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 is a class of knowledge representation languages with high expressive power, and the computational complexities of the queries of these expressive description logics are defined as 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 as compared to original knowledge base, the queries are not. Therefore, we proposed a new knowledge compilation method for description logic to solve the queries in linear time depending on the size of the query. In this paper, we first introduced the concept implicate tree for the ALC concept. Then, we present an algorithm, which can transform an ALC concept into an equivalent implicate concept tree, and proved that each branch of the tree is an implicate of this concept. Finally, we proved that the queries are computable in linear time. The proposed method has an important property that no matter how large the concept implicate tree is, any query can be resolved in linear time depending on the size of the query.

Keywords: ALC, Description logic, Knowledge compilation, PSAPCE-complete, Algorithm Build CIT, Tractable querying.

1. INTRODUCTION

Description logics (DL) is a class of knowledge representation languages, which can model an application domain of interest by a structured and formally well-understood method[1]. In fact, DLs can be used in various areas, for example, Semantic Web [2, 3], Ontologies [4], and software engineering [5]. Schmidt-Schauf and Smolka proposed description logic ALC, and proved that the queries of ALC concepts were PSPACE-complete [6]. Subsequently, Donini et al. stated that the 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 the queries of these expressive description logics are intrate.

Knowledge compilation has emerged 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 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 based on polynomial time or linear time dependent on the size of the compiled knowledge base. Moreover, Murray and Rosenthal introduced the reduced implicate tree that is a target language for knowledge compilation, and proved that a query can be done in linear time considering 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 conducted their studies 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, being 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 a target language for knowledge compilation, by presenting an algorithm that transformed ALC concept to equivalent linkless concept description, and proved that queries for such descriptions were resolved in linear time based on the size of the descriptions [22]. Later, they used this technique for precompiled ALC concepts and TBoxes so that queries can be addressed in linear time [23, 24]. Moreover, Bienvenu proposed the prime implicate normal form for ALC concepts, and concluded that the queries of such forms are based on 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 were also based on the polynomial time or linear time depending on the size of the compiled knowledge base. However, when the compiled knowledge base was exponential in terms of the size of the original knowledge base, the queries were not addressed rapidly. This paper aims to further improve the reduced implicate tree for propositional logic, to make it a much more
efficient description logic for target language. Therefore, we proposed a new knowledge compilation method for the description logic based on the concept implicate tree, for which the queries can be addressed in linear time based on the size of the query regardless of the size of the compiled knowledge base.

In this paper, we first introduced the concept implicate tree for ALC concept, which is a target language for knowledge compilation, and defined the concept represented by concept implicate tree. Then, an algorithm was presented, which can transform ALC concepts into the concept implicate trees. Moreover, we proved that the concept represented by this concept implicate tree was equivalent to the original ALC concept, and each branch of the tree was an implicate of the original concept. Furthermore, we explained that the satisfiability-testing and tautology-testing were carried out in linear time with respect to the concept implicate tree. Finally, we presented an algorithm determining the subsumption of two concepts, and proved that subsumption-testing was computable in linear time based on the size of the query. In a word, this method has an important property that no matter how large the concept implicate tree is, any query can be assessed in linear time depending on the size of the query.

The rest of this paper is organized as follows. In section 2, the concept implicate tree is defined. Section 3 presents the process of transforming an ALC concept into an equivalent concept implicate tree. In Section 4, it is proved that the queries are computable in linear time. Section 5 summarizes the main results.

2. CONCEPT IMPLICATE TREE

Let \( C_A, R_A \) and \( I_A \) be the pairwise disjointing sets of atomic concepts, abstract role names, and abstract individuals, respectively, and \( \sqcup \) operation be the concept disjunction, with \( \cap \) operation being the concept conjunction.

Definition 1. Literal \( L \), ALC concept \( C \), and clausal concept \( cl \), are defined as follows:
\[
L := | A | \cap | \neg A | \cap | \exists R.A | \cap | \forall R.A |
\]
\[
C := L | C \cap C | C \cap N | C
\]
\[
cl := L | cl \cup cl
\]
where \( A \in C_A \), \( R \in R_A \).

Definition 2. In literal \( L \), \( A \) or \( \neg A \) is called the concept literal, and \( A \) is known as the atomic concept variable, with the form \( \exists R.A \) or \( \forall R.A \) known as the role concept literal and also as the role concept variable.

