Semantic Web Verification: Verifying Reasoning in the Logic $\mathcal{ALC}$

José A. Alonso Jiménez
Joaquín Borrego Díaz
María J. Hidalgo Doblado
Francisco J. Martín Mateos y
José Luis Ruiz Reina

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 27 de Junio de 2006 (versión de February 21, 2007)

Abstract

In the Semantic Web, knowledge is usually structured in the form of ontologies, using the Web Ontology Language (OWL), which is based in part on the Description Logics (DLs). DLs are a family of logical formalisms for representing and reasoning about conceptual and terminological knowledge. Among these, the logic $\mathcal{ALC}$ is a ground DL used in practical cases. Formal verification of reasoning algorithms in these logics allows us to increase the reliability of the Semantic Web. In this work, we present a generic and extensible framework to verifying reasoning services in the logic $\mathcal{ALC}$, formalized in the PVS system.

Contents

1 Introduction

One of the main benefits of the envisioned Semantic Web [6] is the logic trust expected to achieve with its adoption [5]. However, several problems may limit this aim, many of them related to Ontological Engineering and Knowledge Representation issues [2]. For example, the lack of ontology evaluation. Among other features, logical consistency of ontologies is a critical feature [11]. Currently, several Automated Reasoning Systems (ARS), as FaCT++ [28], Pellet [26], or Racer [12], can decide such kind of consistency.

This paper concerns about other problem slightly different. In the design of systems for semantic interoperability in critical systems, the trust problem lifts to another: the trust in the ARS used, that is, its verification. This kind of problem can be attacked by

means of the formal verification of the ARS using a (possibly) higher order verification system.

This verification effort strongly depends on the logic behind the ontology language. A family of logic–based KR formalisms for terminological reasoning are the Description Logics (DLs) [4]. From the point of view of the classical logic, DLs are (usually) decidable fragments of First Order Logic (FOL), inheriting thus the FOL style semantics. Many of these logics have nice logical features (sound and complete inference mechanisms). Likewise, computational complexity of standard inference problems are well–known. The standard reasoning has been addressed using mainly tableau–like algorithms, and during the last decade, a lot of work on DLs has been devoted to investigating the classical trade–off between expressiveness and complexity [4, 9]. Nevertheless, from the point of view of verification of the reasoning, the trade–off is between expressiveness and verified reasoning.

Actually, the verification can be focused on two aspects. The verification of self logic [10, 8] and the verification of reasoning algorithms. Our team have developed several works in both lines. In the first aspect, in order to facilitate the reuse of the results, we have driven the research to design generic frameworks easily extendable to more complex logics. Examples are [15], where a generic framework for propositional logic reasoning algorithms is presented, and [2], where it is proposed an extension of OWL to allow to specify the type of reasoning advisable with the ontology. Examples of works based of the second line are the Knuth–Bendix based reasoning [24], the verification of Buchberger algorithm for polynomial based logical reasoning [16, 17] and Conceptual processing in Formal Concept Analysis [3].

With the present paper, we report our first approach to the verification of reasoning algorithms in DLs. Specifically, we verify, using the PVS verification system, the most common reasoning process in a basic DL, $\mathcal{ALC}$ [25], building a generic framework in the same line than [15]. The formal proofs developed in PVS are mainly adapted from Chap. 2, 3 of [27] and Chap. 5 of [19]. As the whole development consists of many PVS theorems and function definitions, we will only scratch its surface presenting the main results and a sketch of how the pieces fit together. We will necessarily omit many details that, we expect, can be inferred from the context. The whole PVS theory developed is avaliable at `http://www.cs.us.es/~mjoseh/alc/`.

2 Overview of PVS

PVS (Prototype Verification System) [20] is a general–purpose environment for developing specifications and proofs. In this section, we present a brief description of the PVS language and prover, introducing some of the notions used in this paper.

