Formal verification of a proof procedure for the description logic $\mathcal{ALC}$

Mohamed Chaabani  Mohamed Mezghiche
LIMOSE, University of Boumerdes
Boumerdes, Algeria
chaabani@umbb.dz  mohamed.mezghiche@yahoo.fr

Martin Strecker
IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse
strecker@irit.fr

Description Logics (DLs) are a family of languages used for the representation and for reasoning about the knowledge base of an application domain, in a structured and formal manner. To achieve this objective, several provers have been implemented such as RACER and FACT++, but these provers themselves have not been certified. In order to insure the soundness of derivations in these DLs, it is necessary to verify formally the deductions applied by these reasoners. Formal Methods offer powerful tools for the specification and verification of proof procedures, among them methods of proving properties such as soundness, completeness and termination of a proof procedure.

In this paper, we present the definition of a proof procedure for the Description Logic $\mathcal{ALC}$, based on a tableau method. We then prove the soundness, completeness and termination of our reasoner with the proof assistant Isabelle. The proof proceeds in two phases, by first establishing these properties on an abstract, set theoretic level, and by instantiating them with an implementation based on lists.

1 Introduction

In this paper, we present a definition of a prover for the description logic $\mathcal{ALC}$ [18] which is based on the method of semantic tableau [5]. We ensure the validity of our method and its implementation by the proof of the properties of its soundness, its completeness and its termination. These proofs are performed using the Isabelle/HOL proof assistant.

Description Logics (DL) are formalisms widely used in several areas such as the Semantic Web and ontology construction. Several description logics, eg, $\mathcal{SHOIQ}$ [12] and $\mathcal{SHOIN}$ [3] are more expressive extensions of $\mathcal{ALC}$. $\mathcal{SHOIN}$, which is considered as a formal model of Web Ontology Language OWL-DL, is used in several provers such as FaCT++ [19] and RACER [10].

Only a formal proof of the validity of the reasoning process applied to DLs can ensure the correctness of derivations of properties in these logics. This is far from being the case for provers available today. The inference engines of provers like RACER and FaCT++ have not yet been certified. In [13], the authors give a detailed description of the various problems posed by incomplete and incorrect provers yet widely used.

Of the many methods used as decision procedures for DLs, the semantic tableau method is the most common. Indeed, FaCT and RACER use it in their reasoning process. We propose in this paper a definition of a prover based on a semantic tableau method for the description logic $\mathcal{ALC}$. We will also present the formal proof in the Isabelle/HOL proof assistant of soundness, completeness and termination of the prover.

Adel Bouhoula, Tetsuo Ida and Fairouz Kamareddine (Eds.): Symbolic Computation in Software Science 2012 (SCSS2012)
EPTCS 122, 2013, pp. 51–61  doi:10.4204/EPTCS.122.5
Formal verification of a proof procedure for the description logic $\mathcal{ALC}$

Work close to our formalization of proof procedures are reported in [16], [20] and [17]. The first describes the formalization of a prover for first-order logic in the Isabelle/HOL proof assistant. The second presents a formalization in the Coq proof assistant of some modal logics, and the third a formalization of a prover for LTL. Please note that DLs can be considered as specific modal logics. Closest to our work comes [11], which describes the formalization of $\mathcal{ALC}$ in the PVS proof assistant. Our formalization simplifies the termination argument, see Section 8, and allows to extract a prover directly executable in a “standard” programming language (Caml). This paper builds on and extends our previous work described in [7], which provides an abstract, non-executable proof procedure for $\mathcal{ALC}$, that extension is presented in [8].

This paper is organized as follows: In Sections 2, 3 and 4 we detail the description logics. In Section 5 we present the procedure of semantic tableau for $\mathcal{ALC}$ and the Sections 6 and 7 we describe the formalization of this method and present the proof of soundness and completeness properties in the Isabelle/HOL proof assistant. In the rest of the article, we present an implementation of a method of proof for the logic $\mathcal{ALC}$ and the proof of its soundness and its termination.

The development described here was carried out in the environment of the Isabelle/HOL proof assistant whose logic HOL is a classical logic. However, the background of the development is largely independent of Isabelle and could easily be simulated in other proof assistants.

2 Description logics

Description Logics [1, 2, 4, 15] are a family of knowledge representation languages which can be used to represent knowledge of an application domain in a structured and formal way. A fundamental characteristic of these languages is that they have a formal semantics. Description logics are used for various applications. Among them are: The representation of ontologies [3], natural language processing [9] and representation of the semantics of UML class diagrams [6].

