

Contents

1	Introduction	2
2	Preliminaries	3
3	Prearrangement	6
3.1	The form of the ABox	7
3.2	Blocking	9
4	Algorithm	12
5	Correctness	16

1 Introduction

Traditional data bases where data are stored solely without any connection towards to themselves like many people would imagine are often not enough any more. The reason is that the data are stored without any semantics. However storing data with semantics can provide additional information. For example we have some data about two objects "Anna" and "Beth". In a traditional data base if not explicitly stated, both data are not related to each other. Nevertheless Anna and Beth can have a relation, which also depends on who or what both are. For example both can be human and Anna is a teacher and Beth is a student. Both are in the same class. By adding solely those information in a traditional data base the information that Anna must teaches Beth is not given. One way to apply semantics to data objects is to use *ontologies*. In biological and (bio)medical researches data bases are often based on ontologies [3]. Ontologies (in the computer science field) can be viewed as formal representation of a certain domain of interest. In data base they are collection of relation between the entities in the data base and are formulated as a fragment of first-order logic (FOL). These fragments of FOL are represented as *Description Logic (DL)*, which is a family of knowledge representation system. DL are mainly built of concepts, which correspond to unary relations in FOL and is often represented by a capital letter, and relation between the concepts, which correspond to binary relations in FOL and is often represented by a lowercase letter. For more complex (compound) concepts operators like \sqcap , \sqcup , \sqsubseteq , \exists and \forall , depending on the DL, are used. For example the statement "All Men and Women are Human" is formalize in FOL as $\forall x. Men(x) \vee Women(x) \rightarrow Human(x)$ and in DL as an *axiom* $Men \sqcup Women \sqsubseteq Human$, where *Men*, *Women* and *Human* are concept names. The statement "All Humans, who have children, are parents" can be formalized in FOL as $\forall x \exists y. Human(x) \wedge hasChildren(x, y) \rightarrow Parent(x)$ and in DL as $Human \sqcap \exists hasChildren. \top \sqsubseteq Parent$, where *Human* and *Parents* are concept names and *hasChildren* is a role name. Restriction with the operators \exists and \forall are called *quantified* restrictions. The second statement can also be formalized with a *qualified* restriction: $Human \sqcap \geq 1 hasChildren. \top \sqsubseteq Parent$. Each quantified restriction can be transformed into a qualified restriction.

One big research field in DL is the determination of satisfiability of an *knowledge base*, which is formulated in DL. A knowledge base normally consists of a *TBox*, which contains the axioms (rules), and of an *ABox* which contains assertions of certain elements (objects). This DL allows conjunctions (\sqcap), disjunctions (\sqcup), negation $\neg C$ and qualifying number restriction ($\leq nr C$ and $\geq nr C$), where n is a number, r a role name, and C a concept name. In [4] a *Tableau*-algorithm is presented for checking satisfiability for an ABox in the DL \mathcal{ALCQ} . A *Tableau*-algorithm applies *completion rules* to a given *set*(ABox) to decompose complex concepts and try satisfying violated *statements*(assertions). If the set concludes something unsatisfiable (clash) then the whole set is unsatisfiable. If no more rules are applicable and the set is not unsatisfiable, then it is otherwise. The satisfiability (of concepts) is stated in [4] as PSPACE-hard problem (without TBox, with TBox it is EXPTIME-hard [1]. In [7] a optimized *Tableau*-algorithm is presented which results in a PSPACE-problem. The optimization is that instead of

keeping n successors to satisfy a restriction $\geq nr.C$ like in [4], the algorithm saves the number of existing successors and by comparing the numbers detects possible clashes. This DL is more expressive than \mathcal{ALCQ} because every qualified restriction $\leq nr.C$ and $\geq nr.C$ can be written in \mathcal{ALCSCC} as $\text{succ}(|r.C| \leq 1)$ and $\text{succ}(|r.C| \geq 1)$.

The expressive DL \mathcal{ALCSCC} extends \mathcal{ALCQ} with *set constraint* and *cardinality constraint*, which lays under the logic of $QFBAPA$ (quantifier-free fragment of Boolean Algebra with Presburger Arithmetic). As the name says we do not have quantifier. Instead we use set expression (Boolean Algebra part) and numerical constraint (Presburger Arithmetic) which is combined together with cardinality functions. For example $Human \sqcap \geq 1 \text{ hasChildren} \sqcap \sqsubseteq Parent$ is written in \mathcal{ALCSCC} as $Human \sqcap \text{succ}(|\text{hasChildren}| \geq 1) \sqsubseteq Parent$. In [1] a solution for the satisfiability problem (without TBox) is presented which has the complexity PSpace: For an ABox we guess the value (true or false) of the top-level variables which can already lead to a *false*-result. If not then the constraint is formulated into a $QFBAPA$ formula, for which an algorithm determine by guessing a number N of Venn-region to be non-empty whether the formula is satisfied or not.

In this work we give another solution for the satisfiability problem with a Tableau-algorithm. As in previous work for other DLs we define completion rules which can be applied onto assertions in the ABox to determine whether \perp can be concluded from it which states its unsatisfiability. If we can not apply any rules any more and we can not conclude \perp , then the ABox is satisfiable. The main difficulty is that unlike \mathcal{ALCQ} , where the bond of number of successor is fixed, in \mathcal{ALCSCC} we can compare two cardinalities, which can vary during the algorithm. Hence we need an approach for counting successors and with it calculating the *correct* cardinality, which is necessary to detect satisfied and violated constraint. For this we introduce *induced interpretation* which can determine the cardinalities after each rule application. We want to make sure, that the variation of the cardinalities are in our favour and hence *block* certain assertion from adding into the ABox. To determine which assertion needs to be blocked we apply the satisfiability algorithm from [1].

2 Preliminaries

In this work \mathbf{C} denotes a set of concept names and \mathbf{R} a set of role names, which are disjoint. Before we define the DL \mathcal{ALCSCC} we have to explain first how the language $QFBAPA$ looks like.

