Concept Implicate Tree for Description Logics

Tingting Zou* and Ansheng Deng

Information Science and Technology College, Dalian Maritime University, Dalian, 116026, China

Abstract: Description logics are a class of knowledge representation languages with high expressive power, and the computational complexities of the queries of these expressive description logics are PSPACE-complete. Moreover, knowledge compilation can be regarded as a new direction of research for dealing with the computational intractable reasoning problems. In fact, knowledge compilation based on description logic has been investigated in recent years. However, when the compiled knowledge base is exponential in the size of original knowledge base, the queries are not fast enough. Therefore, we propose a new knowledge compilation method for description logic so that the queries can be done in linear time in the size of the query. In this paper, we first introduce concept implicate tree for ALC concept. Then, we present an algorithm, which can transform an ALC concept into an equivalent concept implicate tree, and prove that each branch of the tree is an implicate of this concept. Finally, we prove that the queries are computable in linear time. Our method has an important property that no matter how large the concept implicate tree is, any query can be done in linear time in the size of the query.

Keywords: ALC, description logic, knowledge compilation, PSAPCE-complete.

1. INTRODUCTION

Description logics (DL) are a class of knowledge representation languages, which can model an application domain of interest by a structured and formally well-understood way [1]. In fact, DLs can be used in various areas, for example, Semantic Web [2, 3], Ontologies [4], and software engineering [5]. Schmidt-Schauß and Smolka proposed description logic ALC, and proved that queries of ALC concepts were PSPACE-complete [6]. Subsequently, Donini et al. provided that queries of ALCN concepts were also PSPACE-complete [7]. With the rapid development of DLs, abundant DL systems have been presented, such as SHIN [8], SHIQ [9], SHOIQ [10, 11], SROIQ [12] and so on. However, the computational complexities of queries of these expressive description logics are intractably.

Knowledge compilation has been emerging as a new direction of research for dealing with the computational intractability of general propositional reasoning [13]. In this approach, reasoning process is split into two phases: off-line compilation and on-line query-answering [14]. In the first phase, the propositional knowledge base is compiled into some target language, which is typically tractable. In the latter phase, the query is actually answered by using the compiled knowledge base of the first phase. The key of this approach is that the knowledge compilation needs to be done only once to be accessible for different queries. Hence, the compiling time can be amortized by many queries concerning the compiled knowledge base [15]. There are many target languages for knowledge compilation, such as prime implicate [16], DNNF [17], and so on. In fact, the queries for these target languages are polynomial time or linear time in the size of the compiled knowledge base. Moreover, Murray and Rosenthal introduced the reduced implicate trie that is a target language for knowledge compilation, and proved that a query can be done in linear time in the size of the query [18-20].

As mentioned above, knowledge compilation is an efficient method to deal with intractable problems. Therefore, many researchers have focused their study on knowledge compilation for description logics in recent years. Selman and Kautz compiled a concept of DL FL into two approximate concepts of DL FL∗; this is the first knowledge compilation method for DL [21]. Subsequently, Furbach and Obermaier introduced the linkless concept description for ALC concepts, which can be regarded as target language for knowledge compilation, and presented an algorithm that transformed ALC concept to equivalent linkless concept description, and proved that queries for such descriptions are in linear time in the size of the descriptions [22]. Later, they used this technique to precompiled ALC concepts and TBoxes so that queries can be done in linear time [23, 24]. Moreover, Bienvenu proposed the prime implicate normal form for ALC concepts, and provide that queries of such forms are polynomial time [25]. Tingting Zou et al proposed a novel knowledge compilation method for description logic based on the concept extension rule [26].

In fact, the queries of these methods are polynomial time or linear time in the size of the compiled knowledge base. However, when the compiled knowledge base is exponential in the size of original knowledge base, the queries are not fast enough. This paper aims to further improve the reduced implicate trie for propositional logic, so that it can be regarded as a much more efficient target language for description log-
ic. Therefore, we propose a new knowledge compilation method for description logic based on the concept implicate tree, for which the queries can be done in linear time in the size of the query regardless of the size of the compiled knowledge base.

In this paper, we first introduce the concept implicate tree for ALC concept, which is a target language for knowledge compilation, and define the concept represented by concept implicate tree. Then, we present an algorithm, which can transform an ALC concepts into concept implicate trees, moreover, we prove that the concept represented by this concept implicate tree is equivalent to original ALC concept, and each branch of tree is an implicate of original concept. Furthermore, we provide that satisfiability-testing and tautology-testing both are linear time with respect to concept implicate tree. Finally, we present an algorithm determining subsumption of two concepts, and prove that subsumption-testing is computable in linear time in the size of the query. In a word, our method has an important property that no matter how large the concept implicate tree is, any query can be done in linear time in the size of the query.

The rest of this paper is organized as follows. In section 2, we define concept implicate tree. Section 3 presents how to transform an ALC concept into an equivalent concept implicate tree. In Section 4, we prove that queries are computable in linear time. Section 5 summarizes our main results.

2. CONCEPT IMPLICATE TREE

Let $C_A$, $R_A$ and $I_A$ be pairwise disjointing sets of atomic concepts, abstract role names, and abstract individuals, and ⊔ operation be concept disjunction, and ⊓ operation be concept conjunction.

**Definition 1.** Literal $L$, ALC concept $C$, and clausal concept $cl$, are defined as follows:

\[
L := \top \mid \bot \mid \neg A \mid \exists R.L \mid \forall R.L \mid \exists R_1 \exists R_2 \ldots \exists R_n A
\]

\[
C := L \mid C \cup C \mid C \cap C
\]

\[
cl := L \mid cl \cup cl
\]

where $A \in C_A$, $R \in R_A$.