We recall that description logics have as a common basis $\mathcal{AL}$ enriched with different extensions: The description logic $\mathcal{ALC}$, subject of this work, adds negation to $\mathcal{AL}$. Other extensions add the transitive closure of roles, number restrictions on roles, the notion of sub-roles etc.. The formulas $C$ of $\mathcal{ALC}$ logic, called concepts, are constructed inductively by the following grammar:

$$C ::= A \quad (\text{atomic concept})$$
$$| \top \quad (\text{universal concept Top})$$
$$| \bot \quad (\text{empty concept Bottom})$$
$$| \neg C \quad (\text{negation})$$
$$| C \land C \quad (\text{conjunction})$$
$$| C \lor C \quad (\text{disjunction})$$
$$| \forall r. C \quad (\text{universal quantifier})$$
$$| \exists r. C \quad (\text{existential quantifier})$$

Here, $A \in NC$ is an atomic concept name, and $r \in NR$ is a role name. A role is a binary relation between instances of a concept. $\forall$ and $\exists$ are (multi-)modal operators, similar to $\Box$ and $\Diamond$ in traditional modal logics.

3 Syntax of $\mathcal{ALC}$

We now give details of the formal definition of the logic $\mathcal{ALC}$. The type of roles, defined by:

---

1http://www.irit.fr/CMS-DRUPAL7/AcadieProjects/dlverified
datatype 'nr role = AtomR 'nr

It only has a single constructor, but is easily expandable to accommodate more complex logic. Type
definitions are parameterized by the type of role names 'nr and atomic concepts 'nc. Following the
grammar of Section 2, here is the type definition of $\mathcal{A} LC$-concepts:

datatype ('nr, 'nc) concept =
  AtomC 'nc |
  Top |
  Bottom |
  NotC (('nr, 'nc) concept) (('nr, 'nc) concept) |
  AndC (('nr, 'nc) concept) (('nr, 'nc) concept) |
  OrC (('nr, 'nc) concept) (('nr, 'nc) concept) |
  AllC ('nr role) (('nr, 'nc) concept) |
  SomeC ('nr role) (('nr, 'nc) concept)

4 Semantic of $\mathcal{A} LC$

Concepts are interpreted as subsets of a domain of interpretation $\Delta$ and roles as subsets of the product
$\Delta \times \Delta$. An interpretation $\mathcal{I}$ is essentially a couple $(\Delta, \mathcal{I})$ where $\Delta$ is called the domain of
interpretation and $\mathcal{I}$ is an interpretation function that maps an atomic concept $A$ to subset $A^{\mathcal{I}}$ of $\Delta$ and
a role $r$ to subset $r^{\mathcal{I}}$ of $\Delta \times \Delta$. Its extension to other concept constructors is defined, in mathematical
notation, as follows:

$\top^{\mathcal{I}} = \Delta$
$\bot^{\mathcal{I}} = \emptyset$
$(C \cap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$(C \cup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$(-C)^{\mathcal{I}} = \Delta - C^{\mathcal{I}}$
$(\forall r.C)^{\mathcal{I}} = \{ x \in \Delta \mid \forall y : (x,y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}} \}$
$(\exists r.C)^{\mathcal{I}} = \{ x \in \Delta \mid \exists y : (x,y) \in r^{\mathcal{I}} \land y \in C^{\mathcal{I}} \}$

The type $\text{domtype}$ is the type of elements of the interpretation domain. Then The interpretation is
defined as follows:

record ('ni, 'nr, 'nc) Interp =
  idomain :: domtype set
  interp-c :: 'nc \Rightarrow domtype set
  interp-r :: 'nr \Rightarrow (domtype * domtype) set
  interp-i :: 'ni \Rightarrow domtype

The interpretation of roles in Isabelle is given by:

fun interpR :: ('ni, 'nr, 'nc) Interp \Rightarrow 'nr role \Rightarrow (domtype * domtype) set where
  interpR i (AtomR b) = (interp-r i) b