Definition 1 ($QFBAPA$). Let T be a set of symbols

- set terms over T are:
 - empty set \emptyset and universal set \mathcal{U}
 - every set symbol in T
 - if s, t are set terms then also $s \cap t$, $s \cup t$ and s^\neg
- set constraints over T are

- $s \subseteq t$ and $s \not\subseteq t$
- $s = t$ and $s \neq t$

where s, t are set terms

- cardinality terms over T are:

- every number $n \in \mathbb{N}$
- $|s|$ if s is a set term
- if k, l are cardinality terms then also $k + l$ and $n \cdot k$, $n \in \mathbb{N}$

- cardinality constraints over T are:

- $k = l$ and $k \neq l$
- $k < l$ and $k \geq l$
- $k \leq l$ and $k > l$
- $n \text{ dvd } k$ and $n \neg \text{dvd } k$

where k, l are cardinality terms and $n \in \mathbb{N}$

Since $s \subseteq t$ can be expressed as the cardinality constraint $|s \cap t^c| \leq 0$ we will not consider any set constraints further in this work. In case we want to express $x : \text{succ}(s = t)$, with s, t being set terms, we write instead $x : \text{succ}(|s \cap t^c| \leq 0) \sqcap \text{succ}(|s^c \cap t| \leq 0)$. Furthermore instead of $l \geq k$ we write $k \leq l$, instead of $k < l$ we write $k + 1 \leq l$ and instead of $k = l$ we write $k \leq l$ and $l \leq k$. Hence for an assertion $x : \text{succ}(c)$ the cardinality constraint c is either of the form $k \leq l$ or $n \text{ dvd } l$.

Definition 2 (\mathcal{ALCSCC}). \mathcal{ALCSCC} concepts are defined inductively:

- all concept names
- $\text{succ}(c)$ if c is a cardinality constraint over \mathcal{ALCSCC} concepts and role names
- if C, D are concepts then:
 - $\neg C$
 - $C \sqcup D$
 - $C \sqcap D$

An ABox S in \mathcal{ALCSCC} is a finite set of assertions of the form $x : C$ and $(x, y) : s$, where C is a \mathcal{ALCSCC} concept, s a set term and x, y variables. The set $\text{Var}(S)$ is the set of variables occurring in S .

Definition 3 (Interpretation). An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}, \pi_{\mathcal{I}})$ over an ABox S in \mathcal{ALCSCC} consists of a non-empty set $\Delta^{\mathcal{I}}$, an assignment $\pi_{\mathcal{I}}$ and a mapping $\cdot^{\mathcal{I}}$ which maps:

- \emptyset to $\emptyset^{\mathcal{I}}$

- \mathcal{U} to $\mathcal{U}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
- each variable $x \in \text{Var}(S)$ to $x^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
- every concept names $A \in \mathbf{C}$ to $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
- every role name $r \in \mathbf{R}$ to $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, such that every element in $\Delta^{\mathcal{I}}$ has a finite number of successors.

The set $r^{\mathcal{I}}(x)$ contains all elements y such that $(x, y) \in r^{\mathcal{I}}$ e.g. it contains all r -successors of x .

For compound concepts the mapping $\cdot^{\mathcal{I}}$ is extended inductively as follows

- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$ and $\perp^{\mathcal{I}} = \emptyset^{\mathcal{I}}$
- $(C \sqcap D)^{\mathcal{I}} := C^{\mathcal{I}} \cap D^{\mathcal{I}}$, $(C \sqcup D)^{\mathcal{I}} := C^{\mathcal{I}} \cup D^{\mathcal{I}}$
- $(\neg C)^{\mathcal{I}} := \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
- $(s \sqcap t)^{\mathcal{I}} := s^{\mathcal{I}} \cap t^{\mathcal{I}}$, $(s \sqcup t)^{\mathcal{I}} := s^{\mathcal{I}} \cup t^{\mathcal{I}}$
- $(s^{\neg})^{\mathcal{I}} := \mathcal{U}^{\mathcal{I}} \setminus s^{\mathcal{I}}$
- $|s|^{\mathcal{I}} := |s^{\mathcal{I}}|$
- $(k + l)^{\mathcal{I}} := (k^{\mathcal{I}} + l^{\mathcal{I}})$, $(n \cdot k)^{\mathcal{I}} := n \cdot k^{\mathcal{I}}$
- $\text{succ}(c)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \text{the mapping } \cdot^{\mathcal{I}_x} \text{ satisfies } c\}$

The mapping $\cdot^{\mathcal{I}_x}$ maps \emptyset to $\emptyset^{\mathcal{I}_x}$, \mathcal{U} to $\mathcal{U}^{\mathcal{I}_x} := \{\bigcup_{r \in \mathbf{R}} r^{\mathcal{I}_x}(x)\}$, every concept C occurring in c to $C^{\mathcal{I}_x} := C^{\mathcal{I}} \cap \mathcal{U}^{\mathcal{I}_x}$ and every role name r occurring in c to $r^{\mathcal{I}_x} := r^{\mathcal{I}}(x)$.

The mappings satisfies for the cardinality terms k, l

- $k \leq l$ iff $k^{\mathcal{I}} \leq l^{\mathcal{I}}$
- $n \text{ dvd } l$ iff $\exists m \in \mathbb{N} : n \cdot m = l^{\mathcal{I}}$

The assignment $\pi_{\mathcal{I}} : \text{Var}(S) \rightarrow \Delta^{\mathcal{I}}$ satisfies

- $x : C$ iff $\pi_{\mathcal{I}}(x) \in C^{\mathcal{I}}$
- $(x, y) : s$ iff $(\pi_{\mathcal{I}}(x), \pi_{\mathcal{I}}(y)) \in s^{\mathcal{I}}$

$\pi_{\mathcal{I}}$ satisfies an ABox S if $\pi_{\mathcal{I}}$ satisfies every assertion in S . If $\pi_{\mathcal{I}}$ satisfies S then \mathcal{I} is a model of S .

We say a set term t occurs semantically in a cardinality term $k = n_0 + n_1 \cdot |s_1| + \dots + n_j \cdot |s_j|$ if for a $i \in \{1, \dots, j\}$ we have $s_i = t \cup t_{rest}$, where t_{rest} is a set term. Note that a set term t does not occurs semantically in the cardinality term $|t^{\neg}|$.

3 Prearrangement

A Tableau-algorithm consist of completion rules to decide satisfiability of a set of assertions. The rules are applied exhaustively on the set until none is applicable any more. One major characteristic of this algorithm is that it does not matter in which order the rules are applied. Another characteristic is that it works non-deterministically: In case we have disjunctions we can choose between the concepts in this disjunctions. If a choice ends in a *clash* then we track back to the point where we had to chose and take the other choice instead. If all choices ends in a clash then the ABox is unsatisfiable, otherwise it is satisfiable.