**Definition 2.** In literal $L$, $A$ or $\neg A$ is called concept literal, and $A$ is called atomic concept variable, the form $\exists R.L$ or $\forall R.L$ is called role concept variable, and also called role concept variable.

For any concept $C$, $V_{\text{Con}}(C)$ denotes the set of all atomic concept variables of $C$, $V_{\text{Role}}(C)$ denotes the set of all role concept variables of $C$. Moreover, depth($Q.R.L$) denotes the number of the form $Q.R$ in $Q.R.L$, with $Q \in \{\forall, \exists\}$. For example, if

\[
C = (A_1 \cup \neg A_2 \cup \exists R_1 \exists A_3) \cap (A_4 \cup \forall R_1 \exists R_2 \exists A_5)
\]

then,

\[
C_A = \{A_1, A_2, A_3\}, \quad R_A = \{R_1, R_2\}
\]

\[
V_{\text{Con}}(C) = \{A_1, A_2\}, \quad V_{\text{Role}}(C) = \exists R_1 \exists R_2 \exists A_3
\]

\[
depth(\exists R_1 \exists R_2 \exists A_3) = 1, \quad \text{depth}(\forall R_1 \exists R_2 \exists A_3) = 2.
\]

Let $C_1$, $C_2$ be ALC concepts, $B$ be a sub-concept of $C_1$. We will use $C_1 [C_2 / B]$ to refer to the new concept, which is produced by substituting $C_2$ for every occurrence of $B$ that is not in the scope of role restriction in $C_1$. Especially, if $C_2$ is $\top$ or $1$, $B$ is an atomic concept variable $A$ or role concept variable $QR.L$, $Q \in \{\forall, \exists\}$, then $C_1 [\top / B]$ denotes that $\top$ is substituted for $B$. For $\bot$, $\top$ or $1$ is not substituted for $Q.R.B$ or $Q.R.\neg B$.

**Definition 3.** Reduction rules are defined as follows:

\[
C[B / \bot B \bot], \quad C[\bot B \bot B]
\]

\[
C[\bot B \bot B], \quad C[\top B \bot B]
\]

\[
C[\bot B \bot B], \quad C[\top B \bot B]
\]

\[
C[\bot B \bot B], \quad C[\top B \bot B]
\]

\[
C[\bot B \bot B], \quad C[\top B \bot B]
\]

**Definition 4.** Let $V_{\text{Con}}(C) = \{A_i, A_2, \ldots, A_n\}$ be set of atomic concept variables of ALC concept $C$, and $V_{\text{Role}}(C) = \{QR.L_1, \ldots, L_j \}$ be set of role concept variables of ALC concept $C$. A partial ordering relation $\prec$ on sets $V_{\text{Con}}(C)$ and $V_{\text{Role}}(C)$ is defined as follows:

1. $A \prec QR.L$ iff $A \in V_{\text{Con}}(C), \quad QR.L \in V_{\text{Role}}(C)$;
2. $A_i \prec A_j$ iff $i < j$;
3. $Q.R.L_j \prec Q.R.L_i$ iff $\text{depth}(Q.R.L_j) < \text{depth}(Q.R.L_i)$;
4. $Q.R.L_j \prec Q.R.L_i$ iff $i < r$;
5. $Q.R.L_j \prec Q.R.L_i$ iff $j < s$;
6. $\exists RR.L \prec \forall R.L$.

In this paper, we suppose that $V_{\text{Con}}(C)$ and $V_{\text{Role}}(C)$ satisfy this partial ordering relation, that is to say, $V_{\text{Con}}(C)$ and $V_{\text{Role}}(C)$ are ordered sets. For simplicity, we write $V_{\text{Con}}(C)$ as $V_{\text{Con}}$ and $V_{\text{Role}}(C)$ as $V_{\text{Role}}$.

**Definition 5.** Let $C$ be an ALC concept, $cl$ be a clausal concept. Then $cl$ is an implicate of $C$ if and only if $C \subseteq cl$. Moreover, $cl$ is a prime implicate of $C$ if and only if $C \subseteq cl$, and there does not exist an implicate $cl'$ of $C$ such that $C \subseteq cl' \subseteq cl$ and $cl \nsubseteq cl'$.

**Definition 6.** Concept implicate tree(CIT) $T$ for ALC concept $C$ is a tree defined as follows:

1. If $C$ is tautology, then $T$ contains only one root node labelled $\top$;
2. If $C$ is unsatisfiable, then $T$ contains only one root node labelled $\bot$;
3. Otherwise, root node of $T$ is labelled $1$, and for any implicate $cl = L_1 \cup L_2 \cup \cdots \cup L_n$ of $C$, root node has a child
node labelled $L_1$, which is the root of a subtree containing a branch with labels corresponding to $L_2 \sqcup \cdots \sqcup L_m$.

(4) $T$ is reduced by using the rules in Definition 3, until no rule can be applied.

According to definition 6, we can know that each branch of CIT is an implicate of $C$.

**Definition 7.** Let $T$ be a concept implicate tree for ALC concept $C$. Then concept $C_T$ that is represented by the tree $T$ is defined as follows:

1. If $T$ has only one node, then $C_T$ is the label of this node.
2. Otherwise, $C_T$ is the concept disjunction of all concepts, one concept is the label of the root, and the other is concept conjunction of labels of all branches of this root.