For any concept \( C \), \( V_{\text{Comp}}(C) \) denotes the set of all atomic concept variables of \( C \), and \( V_{\text{Role}}(C) \) denotes the set of all role concept variables of \( C \). Moreover, \( \text{depth}(Q.R.L) \) denotes the number of the form \( Q.R \) in \( Q.R.L \), \( Q \in \{ \forall, \exists \} \). For example, if
\[
C = (A_1 \cup \neg A_1 \cup \exists R_1, \neg A_1) \cap (A_2 \cup \forall R_2, \exists R_2, A_2)
\]
then,
\[
C_A = \{ A_1, A_2, A_3 \}, \quad R_A = \{ R_1, R_2 \},
\]
\[
V_{\text{Comp}}(C) = \{ A_1, A_2 \}, \quad V_{\text{Role}}(C) = \{ \exists R_1, \neg A_1, \forall R_1, \exists R_2, A_2 \},
\]
\[
\text{depth}(\exists R_1, \neg A_1) = 1, \quad \text{depth}(\forall R_1, \exists R_2, A_2) = 2.
\]

Let \( C_1 \) and \( C_2 \) be the ALC concepts, and \( B \) is the subconcept of \( C_1 \), \( C_1 \mid (C_2 \mid B) \) is used 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 \( Q.R.L \), \( Q \in \{ \forall, \exists \} \), then \( C_1[\top / B] \) denotes that \( \top \) is substituted for \( B \), and \( 1 \) for \( \neg B \), but \( \top \) or \( 1 \) is not substituted for \( Q.R.B \) or \( Q.R.\neg B \).

Definition 3. Reduction rules are defined as follows:
\[
C[B / B \sqcup \bot], \quad C[\bot / B \sqcup \bot],
\]
\[
C[B / B \sqcup \top], \quad C[\top / B \sqcup \top],
\]
\[
C[\bot / B \sqcup \neg B], \quad C[\top / B \sqcup \neg B],
\]
\[
C[\bot / \exists R.\bot], \quad C[\top / \forall R.\top].
\]

Definition 4. Let \( V_{\text{Comp}}(C) = \{ A_1, A_2, \ldots, A_j \} \) be the set of atomic concept variables of ALC concept \( C \), and
\[
V_{\text{Role}}(C) = \{ Q.R.L \mid Q \in \{ \forall, \exists \}, 1 \leq i \leq p, 1 \leq j \leq q \}
\]
be the set of role concept variables of ALC concept \( C \). A partial ordering relation \( \prec \) on sets \( V_{\text{Comp}}(C) \) and \( V_{\text{Role}}(C) \) is defined as follows:
\[
(1) \quad A \prec Q.R.L \quad \text{iff} \quad A \in V_{\text{Comp}}(C), Q.R.L \in V_{\text{Role}}(C);
\]
\[
(2) \quad A_i \prec A_j \quad \text{iff} \quad i < j;
\]
\[
(3) \quad Q.R.L_i \prec Q.R.L_j \quad \text{iff} \quad \text{depth}(Q.R.L_i) < \text{depth}(Q.R.L_j);
\]
\[
(4) \quad Q.R.L_i \prec Q.R.L_j \quad \text{iff} \quad i < j;
\]
\[
(5) \quad Q.R.L_i \prec Q.R.L_j \quad \text{iff} \quad j < s;
\]
\[
(6) \quad Q.R.L \prec \forall R.L.
\]

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

Definition 5. Let \( C \) be an ALC concept, and \( cl \) 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 \( cl \not\subseteq cl' \).

Definition 6. Concept implicate tree (CIT) \( T \) for ALC concept \( C \) is a tree defined as follows:
\[
(1) \quad \text{If} \ C \text{ is tautology, then} \ T \text{ contains only one root node labelled as} \top;
\]
(2) If $C$ is unsatisfiable, then $T$ contains only one root node labeled as $1$.

(3) Otherwise, root node of $T$ is labelled as $1$, and for any implicate $cI = L_1 \cup L_2 \cup \cdots \cup L_m$ of $C$, root node has a child node labelled as $L_1$, which is the root of a subtree containing a branch with labels corresponding to $L_2 \cup \cdots \cup L_m$.

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

According to definition 6, it can be observed that each branch of CIT $T$ is an implicate of $C$.

**Definition 7.** Let $T$ be the 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 two concepts; one concept is the label of the root, and the other is the concept conjunction of labels of all branches of this root.

**Example 1:** ALC concept

\[ C = (A_1 \cup A_2 \cup \forall A_1 \cdot \neg A_4) \cap (A_1 \cup \forall A_2 \cdot \exists A_4) \]

where, $V_C(C) = \{A_1, A_2\}$, $V_a(C) = \{\forall A_1 \cdot \neg A_4, \forall A_2 \cdot \exists A_4\}$.

Then, the concept implicate tree $T$ of $C$ is shown as follows, and each branch of $T$ is an implicate of $C$. For example,

- $A_1 \cup A_2 \cup \forall A_1 \cdot \neg A_4 \cup \forall A_2 \cdot \exists A_4$,
- $\neg A_1 \cup \neg A_2 \cup \forall A_1 \cdot \exists A_4$,

are all implicates of $C$.