To help the algorithm we want to avoid nested negation e.g. $\neg(\neg(\neg(A \sqcup B)))$. Hence we consider all concepts in *negated normal form (NNF)*.

Definition 4 (Negation Normal Form). A \mathcal{ALCSCC} concept is in *negation normal form (NNF)* if the negation sign \neg appears only in front of a concept name or above a role name. Let C be a arbitrary \mathcal{ALCSCC} concept. With $NNF(C)$ we denote the concept which is obtained by applying the rules below on C until none is applicable any more.

- $\neg \top \rightarrow \perp$
- $\neg \perp \rightarrow \top$
- $\neg \neg C \rightarrow C$
- $\neg(C \sqcap D) \rightarrow \neg C \sqcup \neg D$
- $\neg(C \sqcup D) \rightarrow \neg C \sqcap \neg D$
- $C^\neg \rightarrow \neg C$
- $\neg succ(c) \rightarrow succ(\neg c)$
- $\neg(k \leq l) \rightarrow l \leq k$
- $\neg(n \text{ dvd } k) \rightarrow n \neg \text{dvd } k$
- $\neg(n \neg \text{dvd } k) \rightarrow n \text{ dvd } k$
- $(s \sqcap t)^\neg \rightarrow s^\neg \sqcup t^\neg$
- $(s \sqcup t)^\neg \rightarrow s^\neg \sqcap t^\neg$
- $(s^\neg)^\neg \rightarrow s$

The rule $C^\neg \rightarrow \neg C$ is necessary because C^\neg can be a result of s^\neg , where s is a set term. It can be transformed into $\neg C$: For every interpretation \mathcal{I} of S we have $(C^\neg)^\mathcal{I} = \mathcal{U} \setminus C^\mathcal{I}$ and $(\neg C)^\mathcal{I} = \Delta^\mathcal{I} \setminus C^\mathcal{I}$. Since $\mathcal{U} \subseteq \Delta$ we can conclude that every element in $(C^\neg)^\mathcal{I}$ is also in $(\neg C)^\mathcal{I}$.

The first five rules on the left hand side can be applied in linear time [4],[6]. The first four rules on the right hand side, $C^\neg \rightarrow \neg C$ and $\neg succ(c) \rightarrow succ(\neg c)$ can also be applied in linear time since we only shift the negation sign. the rule $(s^\neg)^\neg \rightarrow s$ works similarly to $\neg \neg C \rightarrow C$ and the rules $(s \sqcap t)^\neg \rightarrow s^\neg \sqcup t^\neg$ and $(s \sqcup t)^\neg \rightarrow s^\neg \sqcap t^\neg$ works the similarly to $\neg(C \sqcap D) \rightarrow \neg C \sqcup \neg D$ and $\neg(C \sqcup D) \rightarrow \neg C \sqcap \neg D$ and can also be applied in linear time. Next we introduce *induced interpretation* with which we can count successors of variables after any rule application.

Definition 5 (Induced Interpretation). An interpretation $\mathcal{I}(S)$ can be induced from an ABox S by the following steps:

- for each variable $x \in Var(S)$ we introduce $x^{\mathcal{I}(S)}$ and add it to $\Delta^{\mathcal{I}(S)}$

- for each $x : C$ such that C is a concept name we add $x^{\mathcal{I}(S)}$ to $C^{\mathcal{I}(S)}$
- for each $(x, y) : r$ such that r is a role name we add $(x^{\mathcal{I}(S)}, y^{\mathcal{I}(S)})$ to $r^{\mathcal{I}(S)}$

Since we can now denote the number of successor of a variable x we can determine which assertion of the form $x : succ(c)$ are violated.

Definition 6 (Violated assertion). Let S be a set of assertion, x be a variable, k be a cardinality term and $n \in \mathbb{N}$. An assertion is *violated* if

- $x : succ(k \leq n)$ and $k^{\mathcal{I}(S)_x} \not\leq n$
- $x : succ(k \leq l)$ and $k^{\mathcal{I}(S)_x} \not\leq l^{\mathcal{I}(S)_x}$
- $x : succ(n \text{ } dvd \text{ } k)$ and $mod(k^{\mathcal{I}(S)_x}, n) \neq 0$

where $n \in \mathbb{N}$.

Like already mentioned an ABox is unsatisfiable if all choices ends in a clash. A clash in a ABox S implies that \perp can be derived from S .

Definition 7 (Clash). An ABox S contains a *clash* if

- $\{x : \perp\} \subseteq S$ or
- $\{x : A, x : \neg A\} \subseteq S$ or
- $\{(x, y) : s, (x, y) : s^{\neg}\} \subseteq S$ or
- $\{x : succ(c)\} \subseteq S$ violated and no more rules are applicable

3.1 The form of the ABox

Like already mention a ABox S in \mathcal{ALCSCC} contains assertions of the form $x : C$ and $(x, y) : C$ where C is a \mathcal{ALCSCC} concept and $x, y \in Var(S)$ are variables. Like already discussed we want constraints to be in NNF and for an assertion $x : succ(c)$ that c is either $k \leq l$ or $n \text{ } dvd \text{ } l$. Our desired form for the ABox is gained by the following algorithm.

Algorithm 1 Transforming ABox

```
ABox  $S$ 
for each assertion  $x : succ(c) \in S$  do
  if  $c$  is  $s_1 \subseteq s_2$  then
     $c := |s_1 \cap s_2^\neg| \leq 0$ ; return;
  end if
  if  $c$  is  $s_1 \supseteq s_2$  and a set term constraint then
     $c := |s_1^\neg \cap s_2| \leq 0$ ; return;
  end if
  if  $c$  is  $s_1 = s_2$  then
    remove  $x : succ(c)$ ;
    add  $x : succ(|s_1 \cap s_2^\neg| \leq 0)$  and  $x : succ(|s_1^\neg \cap s_2| \leq 0)$ ; return;
  end if
  if  $c$  is  $l > k$  then
     $c := k + 1 \leq l$ ; return;
  end if
  if  $c$  is  $k < l$  then
     $c := k + 1 \leq l$ ; return;
  end if
  if  $c$  is  $l \geq k$  then
     $c := k \leq l$ ; return;
  end if
  if  $c$  is  $l = k$  and a cardinality constraint then
    remove  $x : succ(c)$ ;
    add  $x : succ(l \leq k)$  and  $x : succ(k \leq l)$ ; return;
  end if
end for

for each assertion  $x : C$  or  $(x, y) : s$  do
   $C := NNF(C)$  or  $s := NNF(s)$ ;
end for
```

The first part of the transformation runs in worst case in polynomial time: If any of the first four conditions holds, we only replace c which runs in constant time. The complexity of a removing function depends on the implementation: If we have direct access to the elements in the ABox the remove function has a constant (runtime) complexity. If we have some kind of sorting then the (runtime) complexity is logarithm. In worst case if we the ABox is stored as a simple list, then the (runtime) complexity is linear. Hence in worse cast the first part runs in polynomial time. The second part always runs in polynomial time because each transformation into NNF runs in linear time. Hence our desired form of the ABox can be obtained in a polynomial time.

