{ n_1, \ldots, n_m \} \leq r \leq n_1 + \ldots + n_m \) (respectively \( \max \{ p_1, \ldots, p_k \} \leq r \leq p_1 + \ldots + p_k \)).

- Second, they both check whether each distribution has been generated at the previous step satisfies the input at-least and at-most constraints.

Finally, Minimization (respectively Maximization) computes the minimal (respectively, maximal) cardinal number \( r \) of the distributions which satisfied the constraints that were checked at the previous step. If no distribution satisfies the constraints, the initial concept expression is declared inconsistent.

The second step is quite subtle and requires some further explanations. A distribution of objects among a set of concepts is characterized by the number of objects that are distributed in each concept. However, the different concepts are not necessarily disjoint. Therefore, several objects in a given distribution can be common to several concepts. The assignment of a certain number of objects to a concept has to satisfy not only the at-least and the at-most constraints on this concept but also the at-least and the at-most constraints on all the concepts which have a non empty intersection with it.

In addition, some constraints are explicit (those appearing explicitly in the input at-least and at-most lists) but some others can exist implicitly, as a consequence of the explicit ones. Consequently, in order to obtain the implicit constraints that they need to check, the Minimization and Maximization algorithms call each other on subsets of their inputs.

The base case of application of the algorithm Minimization (resp. Maximization) corresponds to the case where the input at-most (resp. at-least) constraints are empty. In that case, no implicit constraint needs to be checked.

The full description of the algorithms ([2]) include several optimization heuristics which aim at generating only distributions that satisfy the implicit constraints, and to order them.

It is important to note that the Minimization and Maximization algorithms resort to the inference rules to check disjointness relations in order to determine the set of non empty intersections between concepts, and to check subsumption relations in order to determine whether implicit constraints have to be taken into account.

\[^3\text{such concepts can be built in our language (see [2] for an example)}\]
5 Discussion

The advantages of a proof-theoretic approach for characterizing subsumption have been advocated by [1] and [6]. In [1], structural subsumption for classical is defined by inference rules. The analogy between conjunction of concepts and record types is also mentioned but is not systematically exploited. The approach of [6] is completely different from ours since it is based on translating description logics expressions into first order formulas and on using the inference rules of sequent calculus, which is known to be sound and complete for first order logic. As a result, they obtain more than one hundred rules to characterize subsumption in BACK.

Other works like [4] are based on an axiomatization of description logics in graded modal logics ([7]). Proofs of subsumptions can then be obtained. However, they involve formulas that result from translation of concept descriptions into modal logic formulas, themselves translated into predicate logic formulas. Furthermore, the steps of the proofs are not subsumption steps but resolution steps and unification. Consequently, neither the formulas nor the proof steps are meaningful for a user in terms of description logics constructors and inferences. Therefore, the axiomatization of description logics in modal logic does not seem to be appropriate to be the basis for a transparent support for subsumption inferences.

6 Appendix

Subtyping rule for record types (up to role permutations):

\[
\Gamma \vdash T_1 \ll T_1 \quad \ldots \quad \Gamma \vdash T_k \ll T_k
\]

\[
\Gamma \vdash \langle R_1 : T_1 ; \ldots ; R_k : T_k ; R_{k+1} : T_{k+1} ; \ldots ; R_n : T_n \rangle \ll \langle R_1 : T_1 ; \ldots ; R_k : T_k \rangle
\]

Rule (a):

\[
C_1 \preceq C, \ldots , C_m \preceq C
\]

\[
(\geq n_1 RC_1) \cap \ldots \cap (\geq m RC_m) \cap (\leq p_1 RD_1) \cap \ldots \cap (\leq p_k RD_k) \preceq (\geq n RC)
\]

(where n is computed by the Minimization algorithm described in section 4)

Rule (b):

\[
D \preceq D_1 \sqcup \ldots \sqcup D_i
\]

\[
(\geq n_1 RC_1) \cap \ldots \cap (\geq m RC_m) \cap (\leq p_1 RD_1) \cap \ldots \cap (\leq p_k RD_k) \preceq (\leq p RD)
\]

(where p is computed by the Maximization algorithm described in section 4)

References