**Example 1:** ALC concept

$$C = (A_1 \cup A_2 \cup \forall R_2 \neg A_1) \cap (A_3 \cup A_4 \cup \forall R_2 \neg A_2)$$

where, $V_r(C) = \{A_1, A_2\}$, $V_a(C) = \{\forall R_2 \neg A_1, \forall R_2 \neg A_2\}$.

Then, the concept implicate tree $T$ of $C$ is shown as follows, and each branch of $T$ is an implicate of $C$. For example,$A_1 \cup A_2 \cup \forall R_2 \neg A_1$, $A_3 \cup A_4 \cup \forall R_2 \neg A_2$, and $\neg A_1 \cup \neg A_2 \cup \forall R_2 \neg A_1$,

are all implicates of $C$.

Moreover, the concept $C_T$ is

$$C_T = (A_1 \cup (A_3 \cup A_4 \cup \forall R_2 \neg A_1) \cap (A_3 \cup A_4 \cup \forall R_2 \neg A_2) \cap \forall R_2 \neg A_1 \cap \forall R_2 \neg A_2)$$

and $C_T$ is reduced by using the rules in Definition 3, until no rule can be applied.

According to definition 6, we can know that each branch of CIT is an implicate of $C$.

3. **TRANSFORMATION**

In this section, we introduce a method to transform an ALC concept into an equivalent concept implicate tree, and prove that each branch of the tree is an implicate of this concept. Let $CIMP(C)$ be the sets of implicates of concept $C$.

**Theorem 1.** Let $C$ be ALC concept, $V_{Con}$ be an atomic concept variable of $C$, $V_{rol}$ be a role concept variable of $C$, or a role concept variable $QR.L$, $Q \in \{\forall, \exists\}$, such that $A \in V_{Con}(C)$ and $A \notin V_{Con}(cl)$ (or $QR.L \in V_{rol}(C)$, $QR.L \notin V_{rol}(cl)$), then

$$cl \in CIMP(C[\bot/A]) \cap CIMP(C[\top/A])$$

or $cl \in CIMP(C[\bot / QR.L]) \cap CIMP(C[\top / QR.L])$.

**Proof.** (1) We first prove that $cl \in CIMP(C[\top/A])$. Let $I = < \Delta^1, \cdot^1 >$ be a model of concept $C[\top/A]$, so $C[\top/A]^1 = \emptyset$. Now, we extend $I$ to $I' = < \Delta^1, \cdot^1 >$ by setting $\Delta^1 = \Delta^0$, $A^1 = \Delta^0$, then $C^1 = C[\top/A]^1 = \emptyset$. So $I'$ is a model of $C$. Because clausal concept $cl$ is an implicate of $C$, so $C^1 \subseteq cl^1$. Since $A \notin V_{Con}(cl)$, then $cl^1 = cl^0$. Hence, $(C[\top/A])^1 = C^0 \subseteq cl^1 = cl^0$. Thus, $C[\top/A] \subseteq cl$. According to the Definition 5, therefore $cl \in CIMP(C[\top/A])$. The proof for $C[\bot / QR.L]$ is similarly.

(2) Then, we prove that $cl \in CIMP(C[\bot/A])$. Let $I = < \Delta^1, \cdot^1 >$ be a model of concept $C[\bot/A]$, so $C[\bot/A]^1 = \emptyset$. Now, we extend $I$ to $I' = < \Delta^1, \cdot^1 >$ by setting $\Delta^1 = \Delta^0$, $A' = \neg \Delta^1$, then $C^1 = C[\bot/A]^1 = \emptyset$. So $I' = < \Delta^1, \cdot^1 >$ is a model of concept $C$. Because clausal concept $cl$ is an implicate of concept $C$, so $C^1 \subseteq cl^1$. Since $A \notin V_{Con}(cl)$, then $cl^1 = cl^0$. Hence, $(C[\bot/A])^1 = C^0 \subseteq cl^1 = cl^0$. Thus, $C[\bot/A] \subseteq cl$. According to the Definition 5, therefore, $cl \in CIMP(C[\bot/A])$. The proof for $C[\bot / QR.L]$ is similarly.

Above all, $cl \in CIMP(C[\bot/A]) \cap CIMP(C[\top/A])$ or $cl \in CIMP(C[\bot / QR.L]) \cap CIMP(C[\top / QR.L])$.

**Theorem 2.** Let $C_1$ and $C_2$ be ALC concepts. Then $CIMP(C_1 \cup C_2) = CIMP(C_1) \cap CIMP(C_2)$.

**Proof.** ($\Rightarrow$) Suppose that a clausal concept $cl \in CIMP(C_1 \cup C_2)$, we must prove that $cl \in CIMP(C_1) \cap CIMP(C_2)$. (1) To see that $cl \in CIMP(C_1)$. Let $I = < \Delta^1, \cdot^1 >$ be a model of $C_1$, then $I$ is also a model of $C_1 \cup C_2$. Since $cl \in CIMP(C_1 \cup C_2)$, then $(C_1 \cup C_2)^1 = C_1^1 \cup C_2^1 \subseteq cl^1$. So $C_1^1 \subseteq cl^1$, $C_2^1 \subseteq cl^1$. Thus
cl ∈ Cimp(C). (2) To see that cl ∈ Cimp(C). Let
I = <Δ', E'> be a model of concept C, then I is also a
model of C1 ∪ C2. Since cl ∈ Cimp(C1 ∪ C2), then
(C1 ∪ C2)′ = C′ 1 ∪ C′ 2 ⊆ cl′. So C′ 1 ⊆ cl′, C′ 2 ⊆ cl . Thus
cl ∈ Cimp(C). Therefore, cl ∈ Cimp(C1) ∩ Cimp(C2), and
Cimp(C1 ∪ C2) ⊆ Cimp(C1) ∩ Cimp(C2).