Moreover, the concept $C_T$ is

\[ C_T = (A_1 \cup (A_1 \cup (\forall A_2 \cdot \neg A_4) \cap (\exists A_4 \cup \forall A_2 \cdot \exists A_4)) \]

\[ \cap (\forall A_2 \cdot \exists A_4) \cap (\neg A_4) \cap ((\forall A_2 \cup \forall A_1 \cdot \exists A_4) \cap (\forall A_2 \cdot \exists A_4)) \]

\[ \cap (\forall A_2 \cdot \exists A_4) \cap (\forall A_2 \cdot \exists A_4) \cap (\forall A_2 \cdot \exists A_4). \]

3. **TRANSFORMATION**

In this section, we introduced a method to transform an ALC concept into an equivalent concept implicate tree, and proved 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 the ALC concept, $V_{\text{Con}}$ be the atomic concept variables of $C$, $V_{\text{rol}}$ be a role concept variables of $C$, and clausal concept $cl$ be an implicate of C. If there exists an atomic concept variable $A$ (or a role concept variable $Q R L$, $Q \in \{\forall, \exists\}$), such that $A \in V_{\text{Con}}(C)$ and $A \notin V_{\text{Con}}(cl)$ (or $QR L \in V_{\text{rol}}(C)$, $QR L \notin V_{\text{rol}}(cl)$), then

\[ cl \in \text{Cimp}(C[\perp / A]) \cap \text{Cimp}(C[\top / A]) \]

or

\[ cl \in \text{Cimp}(C[\perp / QR L]) \cap \text{Cimp}(C[\top / QR L]). \]

Proof. (1) We first proved that $cl \in \text{Cimp}(C[\top / A])$. Let $I = \langle \Delta^\prime, \cdot \rangle$ be a model of concept $C[\top / A]$, therefore, $(C[\top / A])' = \emptyset$. Following this, we extended $I$ to $I = \langle \Delta^\prime, \cdot \rangle$ by setting $\Delta^\prime = \Delta^\prime$, $A^\prime = A^\prime$, and $C^\prime = (C[\top / A])' = \emptyset$. Therefore, $I'$ became the model of C. Because clausal concept $cl$ was an implicate of $C$, therefore, $C' \subseteq cl'$. Since $A \notin V_{\text{Con}}(cl)$, then $cl' = cl'$. Hence, $(C[\top / A]) \subseteq C' \subseteq cl'$. Thus, $(C[\top / A]) \subseteq cl$. According to the Definition 5, $cl \in \text{Cimp}(C[\top / A]).$ The proof for $(C[\top / QR L])$ is similar.

(2) Following this, we proved that $cl \in \text{Cimp}(C[\perp / A])$. Let $I = \langle \Delta^\prime, \cdot \rangle$ be the model of concept $C[\perp / A]$, therefore, $(C[\perp / A])' = \emptyset$. Now, we extended $I$ to $I = \langle \Delta^\prime, \cdot \rangle$ by setting $\Delta^\prime = \Delta^\prime$, $A^\prime = \neg A^\prime$, and $C^\prime = (C[\perp / A])' = \emptyset$. Therefore, $I = \langle \Delta^\prime, \cdot \rangle$ is the model of concept $C$. Because clausal concept $cl$ was an implicate of $C$, therefore, $C' \subseteq cl'$. Since $A \notin V_{\text{Con}}(cl)$, then $cl' = cl'$. Hence, $(C[\perp / A]) \subseteq C' \subseteq cl'$. Thus, $(C[\perp / A]) \subseteq cl$. According to the Definition 5, $cl \in \text{Cimp}(C[\perp / A]).$ The proof for $(C[\perp / QR L])$ is similar.
Above all, \( cl \in \text{Cimp}(C[\bot \cup A]) \cap \text{Cimp}(C[\top \cup A]) \) or \( cl \in \text{Cimp}(C[\bot \cup QR.L]) \cap \text{Cimp}(C[\top \cup QR.L]) \).

**Theorem 2.** Let \( C_1 \) and \( C_2 \) be the ALC concepts. Then, \( \text{Cimp}(C_1 \cup C_2) = \text{Cimp}(C_1) \cap \text{Cimp}(C_2) \).

Proof. (\( \Rightarrow \)) Assuming that a clausal concept \( cl \in \text{Cimp}(C_1 \cup C_2) \), we proved that \( cl \in \text{Cimp}(C_1) \cap \text{Cimp}(C_2) \). (1) To see that \( cl \in \text{Cimp}(C_i) \).

Let \( I = \langle \Delta' \cdot \bullet' \rangle \) be the model of \( C_i \), then \( I \) is also a model of \( C_1 \cup C_2 \). Since \( cl \in \text{Cimp}(C_1 \cup C_2) \), then \( (C_1 \cup C_2)' = C_1' \cup C_2' \subseteq cl' \).

Therefore, \( C_i' \subseteq cl' \), \( C_i \subseteq cl \).

Thus \( cl \in \text{Cimp}(C_i) \). (2) To see that \( cl \in \text{Cimp}(C_2) \).

Let \( I = \langle \Delta' \cdot \bullet' \rangle \) be the model of concept \( C_2 \), then \( I \) is also a model of \( C_1 \cup C_2 \). Since \( cl \in \text{Cimp}(C_1 \cup C_2) \), then \( (C_1 \cup C_2)' = C_1' \cup C_2' \subseteq cl' \).