The interpretation of concepts is described by the function:

fun interpC :: ('ni, 'nr, 'nc) Interp \Rightarrow ('nr, 'nc) concept \Rightarrow domtype set where
  interpC i Bottom = {}
  interpC i Top = UNIV
  interpC i (AtomC a) = interp-c i a
  interpC i (AndC c1 c2) = (interpC i c1) \cap (interpC i c2)
  interpC i (OrC c1 c2) = (interpC i c1) \cup (interpC i c2)
Formal verification of a proof procedure for the description logic $\mathcal{ALC}$ is one of the goals of the Section 8. Does not indicate which instance of a conjunction rule will be applied. Making the calculation more homogeneous format is useful for writing and verifying tactics. For example, we show the rule for the constructor $\Andrule$:

\begin{align*}
\text{definition } \text{Andrule} :: (\text{'}ni, \text{'}nr, \text{'}nc) \text{ rule where} \\
\text{mk-andrule: } & \{\text{Inst } x \text{ (AndC } c1 \text{ } c2) \in b1; - ((\text{Inst } x \text{ } c1) \in b1 \land (\text{Inst } x \text{ } c2) \in b1)\}; \\
& b2 = \{\text{Inst } x \text{ } c2\} \cup \{\text{Inst } x \text{ } c1\} \cup b1 \implies \text{Andrule } b1 \text{ } b2
\end{align*}

It expresses that an instance of the concept $(\text{AndC } c1 \text{ } c2)]$ must be located in the $\text{Abox}$ before applying the rule (this is the condition of applicability), the concept has not been decomposed, and the application of the rule adds the sub-concepts $c1$ and $c2$. Of course, this rule is highly non-deterministic, since it does not indicate which instance of a conjunction rule will be applied. Making the calculation more deterministic is one of the goals of the Section 8.
\[\begin{array}{|c|c|c|c|}
\hline
\text{Rule} & \text{Condition} & \text{Negative Appl Cond} & \text{Action} \\
\hline
\rightarrow \land & x : C_1 \cap C_2 \in \mathcal{A} & x : C_1 \text{ and } x : C_2 \text{ are not both in } \mathcal{A} & \mathcal{A} := \mathcal{A} \cup \{x : C_1, x : C_2\} \\
\rightarrow \lor & x : C_1 \lor C_2 \in \mathcal{A} & \text{neither } x : C_1 \text{ nor } x : C_2 \text{ in } \mathcal{A} & \mathcal{A} := \mathcal{A} \cup \{x : C_1\} \text{ or } \mathcal{A} := \mathcal{A} \cup \{x : C_2\} \\
\rightarrow \forall & x : \forall r \in C \in \mathcal{A} & r x y \in \mathcal{A} \text{ but } y : C \notin \mathcal{A} & \mathcal{A} := \mathcal{A} \cup \{y : C\} \\
\rightarrow \exists & x : \exists r \in C \in \mathcal{A} & \neg \exists y \text{ such that } r x y \text{ and } y : C \text{ are both in } \mathcal{A} & \mathcal{A} := \mathcal{A} \cup \{z : C, r x z\} \\
\hline
\end{array}\]

Table 1: The decomposition rules for the method of semantic tableaux for \(\mathcal{A} L^C\)

For space reasons, we cannot present all the rules in detail. They are reproduced in Table 1. The constructor Some\(r\) requires special attention. As indicated in Table 1, the application of rule requires the use of a new variable. What first comes to mind is to postulate the existence of this variable with an existential quantifier in the precondition of the rule. However, this non-deterministic existential choice would be impossible to implement by any specific generator function, or lead to a very complex notion of correspondence of abstract and implemented states. We therefore parameterize the rule with the generator function gen, which is also used in the implementation (see Section 8).

**inductive Somerule-gen::** \(((\text{'ni', \text{'nr'}, \text{'nc')) abox} \Rightarrow \text{'ni}) => (\text{'ni', \text{'nr'}, \text{'nc}) abox} \Rightarrow (\text{'ni', \text{'nr'}, \text{'nc}) abox} \Rightarrow \text{bool} \text{ where}
\mk-Somerule-gen::[(\text{Inst x (SomeC r c l)}) \in b1]; \forall y. \neg((\text{Rel r x y}) \in b1 \land (\text{Inst y c l}) \in b1); z := \text{gen b1}; b2 := \text{insert (Rel r x z)} (\text{insert (Inst z c l)} b1)]
\Rightarrow \text{Somerule-gen gen b1 b2}

In summary, our rules are:

**definition list-alc-rules::**\(((\text{'ni', \text{'nr'}, \text{'nc}) abox}) \Rightarrow \text{'ni}) => ((\text{'ni', \text{'nr'}, \text{'nc}) rule) list
**where** list-alc-rules gen = [Andrule, Orrule, Allrule, Somerule-gen gen]