(⇐) Suppose that cl ∈ Cimp(C1) ∩ Cimp(C2), we must
prove that cl ∈ Cimp(C1 ∪ C2). Let I = <Δ', E'> be a model
of C1 ∪ C2, then I is a model of C1 or C2. There are three
cases:

1) I is a model of C1, but not a model of C2. Since
cl ∈ Cimp(C1), then C′ 1 ⊆ cl′. So
(C1 ∪ C2)′ = C′ 1 ∪ C′ 2 = C′ 1 ∪ 0 = C′ 1 ⊆ cl′.

2) I is a model of C2, but not a model of C1. Since
cl ∈ Cimp(C2), then C′ 2 ⊆ cl′. So,

(C1 ∪ C2)′ = C′ 1 ∪ C′ 2 = ⊕ ∪ C′ 2 = C′ 2 ⊆ cl′.

3) I is a model of C1, and also is model of C2. Since
cl ∈ Cimp(C1) ∩ Cimp(C2), then C′ 1 ⊆ cl′, C′ 2 ⊆ cl′.
So (C1 ∪ C2)′ = C′ 1 ∪ C′ 2 ⊆ cl′. Therefore,
cl ∈ Cimp(C1 ∪ C2), and
Cimp(C1) ∩ Cimp(C2) ⊆ Cimp(C1 ∪ C2).

Above all, Cimp(C1 ∪ C2) = Cimp(C1) ∩ Cimp(C2).

**Theorem 3.** Let cl be a clausal concept not containing
concept variable E (E=A or E=QR.L), and C be any ALC
concept. Then cl is an implicite of C iff cl is an implicite of
C[∥/E] ∪ C[∨/E].

Proof. According to Theorem 1 and Theorem 2, it is obvi-
ous that this conclusion is corrected.

According to Theorem 3, the set of implicates of C con-
sists of three parts: (1) the first part is the set of implicates
occurring concept variable E, E=A or E=QR.L; (2) the sec-
ond part is the set of implicite occurring concept variable
¬E, E=A or E=QR.L QR.L; (3) the third part is the set of
implicates of C[∥/E] ∪ C[∨/E], which neither contain
concept variable E nor contain concept variable ¬E, E=A or
E=QR.L.

Therefore, a concept implicite tree T can be regarded as a
ternary tree, each node except leaf node has three subtree,
the first subtree T1, the second subtree T2, and the third sub-

tree T3. Let N be a node labelling Ei, and T1, T2, T3 be three
subtrees of node N. Then the root node of T1 is labelled Ei+1, and
T1 contains the sets of implicates occurring Ei+1. Moreo-

ver, the root node of T2 is labelled ¬Ei+1, and T2 contains the
sets of implicates occurring ¬Ei+1. Furthermore, the root
node of T3 is labelled ⊥, and T3 contains the set of implic-
ates not occurring Ei+1 and ¬Ei+1, which are the intersec-
tion of T1 and T2 not considering Ei+1 and ¬Ei+1.

Therefore, we propose a method to build a concept im-
plicit tree of a given concept. First, we define the structure
of a node of the tree in Fig. (1), then we present algorithm
Simplify and BuildCIT, as shown in Figs. (2) and (3). Algo-
nithm BuildCIT has four input parameters,
Algorithm BuildCIT
Input: concept $C$, $V_{con}$, $V_{rol}$, node $N$;
Output: concept implicate tree $T$;
1. If $C = \bot$ or $C = \top$, then
   build a new CITnode $N'$ of tree $T$, and $N'.label = C$,
   return $T$.
2. If $N-nil$, then
   build a new CITnode $N'$, which is root node of tree $T$,
   and $N'.label = \bot$, return BuildCIT($C$, $V_{con}$, $V_{rol}$, $N'$).
3. If $V_{con} = \emptyset$, then
   select the first role concept variable
   $E = QR, L_i \in V_{rol}$;
   Else, select the first atomic concept variable
   $E = A \in V_{con}$.
4. Let $C_1 = \text{Simplify}(C[\bot / E])$,
   $C_2 = \text{Simplify}(C[\top / E])$.
5. If $C_1 = \bot$, then
   build a new CITnode $N_1$ of tree $T$, and $N_1.label=E$,
   $N_1.label = \top$, $N_1.first = N$.
6. If $C_2 = \top$, then
   build a new CITnode $N_2$ of tree $T$, and $N_2.label = \neg E$,
   $N_2.first = True$, $N_2.second = N$.
7. If $C_3 = \top$, then
   $N.three = nil$, $N.three = nil$;
   Else
   build a new CITnode $N_3$ of tree $T$, and $N_3.label = E$,
   $N_3.first = N_1$.
   If $V_{con} \neq \emptyset$, then
   call BuildCIT($C_1$, $V_{con} - \{E\}$, $V_{rol}$, $N_1$);
   Else
   call BuildCIT($C_1$, $V_{con} \cap V_{rol}$, $N_1$).
8. If $C_2 = \top$, then
   $N.second = nil$, $N.three = nil$;
   Else
   build a new CITnode $N_4$ of tree $T$, and $N_4.label = \neg E$,
   $N_4.second = N_2$.
   If $V_{con} \neq \emptyset$, then
   call BuildCIT($C_2$, $V_{con} - \{E\}$, $V_{rol}$, $N_2$);
   Else
   call BuildCIT($C_2$, $V_{con} \cap V_{rol}$, $N_2$).
9. If ($N_1.label$ and $N_2.label$), then
   delete node $N_1$, $N_2$, and $N.label = True$, return $T$.
10. If ($N.first = nil$ and $N.second = nil$), then
    build a new CITnode $N_5$ of tree $T$, and $N.three = N_3$,
    call BuildThird($N_1$, $N_2$, $N_3$), $N.label = \bot$.
11. Return $T$.