Therefore, \( C_2' \subseteq cl' \), \( C_2 \subseteq cl \).

Thus \( cl \in \text{Cimp}(C_2) \). Therefore, \( cl \in \text{Cimp}(C_1 \cup C_2) \), and \( \text{Cimp}(C_1 \cup C_2) \subseteq \text{Cimp}(C_1) \cap \text{Cimp}(C_2) \).

(\( \Leftarrow \)) Assuming that \( cl \in \text{Cimp}(C_1) \cap \text{Cimp}(C_2) \), we proved that \( cl \in \text{Cimp}(C_1 \cup C_2) \). Let \( I = \langle \Delta' \cdot \bullet' \rangle \) be the model of \( C_i \cup C_2 \), then \( I \) is a model of \( C_1 \) or \( C_2 \). There are three cases:

1. \( I \) is a model of \( C_1 \), but not a model of \( C_2 \). Since \( cl \in \text{Cimp}(C_1) \), then \( C_1' \subseteq cl' \).

Therefore, \( (C_1 \cup C_2)' = C_1' \cup C_2' = C_1' \cup \emptyset = C_1' \subseteq cl' \).

2. \( I \) is a model of \( C_2 \), but not a model of \( C_1 \). Since \( cl \in \text{Cimp}(C_2) \), then \( C_2' \subseteq cl' \).

Therefore, \( (C_1 \cup C_2)' = C_1' \cup C_2' = C_1' \cup \emptyset = C_2' \subseteq cl' \).

3. \( I \) is a model of \( C_1 \), and also is a model of \( C_2 \). Since \( cl \in \text{Cimp}(C_1 \cup C_2) \), then \( C_1' \subseteq cl' \), \( C_2' \subseteq cl' \).

Therefore, \( (C_1 \cup C_2)' = C_1' \cup C_2' = \emptyset \cup C_1' = C_2' \subseteq cl' \).

\( \text{Cimp}(C_1 \cup C_2) = \text{Cimp}(C_1) \cap \text{Cimp}(C_2) \).

**Theorem 3.** Let \( cl \) be a clausal concept without the concept variable \( E \) (\( E=A \) or \( E=QR.L \)), and \( C \) be any ALC concept. Then \( cl \) is an implicate of \( C \) if \( cl \) is an implicate of \( C[\bot \cup E] \cup C[\top \cup E] \).

Proof. According to Theorem 1 and Theorem 2, it is obvious that this conclusion is correct.

According to Theorem 3, the set of implicates of \( C \) consists of three parts: (1) The first part is the set of implicates concept variables \( E, E=A \) or \( E=QR.L \); (2) The second part is the set of implicate concept variable \( \neg E, E=A \) or \( E=QR.L \); (3) The third part is the set of implicates of \( C[\bot \cup E] \cup C[\top \cup E] \), which neither contains concept variable \( E \) nor the concept variable \( \neg E, E=A \) or \( E=QR.L \).

Therefore, a concept implicate tree \( T \) can be regarded as a ternary tree, with each node having three subtrees except the leaf node. The first subtree is \( T_1 \), the second subtree is \( T_2 \), and the third subtree is \( T_3 \). Let \( N \) be a node labelling \( E_i \), and \( T_1, T_2, T_3 \) be the three subtrees of node \( N \). The root node of \( T_1 \) is labelled as \( E_{i+1} \), and \( T_1 \) contains the sets of implicates occurring \( E_{i+1} \). Moreover, the root node of \( T_2 \) is labelled as \( \neg E_{i+1} \), and \( T_2 \) contains the sets of implicates occurring \( \neg E_{i+1} \). Furthermore, the root node of \( T_3 \) is labelled as \( \top \), and \( T_3 \) contains the set of implicates not occurring \( E_{i+1} \) and \( \neg E_{i+1} \), which are the intersection of \( T_1 \) and \( T_2 \) irrespective of \( E_{i+1} \) and \( \neg E_{i+1} \).

Therefore, a method was proposed to build a concept implicate tree of a given concept. First, the structure of a node of the tree was defined as shown in Fig. (1), then the algorithms Simplify and BuildCIT were presented as shown in Figs. (2) and (3). Algorithm BuildCIT has four input parameters,
computing the intersection of the first two subtrees which is illustrated in Fig. (4).

Example 2. For the concept $C$ in example 1, the algorithm BuildCIT($C, V_{Con}, V_{Rol}, nil$) built a tree $T$ as follows:

First, a new CITnode $N'$ was built, which is root node of tree $T$, and $N'.label = \bot$, returning to BuildCIT($C, V_{Con}, V_{Rol}, N'$).