The PVS specification language [22] is built on a classical typed higher–order logic with the basic types `bool`, `nat`, `int`, in addition to the function type constructor `[D -> R]` and the product type constructor `[A, B]`. The type system is also augmented with *dependent*

types and *abstract data types*. In PVS, a set `A` with elements of a type `T` is identified with their characteristic predicates and thus, the expressions `pred[T]` and `set[T]` have the same meaning. A feature of the PVS specification language are *predicate subtypes*: the subtype `{x:T | p(x)}` consists of all the elements of type `T` verifying `p`. The notation `(A)` is used to indicate the subtype `{x:T | A(x)}`. Predicate subtypes are used for constraining domains and ranges of functions in a specification and, therefore, for defining partial functions. In general, type-checking with predicate subtypes is undecidable. Therefore, the type-checker generates proof obligations, called *type correctness conditions* (TCCs). This TCCs are either discharged by specialized proof strategies or proved by the user. In particular, for defining a recursive function, it must be ensured that the function terminates. For this purpose, in the definition of a recursive function, the user has to provide a *measure function*. This generates a TCC stating that the measure function applied to the arguments decreases with respect to a well–founded ordering in every recursive call.

A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts of a large number of theories. PVS specifications are packaged as *theories* that can be parametrized with respect to types and constants. The definitions and theorems of a theory can then be used by another theory by *importing* it. Proofs in PVS are presented in a sequent calculus. The commands of PVS prover includes induction, quantifier instantiation, rewriting and simplification.

3 Syntax and semantics of $\mathcal{ALC}$ in PVS

In this section, we present a brief introduction to $\mathcal{ALC}$–logic, its syntax and its semantic, along with the coresponding formalization in PVS.

Let NC be a set of *concept names* and NR be a set of *role names*. The set of $\mathcal{ALC}$–*concepts* is built inductively from these as described by the following grammar, where $A \in \mathsf{NC}$ and $R \in \mathsf{NR}$:

$$C ::= A \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \forall R.C \mid \exists R.C$$

We define the $\mathcal{ALC}$–concepts in PVS as a recursive datatype, using the mechanism for defining abstract datatypes [21], in which we specify the constructors, the accessors and the recognizers.

```
alc_concept[NC: TYPE+, NR: TYPE]: DATATYPE
 BEGIN
  alc_a(name: NC)                       : alc_atomic?
  alc_not(conc: alc_concept)            : alc_not?
  alc_and(conc1, conc2: alc_concept)    : alc_and?
  alc_or(conc1, conc2: alc_concept)     : alc_or?
  alc_all(role: NR, conc: alc_concept)  : alc_all?
  alc_some(role: NR, conc: alc_concept) : alc_some?
 END alc_concept
```

When a datatype is typechecked in PVS, a new theory is created providing axioms and inductions principles for this datatype. In particular, this theory contains the relation `subterm` (specifying the notion of *subconcept*) and the wellfounded relation `<<`, that is useful to make recursive definitions on concepts. For instance, $|C|$ (the number of symbols of C) can be defined recursively using `<<` as justifying measure.

A *general axiom* is an expression of the form $C \sqsubseteq D$ or of the form $C \doteq D$, where C and D are concepts. A *TBox* $\mathcal{T}$ is a finite set of general axioms. Let NI be a set of *individual names*. Given individual names $x, y \in NI$, a concept C, and a role name R, the expressions $x\colon C$, $(x, y)\colon R$ and $x \neq y$ are called *assertional axioms*. An *ABox* $\mathcal{A}$ is a finite set of assertional axioms. A *knowledge base* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$. We define in PVS the general axioms and assertional axioms as datatypes, and the TBox, ABox and knowledge base as types:

```
alc_gtax: DATATYPE
  BEGIN
    gci(antecedent, consequent: alc_concept)        : gci?
    concept_eq(antecedent, consequent: alc_concept) : concept_eq?
  END alc_gtax

assertional_ax: DATATYPE
  BEGIN
    instanceof(left:NI, right:alc_concept) : instanceof?
    related(left:NI, role:NR, right:NI)    : related?
    different_ni(left:NI, right:NI)        : different_ni?
  END assertional_ax

TBox: TYPE = finite_set[alc_gtax]
ABox: TYPE = finite_set[assertional_ax]
KnowledgeBase: TYPE = [TBox, ABox]
```

An $\mathcal{ALC}$*–interpretation* $\mathcal{I}$ is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non–empty set called the *domain*, and $\cdot^{\mathcal{I}}$ is an *interpretation function* that maps every concept name A to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$, every role name R to a binary relation $R^{\mathcal{I}}$ over $\Delta^{\mathcal{I}}$ and every individual x to an element of $\Delta^{\mathcal{I}}$. We represent an interpretation $\mathcal{I}$ as a structure that contains the domain of $\mathcal{I}$ and the functions that define the interpretation of concept names, role names, and the individuals:

```
interpretation: NONEMPTY_TYPE =
 [# int_domain:        (nonempty?[U]),
    int_names_concept: [NC -> (powerset(int_domain))],
    int_names_roles:   [NR -> PRED[[(int_domain),(int_domain)]]],
    int_names_ind:     [NI -> (int_domain)] #]
```

Note that in this specification we have used an universal type U to represent the elements

of the domain. Also, we have taken advantage of the ability of PVS to deal with dependent types.