The idea of a Tableau-algorithm is rather simple: We apply rules in a arbitrary order whenever they are applicable. However often some caution has to be made. In general we

do not want to apply rules which *does not help* to satisfies assertion and hence results in a not necessary long runtime. For example we do not want to repeat any rule applications order, which can be applied infinity times. In [4] the considered DL is \mathcal{ALCQ} which allows qualified number restrictions. The algorithm uses a rule for replacing variables but to prevent endless loops of rule application, which can occur due to the replication, a safeness condition is added. In [5] and [6] the considered DL allows inverse roles which is handled with blocking techniques. A similar blocking technique is used in [2] also to prevent non-termination due to considered cyclic TBoxes. The DL \mathcal{ALCSCC} is a very expressive concept language and hence there are some difficulty to handle for the Tableau-algorithm.

3.2 Blocking

In this section we explain the blocking technique for our algorithm.

For blocking we introduce a blocking set $b(S, x)$ which contains set terms which we want to be blocked in S .

Definition 8 (Blocking Set $b(S, x)$ I). The blocking set $b(S, x)$ denotes all set terms such that for a variable y we can not add $(x, y) : u$ to S if $u = n_1 \cdot |s_1| + \dots + n_j \cdot |s_j|$ and $s_i \cup u \in b(S, x)$, $1 \leq i \leq j$. We say u is blocked for x in S .

Which set term needs to be blocked is explained in this section. We explain based on examples how we select set terms.

First we look into *merging*. For that we consider this example:

Example 1.

$$S = \{x : succ(1 \geq |A \cup B|), x : succ(1 \leq |A|), x : succ(1 \leq |B|)\}$$

Consider the Tableau-algorithm choose to fix the second and third assertion first which leads to $1 \not\geq 2 = |A \cup B|$. Of course we can also some rules have a higher priority but then the whole basic idea of the Tableau-algorithm is gone. Therefore, similar to [4] and [6], where a variable can be replaced by another variable, we want to merge the two variables like in [5].

Definition 9 (Merge). *Merging* y_1 and y_2 results in one variable y : replace all occurrence of y_1 and y_2 with y .

For Example 1 we would merge the two introduced variables to one which means it is of the concept $A \cap B$ and hence the first assertion becomes satisfied again.

By merging variables assertion may become violated. To ensure the termination of the algorithm we intuitively want to avoid violating any assertions especially when they are satisfied. However there are cases where a violation is unavoidable. We look at the following example:

Example 2.

$$S = \{x : succ(|t| + |A \cup B| \geq 4), x : succ(|A \cup B| \leq 1) \\ y_1 : A, y_2 : B, (x, y_1) : t, (x, y_2) : t\}$$

We see that the assertion $x : \text{succ}(|A \cup B| \leq 1)$ is violated and a solution is to merge y_1 and y_2 . This leads to $x : \text{succ}(|t| + |A \cup B| \geq 4)$ being violated. We can fix it by adding new successors. We can easily detect that adding a t -successor leads to a satisfied ABox. On the other hand a successor in $A \cup B$ leads to the initial state where $x : \text{succ}(|A \cup B| \leq 1)$ is violated. Hence the algorithm can run into a endless loop of adding and merging variables. Therefore we want block any introduction of assertion, which increases $|A \cup B|$.

Definition 10 (Blocking Set $b(S, x)$ II - Merging). Let y_1 and y_2 be successors of a variable x . If y_1 and y_2 are merged into y because of an assertion $x : \text{succ}(k \leq l)$ then $b(S, x) := b(S, x) \cup \{s_i | k = n_0 + n_1 \cdot |s_1| + \dots + n_j \cdot |n_j|, \forall i : 1 \leq i \leq j\}$.

In our example after the merging the set term $A \cup B$ is added to $b(S, x)$ and hence this set term is blocked by x , which means that to satisfy $x : \text{succ}(|t| + |A \cup B| \geq 4)$ we can only add a t -successor. This blocking set is then used to determine whether a variable can be introduced or not.

The last examples always compared a cardinality term with a fix number. However our DL is also capable of comparing two cardinality terms as in $k \leq l$, where k and l can vary during the algorithm.

Intuitively for a assertion $x : \text{succ}(k \leq l)$ we want to add variables, such that l increases. However if both k and l can increase by a new variable, we have to avoid cases where k increases faster then l . Another more crucial problem is when by adding successors not only $k^{\mathcal{I}(S)_x}$ and $l^{\mathcal{I}(S)_x}$ can increases, but also for some other assertion $x : \text{succ}(i \leq j)$ the cardinalities $i^{\mathcal{I}(S)_x}$ or $j^{\mathcal{I}(S)_x}$. For that we proceed as follows: First we gather all assertions, which are *depended* to each other, into a set \mathcal{A} , formulate a *QFBAPA* formula of the constraints and then let a *QFBAPA* solver determine whether the formula and hence \mathcal{A} is satisfiable. If the solver states that the formula is unsatisfiable then we want to block any set terms which occurs semantically in any cardinality term in \mathcal{A} .