From these elementary rules, we can construct composite rules by application of rule constructors, such as the following:

**fun disj-rule ::** (\text{'ni', \text{'nr'}, \text{'nc}) rule \Rightarrow (\text{'ni', \text{'nr'}, \text{'nc}) rule \Rightarrow (\text{'ni', \text{'nr'}, \text{'nc}) rule
**where** disj-rule r1 r2 = (\lambda a b. r1 a b \lor r2 a b)

It allows to define by recursion the function disj-rule-list that converts a list of rules in a rule. Finally, we define the rule

**definition alc-rule ::**\(((\text{'ni', \text{'nr'}, \text{'nc}) abox}) \Rightarrow \text{'ni}) \Rightarrow (\text{'ni', \text{'nr'}, \text{'nc}) rule
**where** alc-rule gen = disj-rule-list (list-alc-rules gen)

## 6 Soundness

The first central property of a system of rules is soundness. A rule is called sound, if its conclusion is satisfiable then its premise is satisfiable:

**definition sound ::** (\text{'ni', \text{'nr'}, \text{'nc}) rule \Rightarrow \text{bool
**where** sound r \equiv \forall A1 A2. r A1 A2 \Rightarrow \text{satisfiable-abox A2} \Rightarrow \text{satisfiable-abox A1}

It is easy to show that the elementary rules preserve soundness, and the disjunction of rules:

**lemma disj-rule-sound [simp]:** sound r1 \Rightarrow sound r2 \Rightarrow sound (disj-rule r1 r2)
or the transitive closure:

**Lemma tranclp-rule-sound [simp]:** \(\text{sound } r \implies \text{sound } (r^+)^+\)

Also the proof of soundness of the various rules offers no surprises:

**Lemma alcrule-sound [simp]:** \(\text{sound } (\text{alc-rule gen})\)

### 7 Completeness

To prove completeness, we first define the notion of complete rule. A rule is complete if the satisfiability of Abox A1 implies that there exists at least one satisfiable Abox A2 obtained by rule application from A1.

**Definition complete ::\(\text{('ni, 'nr, 'nc) rule } \Rightarrow \text{bool}\)**

where complete \(r \equiv \forall A1. \exists A2. \text{satisfiable-abox } A1 \rightarrow (r A1 A2) \rightarrow \text{satisfiable-abox } A2\)

We can show this property for each rule. For the rule \(\rightarrow \top\), we obtain:

**Lemma and-complete [simp]:** complete Andrule

An Abox is contradictory if it contains a contradiction (clash), i.e., \(x : C\) and \(x : \neg C\) or \(x : \bot\).

**Function contains-clash ::\(\text{('ni, 'nr, 'nc) aabox } \Rightarrow \text{bool}\)**

where contains-clash AB = 
\[\exists x. ((\text{Inst } x. c) \in AB \land (\text{Inst } x. (\neg C c)) \in AB) \lor ((\text{Inst } x. \bot) \in AB)\]

The fundamental property of the correctness of the tableau algorithm is that if the Abox is closed (contains a clash) then it is unsatisfiable:

**Lemma content-clash-not-satisfiable:[contains-clash AB; satisfiable-abox AB] \rightarrow \text{False}**

An Abox is saturated for a rule if the rule is not applicable to it.

**Definition saturated ::\(\text{('ni, 'nr, 'nc) aabox } \Rightarrow \text{('ni, 'nr, 'nc) rule } \Rightarrow \text{bool}\)**

where saturated AB1 r \(\equiv (\forall AB2, \neg (r AB1 AB2))\)

Finally, if an Abox A is saturated and not contradictory, then it is satisfiable. In this case, there is an interpretation that satisfies A, which is called the canonical interpretation \(\mathcal{I}_A\), whose components are defined as follows:

1. The interpretation domain \(\Delta_{\mathcal{I}_A}\) is the set of all individuals included in A
2. For each concept name \(P\) we define \(P_{\mathcal{I}_A} = \{x | (x : P) \in A\}\)
3. For each role name \(r\) we define \(r_{\mathcal{I}_A} = \{(x, y) | r(x, y) \in A\}\)

Now the goal is to prove that if an Abox is not contradictory and saturated, then it is satisfiable by the canonical interpretation.