For BuildCIT($C, V_{Con}, V_{Rol}, N'$):
1) Let $N = N'$;
2) An atomic concept variable $A$ was selected;
3) $C_1 = $ Simplify($C[\bot / E]$) = ($A_1 \cup \forall R_1 . \forall A_2 \exists R_2 . A_3$);
   $C_2 = $ Simplify($C[\top / E]$) = $\neg A_2 \cup \forall R_2 . \exists R_2 . A_3$;
4) A new CIT node $N_1$ of tree $T$ was built with $N_1.label = A_1, N_1.first = N_1, call BuildCIT(C_1, A_2, V_{Rol}, N_1)$;

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 $N=\text{nil}$, 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
   $E = QR, l_x \in V_{rol}$;
   else, select the first atomic concept variable $E = A \in V_{con}$.
4. Let $C_1 = $ Simplify($C[\bot / E]$),
   $C_2 = $ Simplify($C[\top / E]$).
5. If $C_1 = \bot$, then
   build a new CITnode $N_1$ of tree $T$, and $N_1.label = E, N_1.first = N_1$.
6. If $C_2 = \bot$, then
   build a new CITnode $N_2$ of tree $T$, and $N_2.label = \neg E, N_2.leaf = \text{True}, N_2.second = N_2$.
7. If $C_1 = \top$, then $N_1.first = \text{nil}, N_1.third = \text{nil}$;
   else
   build a new CITnode $N_1$ of tree $T$, and $N_1.label = E, N_1.first = N_1$,
   if $V_{Con} = \emptyset$, then
   call BuildCIT($C_1, V_{Con}, E, V_{Rol}, N_1$);
   else
   call BuildCIT($C_1, V_{Con}, V_{Rol}, E, N_1$).
8. If $C_2 = \top$, then $N_2.second = \text{nil}, N_2.third = \text{nil}$;
   else
   build a new CITnode $N_2$ of tree $T$, and $N_2.label = \neg E, N_2.second = N_2$;
   if $V_{Con} = \emptyset$, then
   call BuildCIT($C_2, V_{Con}, E, V_{Rol}, N_2$);
   else
call BuildCIT($C_2, V_{Con}, V_{Rol}, E, N_2$).
9. If ($N_1.leaf$ and $N_2.leaf$), then
   delete node $N_1, N_2, and N.leaf = \text{True}, return T$.
10. If ($N_1.first \neq \text{nil}$ and $N_2.second \neq \text{nil}$), then
    build a new CITnode $N_3$ of tree $T$, and $N_3.third = N_3$,
call BuildThird($N_1.first, N_2.second, N_3$),
    and $N_3.label = \bot$.
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 $N_1.leaf = \text{true}$ and $N_2.leaf = \text{true}$,
   then $N_3.leaf = \text{true}, return T$.
3. If $N_1.leaf = \text{true}$,
   then $N_1.first = N_1.first, N_1.second = N_1.second, N_1.third = N_1.third, return T$.
4. If $N_2.leaf = \text{true},$
   then $N_1.first = N_1.first, N_1.second = N_1.second, N_1.third = N_1.third, return T$.
5. If $N_1.first = \text{nil}$ and $N_2.first = \text{nil}, then N_3.first = \text {nil};$
   else
   build a new CITnode $N_3$ of tree $T$,
   $N_3.first = N_3$,
call BuildThird($N_1.first, N_2.first, N_3$).
6. If $N_1.second = \text{nil}$ or $N_2.second = \text{nil},$
   then $N_3.second = \text{nil}$;
   else
   build a new CITnode $N_3$ of tree $T$,
   $N_3.second = N_3$,
call BuildThird($N_1.second, N_2.second, N_3$).
7. If $N_1.third = \text{nil}$ or $N_2.third = \text{nil},$
   then $N_3.third = \text{nil}$;
   else
   build a new CITnode $N_3$ of tree $T$,
   $N_3.third = N_3$,
call BuildThird($N_1.third, N_2.third, N_3$).
8. Return $T$.
5) A new CIT node $N_2$ of tree $T$ was built with $N_2.label = \neg A_1, N.second = N_2$, call BuildCIT($C_2, \{A_2\}, V_{Rot}, N_2$).

6) A new CITNode $N_3$ of tree $T$ was built with $N.third = N_3$, call BuildThird($N_1, N_2, N_3$), $N_3.label = \top$.

7) Returning to $T$.

In this algorithm, three algorithms are addressed, BuildCIT($C_1, \{A_1\}, V_{Rot}, N_1$), BuildCIT($C_2, \{A_2\}, V_{Rot}, N_2$), and BuildThird($N_1, N_2, N_3$). The algorithms BuildCIT($C_1, \{A_1\}, V_{Rot}, N_1$) and BuildCIT($C_2, \{A_2\}, V_{Rot}, N_2$) iterate the process of algorithm BuildCIT($C, V_{Con}, V_{Rot}, N'$) and build the first and second sub-trees of node $N'$. Algorithm BuildThird($N_1, N_2, N_3$) builds the third sub-tree of node $N'$. Finally, algorithm BuildCIT($C, V_{Con}, V_{Rot}, N'$) returns the concept implicite tree $T$ of $C$ as shown below.