Definition 11 (The blocking set $b(S, x)$ III - Introducing variables). Let \mathcal{A} be a set of assertion of the form $x : \text{succ}(k \leq l)$. If \mathcal{A} is unsatisfiable then

$$\begin{aligned} b(S, x) &= b(S, x) \cup \\ &\{s_i | k = n_0 + n_1 \cdot |s_1| + \dots + n_j \cdot |n_j|, \forall i : 1 \leq i \leq j, x : \text{succ}(k \leq l) \in \mathcal{A}\} \cup \\ &\{s_i | l = n_0 + n_1 \cdot |s_1| + \dots + n_j \cdot |n_j|, \forall i : 1 \leq i \leq j, x : \text{succ}(k \leq l) \in \mathcal{A}\} \end{aligned}$$

So before the Tableau-algorithm start, we want to divide all assertions of the form $x : \text{succ}(k \leq l)$ into sets:

Algorithm 2 Dividing assertions into depending sets

```
ABox  $S$ 
for each assertion  $x : succ(k \leq l) \in S$  do
  create a new set  $\mathcal{A} = \{x : succ(k \leq l)\}$ 
  remove  $x : succ(k \leq l)$  from  $S$ 
  for each assertion  $x : succ(k \leq l) \in \mathcal{A}$  do
     $l = n_0 + n_1 \cdot |l_1| + \dots + n_j \cdot |l_j|$ 
    for each assertion  $x : succ(p \leq q) \in \mathcal{S}$  do
      if  $\exists i \in \{1, \dots, j\} : l_i$  occurs semantically in  $p$  or  $q$  then
        remove  $x : succ(p \leq q)$  from  $S$ 
         $\mathcal{A} = \mathcal{A} \cup \{x : succ(p \leq q)\}$ 
      end if
    end for
  end for
end for
return all  $\mathcal{A}$ 
```

We then convert each assertion into *QFBAPA* formulas by

- drop $x : succ$
- replacing each role name $r \in \mathbf{R}$ in the assertion by a variable X_r and each concept name $C \in \mathbf{C}$ by X_C
- connect formulas with \wedge

Note in case of r^\neg or $\neg C$ we have in the formula X_r^\neg and X_C^\neg .

The *QFBAPA* formula for Example 1 would be $1 \geq |X_A \cup X_B| \wedge 1 \leq |X_A| \wedge 1 \leq |X_B|$.

We then let a *QFBAPA* solver determine whether the formula has a solution or not.

At the end we grab again the idea that for an assertion $x : succ(k \leq l)$ we want to increase l and want to avoid that k increases faster than l . Hence we introduce the definition of *safe*.

Definition 12 (Safe). Let $x : succ(k \leq l)$ be an assertion in S . Let $u = t_1 \cap \dots \cap t_j$. If $n_k(u) < n_l(u)$ and u is not blocked by x then u is called *safe regarding* $k \leq l$. The number $n_k(u)$ (and $n_l(u)$ respectively) is computed as followed:

Algorithm 3 Compute $n_k(u)$

```
 $n_k(u) := 0$ 
 $k = n_0 + n_1 \cdot |s_1| + \dots + n_j \cdot |s_j|$ 
 $u = t_1 \cap \dots \cap t_o$ 
for each  $1 \leq i \leq j : n_i \cdot |s_i|, s_i = s'_1 \cup \dots \cup s'_p, p \in \mathbb{N}$  do
  if  $\exists q, 1 \leq q \leq p : u = s'_q \cap t'$  then
     $n_k(u) := n_k(u) + n_i$ 
  end if
end for
return  $n_k(u)$ 
```

This says that it is only safe to add a variable if l increases faster than k . As example we look at

Example 3.

$$S = \{x : \text{succ}(|r \cup s| < |r| + |s|)\}$$

The set terms r and s are not blocked but still not safe because $n_{|r \cup s|}(r) = n_{|r|+|s|}(r) = 1$ and $n_{|r \cup s|}(s) = n_{|r|+|s|}(s) = 1$. However the set term $r \cap t$ is safe because $r \cap t \notin b(S, x) = \emptyset$ and $1 = n_{|r \cup s|}(r \cap s) < n_{|r|+|s|}(r \cap s) = 2$.

4 Algorithm

For this algorithm it is important to consider the *signs* of concept names and role names.

Definition 13 (Positive and Negative Sign). Let $(x, y) : s$ be an arbitrary assertion with $x, y \in \text{Var}(S)$ and s being a set term in NNF . A concept name C has a *positive sign* in s if no negation sign appears immediately in front of C . It has a *negative sign* otherwise. A role name r has a *positive sign* if no negation sign appears above it. It has a *negative sign* otherwise.

All concept names in **C** and role names in **R** have a positive sign.

We also lose a bit the property of a Tableau-algorithm that rules can be applied in any order: In case we add $(x, y) : s$ to our ABox S and s is a (always finite) chain of disjunction and conjunction we want to add all assertions of y first before applying any other rules. This way y has all assertion we want to assign to it and hence avoid adding unnecessary variables which do not help to satisfy S or, which is worse, can lead into a violation of other assertions. For example we have

Example 4.

$$S = \{x : \text{succ}(1 \leq |r \cap (A \cup B)|), x : \text{succ}(1 \leq |A|)\}$$

In case we want to satisfy the first assertion we introduced y and add $(x, y) : r \cap (A \cup B)$ to S . If we do not consider the new assertion first we take away the possibility of y being in A and hence the second assertion is already satisfied. Therefore we would introduce a new variable z again to satisfy the second assertion, which is a successor of x and in A . If it happens that $y : A$ then the introduction of z was unnecessary. If instead of 1 there stands a higher number, we can end up with a good amount of unnecessary introduced variables. In a worse case where S has additionally the assertion $x : succ(1 \geq |A|)$ and for y we conclude that it is also a successor in A , the assertion even becomes violated.

Now we can start with the algorithm:

First we do the prearrangement: transforming the ABox into the desired form and determining whether some set of assertions \mathcal{A} is satisfiable or not and if needed fill the blocking sets for each variable in $Var(S)$.

Finally we can present the Tableau-algorithm to determine the satisfiability of an ABox in *ALCSCC*.

Definition 14 (Tableau). Let S be a set of assertions in simplified *NNF*.