Algorithm BuildThird
Input: CIT nodes $N_1$, $N_2$, $N_3$;
Output: tree $T$.
1. $N_1.label = N_1.label$.
2. If $N_1.label = true$ and $N_2.label = true$,
   then $N_3.label = true$, return $T$.
3. If $N_1.label = true$,
   then $N_1.first = N_3.first$, $N_1.second = N_3.second$,
   $N_1.three = N_3.three$, return $T$.
4. If $N_2.label = true$,
   then $N_2.first = N_1.first$, $N_2.second = N_1.second$,
   $N_2.three = N_1.three$, return $T$.
5. If $N_1.first = nil$ or $N_2.first = nil$,
   then $N_3.first = nil$;
   Else
   build a new CITnode $N_31$ of tree $T$,
   $N_1.first = N_31$, call BuildThird($N_1$, $N_31$, $N_31$).
6. If $N_1.second = nil$ or $N_2.second = nil$,
   then $N_3.second = nil$;
   Else
   build a new CITnode $N_32$ of tree $T$,
   $N_1.second = N_32$, call BuildThird($N_1$, $N_32$, $N_32$).
7. If $N_3.three = nil$ or $N_3.three = nil$,
   then $N_3.three = nil$;
   Else
   build a new CITnode $N_33$ of tree $T$,
   $N_3.three = N_33$, call BuildThird($N_1$, $N_33$, $N_33$).
8. Return $T$.

Fig. (4). Algorithm BuildThird.
5) We build a new CITnode $N_2$ of tree $T$, and $N_2.label = \neg A_1$, $N_2.second = N_2$, call BuildCIT($C_2$, \{$A_2\}, V_{rol}, N_2$);
6) We build a new CITnode $N_3$ of tree $T$, and $N.three = N_3$,
call BuildThird($N_1$, $N_2$, $N_3$). $N_3.label = \bot$;
7) Return $T$.

In this algorithm, we call three algorithms,
BuildCIT($C_1$, \{$A_2\}, V_{rol}, N_1$), BuildCIT($C_2$, \{$A_2\}, V_{rol}, N_2$),
BuildCIT($C_3$, \{$A_2\}, V_{rol}, N_3$).
and BuildThird($N_1, N_2, N_3$). Algorithms BuildCIT($C_1$, \{\$A_2\}, VRol,$N_1$) and BuildCIT($C_2$, \{\$A_2\}, VRol,$N_2$) iterate the process of algorithm BuildCIT($C$, VRol,$N'$), build the first and second subtrees of node $N'$. Algorithm BuildThird($N_1, N_2, N_3$) builds the third subtree of node $N'$, finally, algorithm BuildCIT($C$, VRol,$N'$) returns the concept implicate tree $T$ of concept $C$ in the following.

**Theorem 4.** Let $C$ be ALC concept that contains only one concept variable $E$, $T$ be a tree of $C$ built by the algorithm BuildCIT, $C_T$ be a concept represented by the $T$. Then $C_T$ is logically equivalent to $C$, and is one of the concepts $\top$, $\bot$, $E$, $\neg E$.

Proof. The concept $C$ must be one of the following four concepts: $\bot$, $\top$, $E$, $\neg E$. If $C = \bot$ or $C = \top$, then the algorithm BuildCIT builds tree $T$, which contains only one node labelling $\bot$ or $\top$. Thus, $C_T = \bot$ or $C_T = \top$, and $C_T$ is logically equivalent to $C$. If $C = E$, then the algorithm BuildCIT builds tree $T$, which contains a root node labelling $E$ and the first sub-node labelling $E$. Thus, $C_T = E \lor E = E$, and $C_T$ is logically equivalent to $C$. If $C = \neg E$, then the algorithm BuildCIT builds tree $T$, which contains a root node labelling $\bot$ and the second sub-node labelling $\neg E$. Thus, $C_T = \bot \lor \neg E = \neg E$, and $C_T$ is logically equivalent to $C$. Therefore, $C_T$ is logically equivalent to $C$, and is one of the concepts $\bot$, $\top$, $E$, $\neg E$.

**Theorem 5.** Let $C$ be an ALC concept, $V_{Con}$ be an atomic concept variables set of $C$, $V_{Rol}$ be a role concept variables set of $C$, $T$ be a tree of $C$ built by algorithm BuildCIT, $C_T$ be a concept represented by the $T$. Then $C_T$ is logically equivalent to $C$, and each branch of $T$ is an implicate of $C$.

Proof. (1) We first prove logic equivalence by induction on the number $m$ of concept variables in $C$, let $V = V_{Con} \cup V_{Rol} = \{E_1, \ldots , E_m\}$, $E_k \in V_{Con}$ or $E_k = QR_i \cdot L_i \in V_{Rol}$, $1 \leq k \leq m$, $Q \in \{\forall, \exists\}$.