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

Proof. The concept $C$ must be of the following four concepts: $\bot, \top, E$, or $\neg E$. 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, $C_T = \bot \sqcup E = E$, 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 $E$ and the second sub-node labelling $\neg E$. Thus, $C_T = \bot \sqcup \neg E = \neg E$, and $C_T$ is logically equivalent to $C$. Therefore, $C_T$ is logically equivalent to $C$, and is one of the concepts among $\bot, \top, E$, or $\neg E$.

**Theorem 5.** Let $C$ be an ALC concept, $V_{Con}$ be an atomic concept variable set of $C$, $V_{Rot}$ be a role concept variable set of $C$, $T$ be a tree of $C$ built by the algorithm BuildCIT, and $C_T$ 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) First the logic equivalence was verified by induction on the number $m$ of concept variables in $C$, let $V = V_{Con} \sqcup V_{Rot} = \{E_1, \ldots, E_m\}, E_i = A_i \in V_{Con}$ or $E_i = Q R_i L_i \in V_{Rot}, 1 \leq k < l \leq m, Q \in \{\forall, \exists\}$.

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

   <2> Inductive hypothesis: It was assumed that the theorem was true for all concepts with almost all $m$ concept variables.

   <3> Induction: It was assumed that $C$ had $m + 1$ concept variables. Let $E_i$ be any concept variable of $V$, $E_i \in V_{Con}$ or $E_i \in V_{Rot}, 1 \leq i \leq m$, now assuming that $E_i$ is an atomic concept variable form $V_{Con}$, then, it must be proved that:

   $$C \equiv C_T = (E_1 \sqcup C_{BuildCIT}(\bot / E_1, V_{Con}, V_{Rot}, N_1)) \cap (\neg E_1 \sqcup C_{BuildCIT}(\top / E_1, V_{Con}, V_{Rot}, N_1)) \cap (C_{BuildCIT}(\bot / E_i, V_{Con}, V_{Rot}, N_1)) \cap (\neg E_i \sqcup C_{BuildCIT}(\bot / E_i, V_{Con}, V_{Rot}, N_1)) \cap (C_{BuildCIT}(\bot / E_i, V_{Con}, V_{Rot}, N_1)).$$

   According to the inductive hypothesis, we obtain,

   $$C[\bot / E_i] \equiv C_{BuildCIT}(\bot / E_i, V_{Con}, V_{Rot}, N_1),$$

   $$C[\top / E_i] \equiv C_{BuildCIT}(\top / E_i, V_{Con}, V_{Rot}, N_1),$$

   $$C[\bot / E_i] \cap C[\top / E_i].$$

   So,

   $$C_T = (E_1 \sqcup C[\bot / E_1]) \cap (\neg E_1 \sqcup C[\top / E_1]) \cap (C[\bot / E_i]) \cap (\neg E_i \sqcup C[\bot / E_i]).$$

   Let $I = \langle \Delta', \bullet \rangle$ be any model of concept $C$, so $C' = \emptyset$, and there exists an individual $a$ such that $a \in C'$. Following are the two cases of the individual $a$.

   Case 1, supposing $a \in E_i'$, hence, $a \in (C[\bot / E_i])'$, and $a \in (C[\bot / E_i] \sqcup C[\top / E_i])'$. Therefore, $a \in (C_T)'$. Thus, $C_T' \subseteq (C_T)'$, and $C \subseteq C_T$.

   Case 2, supposing $a \notin E_i'$, hence $a \in (\neg E_i)'$, $a \in (C[\bot / E_i])'$, and $a \in (C[\bot / E_i] \sqcup C[\top / E_i])'$. Therefore, $a \in (C_T)'$. Thus, $C_T' \subseteq (C_T)'$, and $C \subseteq C_T$.

   Therefore, $C \subseteq C_T$, suggesting that $C_T \subseteq C$ is similar. Hence, $C \equiv C_T$, that is to say, $C_T$ is logically equivalent to $C$.

   (2) It was shown that each branch of $T$ is an implicate of $C$. According to the distributive laws of description logic that is similar to the 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, each branch of $T$ is an implicate of $C$.  

Zou and Deng
Theorem 6. Let $C$ be the ALC concept, $V_{Con}$ be an atomic concept variable set of $C$, and $V_{Rot}$ be a role concept variable set of $C$. Then algorithm BuildCIT is valid and complete.

Proof. (1) First, the validity of the algorithm was proved. The algorithm BuildCIT built a tree $T$, and according to theorem 5, each branch of $T$ was an implicate of $C$, thereby making $T$ a concept implicate tree of $C$. Thus, the algorithm BuildCIT was proved to be valid.

(2) Now, the complete algorithm is explained below. For any concept $C$, the algorithm BuildCIT can build a corresponding tree $T$, and there does not exist a concept without the corresponding tree. Thus, the algorithm BuildCIT is complete.