1. \sqcap -rule: S contains $x : C_1 \sqcap C_2$ but not both $x : C_1$ and $x : C_2$
 $\rightarrow S := S \cup \{x : C_1, x : C_2\}$
2. \sqcup -rule: S contains $x : C_1 \sqcup C_2$ but neither $x : C_1$ nor $x : C_2$
 $\rightarrow S := S \cup \{x : C_1\}$ or $S := S \cup \{x : C_2\}$
3. *choose*-rule: S contains
 - $x : succ(k \leq l)$
 - $(x, y) : k'$, $k = n \cdot |k' \cup u_1| + m \cdot |k' \cap u_2| + s$, $n, m \in \mathbb{N}_0$, u_1, u_2 are set terms and s a cardinality term
 - but neither $(x, y) : k$ nor $(x, y) : k^\neg$ \rightarrow either $S := S \cup \{(x, y) : k\}$ or $S := S \cup \{(x, y) : k^\neg\}$. Then jump to rule 9
4. *choose-a-role*-rule: S contains $(x, y) : s$ but for any $r \in \mathbf{R}$: $(x, y) : r \notin S$
 \rightarrow choose $r \in \mathbf{R}$, such that $(x, y) : r^\neg \notin S$. $S := S \cup \{(x, y) : r\}$. Then jump to rule 9
5. *divide*-rule: S contains $x : succ(n \, dvd \, l)$, $l = n_1 \cdot |s_1| + \dots + n_i \cdot |s_i| + \dots + n_j \cdot |s_j|$, which is violated
 \rightarrow introduce a new variable y , choose $s = s_1 \cap \dots \cap s_i$, $1 \leq i \leq j$ and $S := S \cup \{(x, y) : s\}$. Then jump to rule 9
6. \leq -rule: S contains
 - $x : succ(k \leq l)$, which is violated
 - there is a set term $s := |s_1 \cap \dots \cap s_i|$, $l = n_1 \cdot |s_1| + \dots + n_i \cdot |s_i| + \dots + n_j \cdot |s_j|$, which is safe regarding $k \leq l$ $\rightarrow S := S \cup \{(x, y) : s\}$. Then jump to rule 9

7. *merge-rule*: S contains

- $x : succ(k \leq l)$, which is violated
- $(x, y_1) : s_1$ and $(x, y_2) : s_2$, such that $y_1 \neq y_2$ and $k = n \cdot |s_1 \cap s_2| + u$ or $k = n_1 \cdot |s_1 \cup t_1| + n_2 \cdot |s_2 \cup t_2| + u$, where u is a cardinality term, t_1, t_2 are set terms, $n, n_1, n_2 \in \mathbb{N}$

→ merge y_1 and y_2

8. ≤ 0 -rule: S contains

- $x : succ(|s_1 \cap \dots \cap s_j| \leq 0)$
- $(x, y) : s_1, \dots, (x, y) : s_i, 1 \leq i < j$
- but not $y : s_{i+1}, \dots, y : s_j$

→ choose $n \in \{i+1, \dots, j\}$, extend $S := S \cup \{(x, y) : s_n^-\}$ and then jump to rule 9

9. *set.term-rule* (Repeat until inapplicable): In S is $(x, y) : s$ and

- $s = s_1 \cap s_2$ but $\{(x, y) : s_1, (x, y) : s_2\} \not\subseteq S$
→ $S := S \cup \{(x, y) : s_1, (x, y) : s_2\}$
- $s = s_1 \cup s_2$ and neither $\{(x, y) : s_1\} \subseteq S$ nor $S \setminus \{(x, y) : s_2\} \subseteq S$
→ either $S := S \cup \{(x, y) : s_1\}$ or $S := S \cup \{(x, y) : s_2\}$
- $s = C$ and $y : C \notin S$, where C is an $\mathcal{ALCCSCC}$ concepts
→ $S := S \cup \{y : C\}$

We now explain the rules of the Tableau-algorithm and their intention.

The first rule decompose the conjunction and the second rule adds non-deterministically the right assertion.

The *choose*-rule is important because we need to know of every successor what kind of role successors and in which concepts they are. For an assertion $x : succ(k \leq l)$ it is important that $k^{\mathcal{I}(S)_x}$ and $l^{\mathcal{I}(S)_x}$ counts the successors correctly. In the following case the successor y is not counted in $l^{\mathcal{I}(S)_x} := (|r \cap s|^{\mathcal{I}(S)_x})$ while $x : succ(k \leq l)$ is violated.

Example 5.

$$S = \{x : succ(1 \leq |r \cap s|), (x, y) : r\}$$

There might be an model \mathcal{I}' where y is also a s -successor of x and hence $l^{\mathcal{I}(S)_x} < l^{\mathcal{I}'(S)_x}$, which means that the assertion is in \mathcal{I}' satisfied, but in $\mathcal{I}(S)$ not. However the Tableau-algorithm should be able to construct every model of S if S is consistent. Therefore this rule adds non-deterministically either $(x, y) : s$ or $(x, y) : s^-$ which are the only two possibilities. This way we are also able to construct \mathcal{I}' .

The *choose-a-role*-rule is necessary because in a ABox we may have assertions, which state that x must have a successor but there are no assertions, which states what kind of successor it is. Hence we have to pick the *right* role name. As example we have

Example 6.

$$\begin{aligned}\mathbf{R} &= \{r, s\} \\ S &= \{x : succ(|r^\neg| \leq 1)\}\end{aligned}$$

It states that x have at least one successor which is not a r -successor. Since \mathbf{R} only contains r and s we know that the successors must be an s -successor. First we apply rule 6 to actually add a successor. Therefore y is introduced and $(x, y) : r^\neg$ is added to S . Now no more rules are applicable except for the *choose-a-role*-rule. With that rule we can not pick r because r^\neg occurs in the assertion. Therefore we have to pick s . Another more simple but not so significant example is

Example 7.

$$\begin{aligned}\mathbf{R} &= \{r, s\} \\ S &= \{x : succ(|A| \geq 1)\}\end{aligned}$$

We know that x must have a successor in A but we still need to assign a role. In this case we can choose between r and s .

The *divide*-rule is straightforward: We choose one set term $s = s_1 \cap \dots \cap s_i$ such that $l = n_1 \cdot |s_1| + \dots + n_i \cdot |s_i| + \dots + n_j \cdot |s_j|$ and introduce a new variable y and add $(x, y) : s$ to S . For any $x : succ(n \text{ d}vd l)$ we know that the chain of this rule application is finite because in worst case we have to introduce n new variables with the same set term.

The main idea of the \leq -rule is simple: We want to increase l until the constraint is satisfied. By the condition it has to be safe we know that l increases faster than k and that we do not end in a endless loop of adding and merging variables.

For the *merge*-rule we restrict the merging to the left hand side of cardinality constraints $k \leq l$: It can be reasonable for the right hand side if by merging l increases, for example $x : succ(1 \leq |r \cap t|)$ with $(x, y_1) : r$ and $(x, y_2) : t$. However the easiest solution is just to add an $r \cap t$ -successor. In case it is important to also restrict the number of $r \cap t$ -successor e.g. $x : succ(|r \cap t| < 2)$ we can define with the *choose*-rule whether y_1 and y_2 are also $r \cap t$ -successor. And then with the assertion $x : succ(|r \cap t| < 2)$ we can merge y_1 and y_2 .