<br>

1> Base case: Let $m=1$, According to theorem 4, then $C_T$ is logically equivalent to $C$.

2> Inductive hypothesis: Assume the theorem holds for all concepts with at most $m$ concept variables.

3> Induction: Suppose that $C$ has $m+1$ concept variables. Let $E_i$ be any concept variable of $V$, $E_i \in V_{Con}$ or $E_i \in V_{Rol}$, $1 \leq i \leq m$, now assume $E_i$ is an atomic concept variable form $V_{Con}$. Then we must prove that $C \equiv C_T = (E_i \cup C_{BuildCIT(C_i/\{E_i\}, VRol_{\{E_i\}})}) 
\supseteq (\neg E_i \cup C_{BuildCIT(C_i/\{E_i\}, VRol_{\{E_i\}})}) 
\supseteq (C_{BuildCIT(C_i/\{E_i\}, VRol_{\{E_i\}})}).$

According to the inductive hypothesis, we obtain, $C[\top / E_i] \equiv C_{BuildCIT(C_i/\{E_i\}, VRol_{\{E_i\}})}$.

$C[\bot / E_i]$ logically equivalence to the concept conjunction of the labels $\top$ that is similar to distributive laws of proposition logic, $C_T$ is logically equivalent to $C_T$ in the following.

Let $T = \langle \Delta', \bullet' \rangle$ be any model of concept $C$, so $C_T \equiv \emptyset$, and there exists an individual $a$ such that $a \in C_T$. In the following, we consider two cases of the individual $a$.

Case 1, we suppose $a \in E_i$. Hence, $a \in (C[\top / E_i])$, and $a \in (C[\bot / E_i])$, $a \in (C[\top / E_i])$. Thus, $a \in (C_T)$. Hence, $C_T \subseteq C_T$, and $C_T \subseteq C_T$. Case 2, we suppose $a \not\in E_i$. Hence $a \in (E_i \cup C[\bot / E_i])$, $a \not\in (C[\bot / E_i])$. Thus $a \not\in (C_T)$, and $C_T \subseteq C_T$. Therefore, $C_T \subseteq C_T$. The proof that $C_T \subseteq C_T$ is similar.

(2) Now, we show that each branch of $T$ is an implicate of $C$. According to the distributive laws of description logic that is similar to distributive laws of proposition logic, $C_T$ is logically equivalent to the concept conjunction of the labels of its branches. Moreover, due to the interpretation of concept conjunction, therefore, each branch of $T$ is an implicate of $C$.

**Theorem 6.** Let $C$ be an ALC concept, $V_{Con}$ be an atomic concept variables set of $C$, $V_{Rol}$ be a role concept variables set of $C$. Then algorithm BuildCIT is sound and complete.

Proof. (1) We first prove the sound of algorithm. Algorithm BuildCIT builds a tree $T$, and according to theorem 5, each branch of $T$ is an implicate of $C$, so $T$ is a concept implicate tree of $C$. Thus, the algorithm BuildCIT is sound.

(2) Now, we must show the complete of algorithm. For any concept $C$, algorithm BuildCIT can builds a corresponding tree $T$, and there does not exist a concept that has no corresponding tree. Thus, the algorithm BuildCIT is complete.

### 4. TRACTABLE QUERYING

In this section, for any concept represented by a concept implicate tree, we show that queries are computable in linear time in the size of the query.

Let $C$ be any ALC concept and $T$ be a concept implicate tree of $C$. There are three queries for ALC concepts, satisfiability-testing, tautology-testing, and subsumption-testing.

Considering the satisfiability-testing, if $T$ contains only one node that is labelled $\bot$, then $C$ is unsatisfiability, otherwise $C$ is satisfiability. With regard to the tautology-testing, if $T$ contains only one node that is labelled $\top$, then $C$
is tautology, otherwise C is not tautology. Obviously, these two queries can be done in linear time.

In order to test subsumption between two concepts, we first provide some theorems in the following.

Let \( cl \) be a clausal concept containing a concept literal \( L \) or a role concept literal \( L \). Then, \( cl|/L) \) denotes a new clausal concept that deletes literal \( L \) from \( cl \). A prefix of a clausal concept \( cl = L_1 \cup \{ L_2 \} \cup \cdots \cup L_s \) is a clausal concept of the form \( cl' = L_1 \cup L_2 \cup \cdots \cup L_s \), \( 0 \leq t \leq s \). If \( t = 0 \), then the prefix is \( \perp \).

**Theorem 7**. Let \( C \) be an ALC concept, and \( cl \) be an implicate of \( C \) containing a literal \( L \). Then \( (cl / \{ L \}) \in \text{Cimp}(C / \{ L \}) \).

Proof. Let \( I = \langle \Delta, \bullet' \rangle \) be any model of concept \( C / \{ L \} \), then \( (C / \{ L \}) \neq \emptyset \). Now, we extend \( I \) to \( I' = \langle \Delta', \bullet' \rangle \) by setting \( \Delta' = \Delta \cup L = \emptyset \), \( \bullet' = \emptyset \), then \( I' \) is a model of concept \( C \). So, \( (C / \{ L \}) = (cl / \{ L \}) \). Since \( cl \) is an implicate of concept \( C \), hence, \( C' \subseteq cl' \). Moreover, \( cl' = (cl / \{ L \})' \). Thus, \( (C / \{ L \}) \subseteq (cl / \{ L \})' \), and \( C / \{ L \} \supseteq (cl / \{ L \}) \). Therefore, \( (cl / \{ L \}) \in \text{Cimp}(C / \{ L \}) \).