4. TRACTABLE QUERYING

In this section, for any concept represented by a concept implicate tree, the queries are computable in the linear time depending on 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 as $\perp$, then $C$ is characterized with unsatisfiability, otherwise with satisfiability. With regard to the tautology-testing, if $T$ contains only one node that is labelled as $T$, then $C$ has tautology, otherwise $C$ has no tautology. Obviously, these two queries can be addressed in the linear time.

In order to test subsumption between the two concepts, the paper provides some theorems as follows.

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

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

Proof. Let $I = < \Delta^i, \bullet^i >$ be any model of concept $C \{L / L\}$, then $(C \{L / L\})^i = \emptyset$. Now, $I$ is extended to $I' = < \Delta', \bullet' >$ by setting $\Delta' = \Delta^i$, $L' = \emptyset$, then $I'$ is a model of concept $C$. Therefore, $(C \{L / L\})^i = C^i$. Since $cl$ is an implicate of concept $C$, hence, $C^i \subseteq cl^i$. Moreover, $cl^i = (cl / \{L\})^i$. Thus, $(C \{L / L\})^i \subseteq (cl / \{L\})^i$, and $(C \{L / L\}) \supseteq (cl / \{L\})$. Therefore, $(cl / \{L\}) \in \text{Cimp}(C \{L / L\})$.

Theorem 8. Let $C$ be an ALC concept, $V_{Con}$ be an atomic concept variable set of $C$, $V_{Rot}$ be a role concept variable set of $C$, $T$ be a tree of $C$ built by the algorithm BuildCIT, and $cl$ be an implicate of $C$. Then there is a unique prefix of $cl$ that is a branch of $T$.

Proof. It was proved that there is a unique prefix of $cl$, which is a branch of $T$. By induction on the number $m$ of concept variables in $C$, let $V = V_{Con} \cup V_{Rot} = \{E_1, \ldots, E_m\}$, $E_k = A_k \in V_{Con}$ or $E_i = QR, L_i \in V_{Rot}$, $1 \leq k < l \leq m$.

1) Base case: Theorem 4 considers the case $m=1$.

2) Inductive hypothesis: It was assumed that the theorem was true for all concepts with almost all $m$ concept variables.

3) Induction: Assuming 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 $d_1 < d_2 < \cdots < d_s$.

Therefore, it must be proved that there is a unique prefix of $cl$ that is a branch of $T$. Let $E_i$ be any concept variable of $V$, $E_i \in V_{Con}$ or $E_i \in V_{Rot}$, $1 \leq i \leq m$, now assuming that $E_1$ is an atomic concept variable of the form $V_C$, then it must be proved that there is a unique prefix of $cl$ that is a branch of $C_T$, which is the concept

$$C_T = (E_1 \cup C_{\text{BuildCIT}}(C \{L / E_1\} \cup V_{Con} - \{E_1\} \cup V_{Rot} - \{E_1\}))$$

Then there is a unique prefix of $cl$ that is a branch of $T$. Let $E_i$ be any concept variable of $V$, $E_i \in V_{Con}$ or $E_i \in V_{Rot}$, $1 \leq i \leq m$, now assuming that $E_1$ is an atomic concept variable of the form $V_C$, then it must be proved that there is a unique prefix of $cl$ that is a branch of $C_T$, which is the concept

$$C_T = (E_1 \cup C_{\text{BuildCIT}}(C \{L / E_1\} \cup V_{Con} - \{E_1\} \cup V_{Rot} - \{E_1\}))$$

By the inductive hypothesis, there is a unique prefix of $cl$ that is a branch of the intersection of two subtrees $C_{\text{BuildCIT}}(C \{L / E_1\} \cup V_{Con} - \{E_1\} \cup V_{Rot} - \{E_1\})$ and $C_{\text{BuildCIT}}(C \{L / E_1\} \cup V_{Con} - \{E_1\} \cup V_{Rot} - \{E_1\})$. In this case, the theorem is true for the third branch of $C_T$. Moreover, if $d_i > 1$, then nothing is needed to prove, therefore, assuming $d_i = 1$. $L_1$ is either $E_1$ or $\neg E_1$; these are the two cases.

Case 1: Assuming that $L_1 = E_1$, then according to Theorem 7, $cl / \{E_1\}$ is an implicate of $C \{L / E_1\}$. Moreover, by the inductive hypothesis, there is a unique prefix $G$ of $cl / \{E_1\}$ that is a branch of $C_{\text{BuildCIT}}(C \{L / E_1\} \cup V_{Con} - \{E_1\} \cup V_{Rot} - \{E_1\})$. Therefore, $H$ is a prefix of $cl$ that is a branch of $T$. Now, it is to be proved that $H$ is a unique prefix of $cl$ by contradiction. Assuming 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 $cl / \{E_1\}$ that is a branch of $C_{\text{BuildCIT}}(C \{L / E_1\} \cup V_{Con} - \{E_1\} \cup V_{Rot} - \{E_1\})$. However, $G$ is a unique prefix of $cl / \{E_1\}$ by the inductive hypothesis, therefore, $G' = G$, $H' = H$. Therefore, $H$ is a unique prefix of $cl$ that is a branch of $T$.