The ≤ 0 -rule deal with an assertion with a set constraint $s_1 \subseteq s_2$, which is written here as cardinality constraint $|s_1 \cap s_2^\neg| \leq 0$. Those cardinality constraint can not be dealt with the other rules. In case the left side has at least three set term e.g. $|s_1 \cap s_2 \cap s_3|$ we have can have multiple possible solutions e.g. $(x, y) : s_1 \cap s_2 \cap s_3^\neg$, $(x, y) : s_1 \cap s_2^\neg \cap s_3$ and $(x, y) : s_1 \cap s_2^\neg \cap s_3^\neg$. Hence we let the algorithm choose and backtrack if needed.

The *set.term*-rules are applied immediately after a new assertions $(x, y) : s$ is added to S . The reason for that is, that we want to add all needed assertions for y and hence update all $k^{\mathcal{I}(S)_x}$ correctly. We know that the number of this application is finite because an ABox is finite and hence the number of concept names and role names occurring in this ABox is also finite. Since the constraints are in NNF set terms can never be infinite and hence this rule applies only a finite times.

5 Correctness

For the correctness proof of the Tableau-algorithm we have to show that

- For every input the Tableau-algorithm terminates
- If no more rules are applicable on a clash-free ABox S then S is satisfiable
- If S is satisfiable then the Tableau-algorithm terminates without a clash

First we prove that the tableau algorithm terminates.

Proposition 1. Let C be a concept in simplified $CNNF$. Then there is no infinite chain of applications of any tableau rules issuing from $\{x : C\}$.

Definition 15 (Derived Set). A *derived set* is an ABox S' where rule 9 is not applicable.

In order words a derived set is an ABox on which we applied a rule completely e.g. every time we add a new assertion $(x, y) : s$ we add all assertion concluded by it to S (rule 9) first.

To prove this we map any derived set S to an element $\Psi(S)$ from a set Q . We then show that the elements in Q can be ordered by a well-founded relation \prec . A well-founded relation says that there is no infinite decreasing chain. If we can show that by obtaining a derived set S' from another set S we have $\Psi(S') \prec \Psi(S)$ then the algorithm terminates. The elements in Q are finite multisets of septuples and the elements of the septuples are either integers or mutlisets of integers. For two septuples $q = (q_1, \dots, q_7)$ and $q' = (q'_1, \dots, q'_7)$ it holds $q \prec q'$ if for the first i , $1 \leq i \leq 7$, for which q_i and q'_i differs it holds that $q_i \prec q'_i$ (also called lexicographical ordering). For two mutlisets of integers q_i and q'_i it holds $q'_i \prec q_i$ if q'_i can be obtained from q_i by replacing an integer c in q'_i by a finite number of integers which are all smaller than c . The relation \prec for those multisets is well-founded because we work with integers. That means from a multiset $\{0, \dots, 0\}$ the smallest multiset which can be obtain is $\{\}$ by removing all 0s.

For a concept C its size $size(C)$ is inductively defined as

- 0, if C is \perp
- 1, if C is a concept name of \mathbf{C}
- $size(\neg C) = 1 + size(C)$
- $size(succ(c)) = 1 + \sum_{C \text{ in } c} size(C)$
- $size(C \sqcap D) = size(C \sqcup D) = size(C) + size(D)$

The asymmetrical difference of two numbers n, m is denoted by

$$n \trianglelefteq m \begin{cases} n - m & \text{if } n > m \\ 0 & \text{if } n \leq m \end{cases}$$

The septuples in Q are defined as follows

Definition 16. Let S be an ABox. The multiset $\Psi(S)$ consist of septuples $\psi_S(x)$ for each variable x . The component of the septuples are structured as follows

- the first component is a non-negative integer $\max\{size(C) \mid x : C \in S\}$
- the second component is a multiset of integers containing for each $x : C \sqcap D$, on which the \sqcap -rule is applicable, the non-negative integer $size(C \sqcap D)$ (respectively for $C \sqcup D$)
- the third component is the number of x 's successors on which rule 4 is applicable
- the fourth component is a multiset of integers containing for each $x : succ(k \leq n) \in S$ the number of all successors y of x on which rule 3 is applicable
- the fifth component is a multiset which denotes for every $x : succ(k \leq l)$ and $x : succ(n \text{ dvd } l)$ the integer $k^{\mathcal{I}(S)_x} \trianglelefteq l^{\mathcal{I}(S)_x}$ and $n \trianglelefteq l^{\mathcal{I}(S)_x}$ respectively
- the sixth component denotes the number of all successors of x .
- the seventh component is a multiset which denotes for each $x : succ(k \leq 0)$ the number of successors on which the rule 6 is applicable

Lemma 1. The following properties hold

1. For any concept C we have $size(C) \geq size(NNF(\neg C))$
2. Any variable y in a derived set S has at most one predecessor x in S
3. If $(x, y) : r \in S$ for a $r \in \mathbf{R}$ (and y is a introduced variable) then

$$\max\{size(C) \mid x : C \in S\} > \max\{size(D) \mid y : D \in S\}$$

Proof.

1. By induction over the number of applications to compute the negation normal form we have $size(C) = size(NNF(\neg C))$. Because $\neg succ(0 \leq k)$ can be replace by \perp which is *smaller*, we have $size(C) \geq size(NNF(\neg C))$. This can be done because $\neg succ(0 \leq k) = succ(k < 0)$ which is impossible to satisfy and therefore $\neg succ(0 \leq k) = \perp$.
2. If y is a newly introduced variable, then it can only be introduced by exactly one variable x which is y 's only predecessor. If two variables are merged together by rule 7 then both variables must have the same predecessor x by the condition of that rule.
3. By the second fact we know that x is the only predecessor of y . When y is introduced by applying 6 on a assertion $x : succ(k \leq l)$ then we have $y : C$ for every concept C occurring in l (for $\neg C$ we have $y : \neg C$). We know that $size(succ(k \leq l))$ is greater then $size(C)$ therefore Lemma 1.3 holds. A new assertion $y : D$ can occur

either because rule 1 or 2 are applicable on $y : C$ with $C = D \sqcap D'$ or $C = D \sqcup D'$, which neither raise $\max\{size(D) \mid y : D \in S\}$, or because rule 3 is applicable but that also does not raise $\max\{size(D) \mid y : D \in S\}$: If rule 3 is applicable on $x : succ(k \leq l)$ then for every added assertion $y : D$ the concept D must occur in k and therefore $size(succ(k \leq l)) > size(D)$. If y gets merged together with another variable z , then y and z must have the same predecessor which means that all concept sizes regarding z are also smaller than $\max\{size(C) \mid x : C \in S\}$.