**Theorem 8**. Let \( C \) be an ALC concept, \( V_{\text{Con}} \) be an atomic concept variables set of \( C \), \( V_{\text{Rol}} \) be a role concept variables set of \( C \). \( T \) be a tree of \( C \) built by algorithm BuildCIT, and clausal concept \( cl \) be an implicate of \( C \). Then there is a unique prefix of \( cl \) that is a branch of \( T \).

Proof. We prove that there is a unique prefix of \( cl \), which is a branch of \( T \), by induction on the number of concept variables in \( C \). Let \( V = V_{\text{Con}} \cup V_{\text{Rol}} = \{ E_1, \ldots, E_m \} \), \( E_k \in V_{\text{Con}} \) or \( E_i = QR_j, L_i \in V_{\text{Rol}}, 1 \leq k < l \leq m \).

1) Base case: Theorem 4 takes care of the case \( m = 1 \).

2) Inductive hypothesis: Assume the theorem holds for all concepts with at most \( m \) concept variables.

3) Induction: Suppose that \( C \) has \( m+1 \) concept variables. Let \( cl = L_{d_1} \cup L_{d_2} \cup \cdots \cup L_{d_s} \) be an implicate of \( C \), where \( L_{d_i} \) is either \( E_{d_i} \) or \( \neg E_{d_i} \), and \( d_1 < d_2 < \cdots < d_s \).

So, we must show that there is a unique prefix of \( cl \) that is a branch of \( T \). Let \( E_i \) be any concept variable of \( V_{\text{Con}} \) or \( E_i \in V_{\text{Rol}}, 1 \leq i \leq m \), now assume \( E_1 \) is an atomic concept variable form \( V \). Then we must show that there is a unique prefix of \( cl \) that is a branch of \( C_T \), which is the concept

\[
C_T = (E_1 \cup C_{\text{BuildCIT}}(\{ E_1 \} \cup V_{\text{Con}} \cup \{ E_i \} \cup V_{\text{Rol}})) \cap (\neg E_2 \cup C_{\text{BuildCIT}}(\{ E_1 \} \cup V_{\text{Con}} \cup \{ E_i \} \cup V_{\text{Rol}})) \cap (C_{\text{BuildCIT}}(\{ E_1 \} \cup V_{\text{Con}} \cup \{ E_i \} \cup V_{\text{Rol}}))
\]

By the inductive hypothesis, there is a unique prefix of \( cl \) that is a branch of the intersection of two subtrees \( C_{\text{BuildCIT}}(\{ E_1 \} \cup V_{\text{Con}} \cup \{ E_i \} \cup V_{\text{Rol}}) \) and \( C_{\text{BuildCIT}}(\{ E_1 \} \cup V_{\text{Con}} \cup \{ E_i \} \cup V_{\text{Rol}}) \). In this case, the theorem holds for the third branch of \( C_T \). Moreover, if \( d_i > 1 \), nothing is needed to prove, so assume \( d_i = 1 \). Let \( L_i \) be either \( E_i \) or \( \neg E_i \), there are two cases.

Case 1: we suppose that \( L_i = E_i \). According to Theorem 7, then \( cl | E_i \) is a branch of \( C \). Moreover, by the inductive hypothesis, there is a unique prefix \( G \) of \( cl | E_i \) that is a branch of \( BuildCIT(\{ E_1 \} \cup V_{\text{Con}} \cup \{ E_i \} \cup V_{\text{Rol}}) \). So \( H = E_i \cup G \) is a prefix of \( cl \) that is a branch of \( T \). Now, we prove that \( H \) is a unique prefix of \( cl \) by contradiction. Suppose that \( H' \) is another prefix of \( cl \) that is a branch of \( T \). Let \( H' = E_i \cup G' \), then \( G' \) is a prefix of \( cl | E_i \) that is a branch of \( BuildCIT(\{ E_1 \} \cup V_{\text{Con}} \cup \{ E_i \} \cup V_{\text{Rol}}) \). However, \( G \) is a unique prefix of \( cl | E_i \) by the inductive hypothesis, so \( G' = G \), \( H' = H \). Therefore, \( H \) is a unique prefix of \( cl \) that is a branch of \( T \).

Case 2: we suppose that \( L_i = \neg E_i \). According to Theorem 7, then \( cl | \neg E_i \) is an implicate of \( C \). Moreover, by the inductive hypothesis, there is a unique prefix \( G \) of \( cl | \neg E_i \) that is equivalent to \( C \). Moreover, by the inductive hypothesis, there is a unique prefix \( G \) of \( cl | \neg E_i \) that is a branch of \( BuildCIT(\{ E_1 \} \cup V_{\text{Con}} \cup \{ E_i \} \cup V_{\text{Rol}}) \). So \( H = E_i \cup G \) is a prefix of \( cl \) that is a branch of \( T \). Now, we prove that \( H \) is a unique prefix of \( cl \) by contradiction. Suppose that \( H' \) is another prefix of \( cl \) that is a branch of \( T \). Let \( H' = E_i \cup G' \), then \( G' \) is a prefix of \( cl | \neg E_i \) that is a branch of \( BuildCIT(\{ E_1 \} \cup V_{\text{Con}} \cup \{ E_i \} \cup V_{\text{Rol}}) \). However, \( G \) is a unique prefix of \( cl | \neg E_i \) by the inductive hypothesis, so \( G' = G \), \( H' = H \). Therefore, \( H \) is a unique prefix of \( cl \) that is a branch of \( T \).