Case 2: Assuming that $L_1 = \neg E_1$, then according to Theorem 7, $cl / \{-E_1\}$ is an implicate of $C \{L / \neg E_1\}$ that is equiva-
lent to \( C[\top / E_i] \). Moreover, by the inductive hypothesis, there is a unique prefix \( G \) of \( cl/\{\neg E_i\} \) that is a branch of 
\[ \text{BuildCTT}(C[\top / E_i], V_{con} - \{E_i\}, V_{rol}, N_i). \] 
Therefore, 
\[ H = \neg E_i \cup G \] is a prefix of \( cl \) that is a branch of \( T \). Now, it is
to be proved that \( H \) is a unique prefix of \( cl \) by contradiction. Assuming that \( H' \) is another prefix of \( cl \) that is a branch of \( T \), let \( H' = \neg E_i \cup G' \), then \( G' \) is a prefix of 
\[ \text{BuildCTT}(C[\top / E_i], V_{con} - \{E_i\}, V_{rol}, N_i). \] 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, \( V_{con} \) be an atomic concept variable set of \( C \), \( V_{rol} \) be a role concept variable set of \( C \), and \( T \) 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 is unique. Thus, it is obvious that the conclusion holds true.

In example 2, all prime implicates of concept \( C \) are
\[ A_1 \cup A_2 \cup \forall R_2 \neg A_4, A_1 \leq \forall R_2 \exists R_2 . A_4, \]
\[ \neg A_2 \cup \forall R_2 . \exists R_2 . A_5 . \]

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

**Theorem 10.** Let \( C \) be an ALC concept, \( V_{con} \) be an atomic concept variable set of \( C \), \( V_{rol} \) be a role concept variable set of \( C \), and \( T \) be a concept implicate tree of \( C \). Then every subsuming 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 true.

In example 2, an implicate \( A_1 \cup A_2 \cup \forall R_2 \neg A_4 \) contains the literal \( \forall R_2. \neg A_4 \), which is a label of the leaf of a branch of \( T \). Moreover, all other implicates of \( C \) in \( T \), for example are,
\[ A_1 \cup A_2 \cup \exists R_2 . A_4 \cup \forall R_2 . \exists R_2 . A_5, \]
\[ A_1 \cup A_2 \cup \forall R_2 . \exists R_2 . A_5, \]
\[ A_1 \cup \neg A_4 \cup \forall R_2 \neg A_4 \cup \forall R_2 . \exists R_2 . A_4, \]
\[ A_1 \cup \neg A_4 \cup \exists R_2 . A_4 \cup \forall R_2 . \exists R_2 . A_4, \]
\[ A_1 \cup \neg A_4 \cup \forall R_2 . \exists R_2 . A_5, \]
\[ A_1 \cup \forall R_2 . \neg A_4 \cup \forall R_2 . \exists R_2 . A_4, \]
\[ A_1 \cup \exists R_2 . A_4 \cup \forall R_2 . \exists R_2 . A_4, \]
\[ A_1 \cup \exists R_2 . A_4 \cup \forall R_2 . \exists R_2 . A_5, \]

and \( \neg A_4 \cup \forall R_2 . \exists R_2 . A_4 \), contain the literal \( \forall R_2 . \exists R_2 . A_4 \), which is a label of the leaf of a branch of \( T \).

Based on the above theorems, if clausal concept \( cl = L_1 \cup L_2 \cup \cdots \cup L_s \) is an implicate of concept \( C \), and \( T \) is a concept implicate tree of \( C \), then there is a unique prefix \( cl' = L_1 \cup L_2 \cup \cdots \cup L_s \) of \( cl \) that is a branch of \( T \), \( 1 \leq i \leq s \), and each literal \( L_i \) is a label of that branch, \( 1 \leq i \leq s \), and \( L_i \) is a label of the leaf node. Therefore, the study presents the algorithm Subsume as shown in Fig. (5). The main idea is to
determine whether \( cl \) is an implicate of \( C \) if there exists a branch that 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 depending on the size of the query, but not on the size of \( T \). This is an important property of the proposed method.

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

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 literals in \( cl \) to decide whether \( C \subseteq cl \). Therefore, determining whether \( C \subseteq cl \) can be done in linear time in \( |cl| \).
CONCLUSION

In this paper, knowledge compilation for description logic was presented based on the concept implicate tree. Firstly, the concept implicate tree was defined for the ALC concept. Moreover, the study also provided an algorithm to translate the arbitrary ALC concept into an equivalent concept implicate tree. Finally, it was proved that satisfiability-testing, autology-testing and subsumption-testing were computable in linear time with respect to the concept implicate tree. It was concluded that any query can be done in linear time based on the size of the query, regardless of the size of the concept implicate tree. In other words, the proposed 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