**Lemma canon-interp-sat-fact:[\text{inj i; } \neg \text{contains-clash AB}; \text{saturated AB } (\text{alc-rule gen}); \text{is-Normal-Abox AB}; f \in AB] \implies \text{satisfies-fact } (\text{canon-interp } i AB) f**
8 Implementation

In this section, we propose an implementation, i.e. an executable proof procedure for the description logic of $L^C$. It is based on lists as data structure to implement the $Abox$. The tableau is encoded as a list of $Abox$. The different rules are defined as functions whose argument is list $abox-impl$ and return a list of $Abox$ (Tableau).

The $Abox$ is implemented as a list of facts ($abox_impl$).

\textbf{type-synonym} $'(ni, nr, nc) abox-impl = ((ni, nr, nc) fact) list$

The implementation of a rule is defined as a function that transforms an $Abox_impl$ to a list of $Abox_impl$, just called a tableau.

\textbf{type-synonym} $'(ni, nr, nc) rule-impl=$

$'(ni, nr, nc) abox-impl \Rightarrow (ni, nr, nc) abox-impl list$

The type of abstraction of an ($Abox_impl)$ to an $Abox$ is given by:

\textbf{type-synonym} $'(ni, nr, nc) abstraction= (ni, nr, nc) abox-impl \Rightarrow (ni, nr, nc) abox$

Tableau is simply a list of $Abox_impl$.

\textbf{type-synonym} $'(ni, nr, nc) tableau = (ni, nr, nc) abox-impl list$

8.1 Implementing Rules

Each rule is encoded as a pair consisting of the condition of applicability and the given action ($Condition$, $Action$). The type of condition is given by:

\textbf{type-synonym} $'(ni, nr, nc) appcond= (ni, nr, nc) abox-impl \Rightarrow (ni, nr, nc) fact ⇒ bool$

The type of the action of a rule is defined by:

\textbf{type-synonym} $'(ni, nr, nc) action =$

$(ni, nr, nc) abox-impl * (ni, nr, nc) fact * (ni, nr, nc) abox-impl$

$\Rightarrow (ni, nr, nc) abox-impl list$

The rule is implemented in Isabelle by the function:

\textbf{datatype} $(ni, nr, nc) srule = Rule (ni, nr, nc) appcond * (ni, nr, nc) action$

Once the data structures and format rules are defined, we can encode each rule. For this, we determine for each rule the condition of its applicability and the action of this rule. For example the rule $\rightarrow_{\rightarrow}$ is coded as follows: The condition of applicability of the rule $\rightarrow_{\rightarrow}$ is given by the following function:

\textbf{fun} appcond-and :: $(ni, nr, nc) appcond

\textbf{where} appcond-and Ab-i (Inst x (AndC c1 c2)) =

$(\neg \text{list-ex (is-x-c-inst x c1) Ab-i}) \lor \neg \text{list-ex (is-x-c-inst x c2) Ab-i})$

| appcond-and Ab-i - = False

The function $\text{is-x-c-inst}$ means: $\text{is-x-c-inst x c f} = (f = \text{Inst x c})$

The action provided by the application of this rule is:

\textbf{fun} action-and :: $(ni, nr, nc) action

\textbf{where} action-and (prefix, (Inst x (AndC c1 c2)), suffix) =

$[[\text{Inst x c1}, \text{Inst x c2}] @ \text{prefix} @ [[\text{Inst x (AndC c1 c2)}] @ \text{suffix}]$

| action-and - = []
prefix denotes the elements of the list before the fact Inst x (AndC c1 c2) and suffix denotes the elements after the fact. The rule → is implemented by:

definition and-srule :: ('ni,'nr,'nc)rule
  where and-srule == Rule (appcond-and,action-and)

Its application is defined in Isabelle by:

definition and-rule::('ni,'nr,'nc) rule-impl where and-rule ≡ apply-srule and-srule

To generalize our implementation for all $AL^C$ rules, we define a list of implemented rules.

definition list-alc-rules-impl-gen ::
  (((ni,nr,nc)abox-impl) ⇒ (ni:: alloc, nr, nc) rule-impl list)
list-alc-rules-impl-gen gen == [and-rule, or-rule, all-rule, some-rule-gen gen]

8.2 Soundness

We have shown in section 6 the proof of soundness property of rules on the abstract level. The following result allows us to demonstrate the soundness of our implementation.