□

From the next Lemma we can conclude that the Tableau-algorithm terminates.

Lemma 2. If S' is a derived set obtained from the derived set S , then $\Psi(S') \prec \Psi(S)$

Proof. The following proof is sectioned by the definition of obtaining a derived set.

1. S' is obtained by the application of rule 1 on $x : C \sqcap D$:

The first component remains the same because $size(C) < size(C \sqcap D)$ and $size(D) < size(C \sqcap D)$. The second component decreases because rule 1 can not be applied on $x : C \sqcap D$ any more meaning that the corresponding entry in the multiset is removed. If C (or D) happens to be a disjunction ($C' \sqcup D'$) or a conjunction ($C' \sqcap D'$) then the second component also becomes smaller because $size(C')$ and $size(D')$ are always smaller than the disjunction or conjunction of them and therefore also smaller than $size(C \sqcap D)$. Hence the entry for $size(C \sqcap D)$ can be replaced by the smaller $size(C' \sqcup D')$ or $size(C' \sqcap D')$.

Consider now a tuple $\psi_S(y)$ such that $x \neq y$. $\psi_S(y)$ can only be affected if x is a successor of y . The first and second component of $\psi_S(y)$ remain unaffected because both are independent from x . The third and fourth component of can never increase because we do not add any new successors. If at all the components can decrease because it may happen that by adding assertions regarding x rule 3 (or rule 4) can not be applied any more on some assertion $y : succ(k \leq l)$. The same goes for the fifth component: If there is an assertion $y : succ(k \leq l)$ and by adding $x : C$ and $x : D$ to S neither $k^{\mathcal{I}(S')_y}$ nor $l^{\mathcal{I}(S)_y}$ change because by the definition of $\mathcal{I}(S)$ we have $x \in (C \sqcap D)^{\mathcal{I}(S)_y} = C^{\mathcal{I}(S)_y} \sqcap D^{\mathcal{I}(S)_y} \subseteq C^{\mathcal{I}(S)_y}$ and $\subseteq D^{\mathcal{I}(S)_y}$. This means that x is not added to any further extension in $\mathcal{I}(S')$ and hence $k^{\mathcal{I}(S')_y}$ nor $l^{\mathcal{I}(S)_y}$ remain the same. The sixth component also do not change: From the beginning we have $x : C \sqcap D$, which means that if rule 6 is not applicable then it stays inapplicable after $x : C$ or $y : D$ is added. We can conclude that the tuple $\psi_S(y)$ remains in total unchanged or decreases.

This means that we can obtain $\Psi(S')$ from $\Psi(S)$ by replacing $\psi_S(x)$ with the smaller tuple $\psi_{S'}(x)$ and, if needed, $\psi_S(y)$ with a smaller tuple $\psi_{S'}(y)$.

2. S' is obtained by the application of rule 2 on $x : C \sqcup D$:
similar to above
3. S' is obtained by the application of rule 3 on $x : succ(k \leq l)$ for a successor y and of rule 9:

The first and second component of $\psi_S(x)$ do not change because this step only adds assertion regarding y . The third component can not increase: If rule 4 is applicable then there is a change that by adding the new assertion $(x, y) : k$ we also add $(x, y) : r, r \in \mathbf{R}$ (by rule 9), which leads to the fact that rule 4 is not applicable any more and hence the tuple becomes smaller. The fourth component becomes smaller because rule 3 can not be applied any more on y and hence the corresponding number in the multiset decreases which lead to a smaller tuple $\psi_{S'}(x)$.

We consider now a tuple $\psi_S(z)$ such that $y \neq z$ and $x \neq z$. The tuple $\psi_S(z)$ can only be affected if z is the predecessor of y . However by Lemma 1.2 a variable can only have one predecessor and hence z must be x .

This means that we can obtain $\Psi(S')$ from $\Psi(S)$ by replacing $\psi_S(x)$ with the smaller tuple $\psi_{S'}(x)$.

4. S' is obtained by the application of rule 4 on a successor y and of rule 9:
The first two component remains unchanged. The third component always decreases because the number of x 's successor on which rule 4 is applicable decreases. We consider now a tuple $\psi_S(z)$ such that $y \neq z$ and $x \neq z$. The tuple $\psi_S(z)$ can only be affected if z is the predecessor of y . However by Lemma 1.2 a variable can only have one predecessor and hence z must be x .
This means that we can obtain $\Psi(S')$ from $\Psi(S)$ by replacing $\psi_S(x)$ with the smaller tuple $\psi_{S'}(x)$.
5. S' is obtained by the application of rule 5 on a assertion $x : succ(ndvd\ l)$ and of rule 9:
The first two components again remains unchanged. lol
6. S' is obtained by the application of rule 7 on an assertion $x : succ(k \leq l)$:
The first two components remains unchanged. The third and fourth component can only decrease because we have one less successor. Furthermore if rule 3 (or 4) is applicable on at least one of the two variables then by merging we can gain a new variable, on which the rule is inapplicable. Therefore those four component can not increase. The fifth component also can not increase: Since we can apply rule 7 on $x : succ(k \leq l)$ we know that this assertion is violated, hence $k^{\mathcal{I}(S)_x} > l^{\mathcal{I}(S)_x}$.
By applying this rule help....

□

References

- [1] F. Baader. A New Description Logic with Set Constraints and Cardinality Constraints on role Successors, 2017.
- [2] F. Baader, I. Horrocks, C. Lutz, and U. Sattler. *An Introduction to Description Logic*. Cambridge University Press, 2017.

- [3] R. Hoehndorf, P. N. Schofield, and G. V. Gkoutos. The role of ontologies in biological and biomedical research: a functional perspective. *Brief. Bioinform*, page 1069–1080, 2015.
- [4] B. Hollunder and F. Baader. Qualifying Number Restrictions Concept Languages. Technical report, Research Report RR-91-03, 1991.
- [5] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *CoRR*, cs.LO/0005014, 05 2000.
- [6] S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, Rheinisch-Westfälischen Technischen Hochschule Aachen, 2001.
- [7] S. Tobies and H. Ganzinger. A PSpace Algorithm for Graded Modal Logic. In *In Proc. CADE 1999, Lecture Notes in Artificial Intelligence*, volume 1632, pages 674–674, 01 1999.