Above all, there is a unique prefix of \( cl \) that is a branch of \( T \).

**Theorem 9**. Let \( C \) be an ALC concept, \( V_{\text{Con}} \) be an atomic concept variables set of \( C \), \( V_{\text{Rol}} \) be a role concept variables set of \( C \). \( T \) be a concept implicate tree of \( C \). Then every prime implicate of \( C \) is a branch of \( T \).

Proof. According to Theorem 8, the prefix of an implicate has unique. Thus, it is obvious that the conclusion holds.

In example 2, all prime implicants of concept \( C \) are \( A_i \cup A_j \cup \forall R_i, \neg A_i \), \( A_k \subseteq \forall R_i, \exists R_j A_j \).

\[ \neg A_2 \cup \forall R_i, \exists R_j A_j \]

Considering concept implicate tree \( T \), they both are branch of \( T \).

**Theorem 10**. Let \( C \) be an ALC concept, \( V_{\text{Con}} \) be an atomic concept variables set of \( C \), \( V_{\text{Rol}} \) be a role concept variables set of \( C \). \( T \) be a concept implicate tree of \( C \). Then every sub-
suming implicate (including any prime implicate) of a branch of T contains the literal labelling the leaf of that branch.

Proof. According to Theorem 8, it is obvious that this conclusion holds.

In example 2, an implicate \( A_i \cup A_j \cup \forall R_i \neg A_k \) contains the literal \( \forall R_i \neg A_k \), which is a label of the leaf of a branch of T. Moreover, all other implicates of C in T, for example, \( A_i \cup A_j \cup \forall R_i \exists R_j A_k \), \( A_i \cup A_j \cup \forall R_i \exists R_j A_k \), \( A_i \cup \neg A_k \cup \forall R_i \exists R_j A_k \), \( A_i \cup \neg A_k \cup \forall R_i \exists R_j A_k \), \( A_i \cup \forall R_i \exists R_j A_k \), \( \neg A_k \cup \forall R_i \exists R_j A_k \), \( \neg A_k \cup \forall R_i \exists R_j A_k \), \( \neg A_k \cup \forall R_i \exists R_j A_k \), \( \neg A_k \cup \forall R_i \exists R_j A_k \), contain the literal \( \forall R_i \exists R_j A_k \), which is a label of the leaf of a branch of T.

Based on the above theorems, if clausal concept \( cl = L_1 \cup L_2 \cup \cdots \cup L_d \) is an implicate of concept C, and T is a concept implicate tree of C, then there is a unique prefix \( cl' = L_1 \cup \cdots \cup L_t \) of cl that is a branch of T, \( 1 \leq t \leq d \), and each literal \( L_i \) is a label of that branch, \( 1 \leq i \leq t \), and \( L_i \) is a label of leaf node. Therefore, we present algorithm Subsume as shown in Fig. (5). The main idea is that determining whether \( cl \) is an implicate of C iff there exists a branch labelled the prefix of \( cl \).

According to the algorithm, it is obvious that the subsumption-testing can be done by traversing a single branch. Therefore, the time complexity is linear time in the size of the query, but not in the size of T. This is an important property of our method.

**Theorem 11.** Let C be an ALC concept, and T be a concept implicate tree of C, and \( cl \) be a clausal concept. Then it can be decided in linear time in \(|cl|\) whether \( C \subseteq cl \). \(|cl|\) denotes the number of all literal in \( cl \).

### Algorithm Subsume

**Input:** concept implicate tree T of concept C, clausal concept \( cl \);

**Output:** Yes, if \( C \subseteq cl \); No, if \( C \not\subseteq cl \).

1. If \( cl = \top \), then return Yes.
2. If tree T has only one node labelled \( \bot \), then return Yes.
3. If tree T has only one node labelled \( \top \), then return No.
4. If \( cl = \bot \), then return No.
5. For each literal \( L_i \) of \( cl \)
   - If there exists a branch \( w \) of T, such that \( L_1 \cup \cdots \cup L_i \) is a label of \( w \) and \( L_i \) is a label of leaf node of \( w \), then return Yes.
6. Return No.

**Fig. (5).** Algorithm Subsume.

Proof. Considering the algorithm Subsume, it is obvious that the first four steps of the algorithm can be done in linear time. For the fifth step, the algorithm detects all the literal in \( cl \) to decide whether \( C \subseteq cl \). Therefore, determining whether \( C \subseteq cl \) can be done in linear time in \(|cl|\).

Above all, three queries for ALC concepts, satisfiability-testing, tautology-testing, and subsumption-testing, can be done in linear time in the size of the query.

### CONCLUSION

In this paper, we presented knowledge compilation for description logic based on concept implicate tree. Firstly, we defined concept implicate tree for ALC concept. Moreover, we provided an algorithm to translate arbitrary ALC concept into an equivalent concept implicate tree. Finally, we proved that satisfiability-testing, autology-testing and subsumption-testing were computable in linear time with respect to the concept implicate tree. It is important that any query can be done in linear time in the size of the query, regardless of the size of the concept implicate tree. In a word, our method is an effective method to deal with knowledge compilation for description logic.

### CONFLICT OF INTEREST

The authors confirm that this article content has no conflict of interest.

### ACKNOWLEDGEMENTS

This work was partially supported by Liaoning Province Natural Science Fund Project of China (2015020034), and National Natural Science Foundation of China (61272171), and the Fundamental Research Funds for the Central Universities of China (3132015044).

### REFERENCES