The idea can be illustrated by the following diagram. Above, we see the proof on an abstract level: the application of an abstract rule (r-logic), applied to an Abox, generates a set of Abox successors. The implementation of Abox by lists gives a Abox_impl, whose abstraction with a set gives a set of facts. The application of a rule for this implementation (r-impl) must provide a list of Abox implementations that have the same abstraction.

More formally, the definition in Isabelle:

definition sound-rule-impl::
  (ni,nr,nc) abstraction⇒(ni,nr,nc) rule⇒(ni,nr,nc) rule-impl⇒ bool where
sound-rule-impl abstr r r-impl⇒ ∀ ai ai', (ai' ∈ set(r-impl ai)) −→ r(abstr ai)(abstr ai')

8.3 Termination

Termination is an important property of rewriting systems. A standard method for proving termination of a rewrite system is to exhibit a well-founded ordering on terms, such that if $A_1$ is rewritten to $A_2$ then $A_1 \gg A_2$.

Formally, for proving termination, we associate to each implementation of a Abox a measure. If this measure decreases with every rule in a well-founded ordering, termination is assured. In our case, a measure is a function of type Abox_impl → T for T a domain equipped with a well founded relation $\gg$ in T which we define in the sequel.

We introduce the constructors necessary to define the measure:

The function sizeC calculates the size of a concept, it is defined as the number of constructors of this concept.

• The measure in our case is a multiset of pairs of natural numbers.
• For each axiom (fact) of Abox\_imp, we associate a pair, depending on the structure of the axiom and the Abox\_imp.

• The measure of the element that is applicable must decrease, without affecting the measure of other elements.

We now define the measure of the axioms (this is the function meas\_comp defined in the following). If the axiom is:

• A relation \(x r y\), we associate the pair \((0,0)\)

• An instance \(x : D\). Depending on the structure of \(D\), there are three cases:
  1. If \(D\) is an Atom \((x : A)\) or Negation \((x : \neg A)\), we associate the value \((0,0)\);
  2. If \(D\) is a conjunction, disjunction or existential quantifier:
     • If the corresponding rule is applicable on the axiom, we associate the pair \((\text{sizeC}(D),0)\),
     • else \((0,0)\);
  3. If \(D\) is a universal quantifier \((D = \forall r C)\) we associate the pair \((\text{Comp}_1,\text{Comp}_2)\) such that:
     (a) \(\text{Comp}_1 = \text{sizeC}(D)\);
     (b) \(\text{Comp}_2 = \text{Comp}_21 + \text{Comp}_22\) such that:
        • \(\text{Comp}_21\) is the number of applicability of the rule \(\rightarrow_y\), ie the number of \(x r y\) in Abox\_imp such that \(y : C\) is not in the Abox\_imp. It is noted here that \(\text{Comp}_21\) decreases if the rule \(\rightarrow_y\) is applicable, but if we apply the rule \(\rightarrow_\exists\) on another fact, this measure may increase. For this, we add the component \(\text{Comp}_22\) which ensures that this remains constant by the application of another rule;
        • \(\text{Comp}_22\) is the number of \(\exists\)-terms reducible and hidden (in the sense that they appear in the structure of the concept) in the Abox. This value decreases if the rule \(\rightarrow_\exists\) is applied and remains constant or decreases if another rule is applied.

In the end we can prove that measure is well founded.

\textbf{lemma} \(\text{wf-measure-abox-impl-order}: \text{wf measure-abox-impl-order}\)

9 \textbf{Conclusion}

In this paper we have presented a definition of a reasoner validated for the description logic \(\mathcal{ALC}\) based on the method of semantic tableaux. This formalization in Isabelle and development is based on several modules:

• Specifying the syntax and semantics of \(\mathcal{ALC}\),
• Coding of Abox and the formalization of the transformation rules of semantic tableaux,
• The proof of the soundness and completeness of semantic tableaux,
• The proof of termination requires the definition of a measure for each Abox. We have shown that this measure decreases for each application of a rule.
• The implementation of Abox, tableaux and rules transformation
• Defining a strategy of proof for \(\mathcal{ALC}\)
• Finally, the extraction of an executable, certified reasoner in the Caml language.

We will consider several extensions of this work, among which are:
• Extensions of this work to more expressive logics used in the semantic Web, such as $\mathcal{SHOIQ}$ and $\mathcal{SHOIN}$.
• Sets of refinements by other more efficient data structures as lists, and in particular the use of indexing techniques to speed up testing unsatisfiability of a table (“clash”) or to identify the applicable rules.

References