The interpretation function is extended to non atomic concepts as follows:

$$\begin{array}{lcl}(\neg D)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus D^{\mathcal{I}} \\ (C_1 \sqcap C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\ (C_1 \sqcup C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}} \\ (\forall R.D)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} : (\forall b \in \Delta^{\mathcal{I}})[(a,b) \in R^{\mathcal{I}} \rightarrow b \in D^{\mathcal{I}}]\} \\ (\exists R.D)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} : (\exists b \in \Delta^{\mathcal{I}})[(a,b) \in R^{\mathcal{I}} \wedge b \in D^{\mathcal{I}}]\}\end{array}$$

We have specified this notion in PVS in a natural way, by recursion in C

```
int_concept(C,I): RECURSIVE set[U] =
 CASES C OF
  alc_a(B):       int_names_concept(I)(B),
  alc_not(D):     difference(int_domain(I), int_concept(D,I)),
  alc_and(C1,C2): intersection(int_concept(C1,I),int_concept(C2,I)),
  alc_or(C1,C2):  union(int_concept(C1,I),int_concept(C2,I)),
  alc_all(R,D):   {a:(int_domain(I)) |
                    FORALL (b:(int_domain(I))):
                      int_names_roles(I)(R)(a,b) IMPLIES
                      int_concept(D,I)(b)},
  alc_some(R,D):  {a:(int_domain(I)) |
                    EXISTS (b:(int_domain(I))):
                      int_names_roles(I)(R)(a,b) AND
                      int_concept(D,I)(b)}
 ENDCASES
MEASURE C BY <<
```

An interpretation $\mathcal{I}$ is a *model* of concept C if $C^{\mathcal{I}} \neq \emptyset$. C is called *satisfiable* if it has a model.

```
is_model_concept(I,C): bool = nonempty?(int_concept(C,I))
concept_satisfiable?(C): bool = EXISTS I: is_model_concept(I,C)
```

An interpretation $\mathcal{I}$ *satisfies* a general axiom $C \sqsubseteq D$ ($C \doteq D$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ ($C^{\mathcal{I}} = D^{\mathcal{I}}$). It *satisfies* $\mathcal{T}$ iff it satisfies every axiom in $\mathcal{T}$. In this case, $\mathcal{T}$ is called *satisfiable*, $\mathcal{I}$ is called a *model* of $\mathcal{T}$ and we write $\mathcal{I} \models \mathcal{T}$.

```
satisfies_gtax(I,Ax): bool =
  CASES Ax OF
    gci(C,D)        : subset?(int_concept(C,I),int_concept(D,I)),
    concept_eq(C,D) : int_concept(C,I) = int_concept(D,I)
  ENDCASES

is_model_TBox(I,T): bool =
```

```
  FORALL Ax: member(Ax,T) IMPLIES satisfies_gtax(I,Ax)

tbox_consistent(T): bool = EXISTS I: is_model_TBox(I,T)
```

Similary, an interpretation $\mathcal{I}$ *satisfies* an assertional axiom $x{:}C$ if $x^{\mathcal{I}} \in C^{\mathcal{I}}$, it satisfies $(x,y){:}\,R$ if $(x^{\mathcal{I}}, y^{\mathcal{I}}) \in R^{\mathcal{I}}$, and it satisfies $x \neq y$ if $x^{\mathcal{I}} \neq y^{\mathcal{I}}$. It *satisfies* $\mathcal{A}$ iff it satisfies every axiom in $\mathcal{A}$. In that case, $\mathcal{A}$ is called *satisfiable*, $\mathcal{I}$ is called a *model* of $\mathcal{A}$ and we write $\mathcal{I} \models \mathcal{A}$.

```
satisfies_aax(I,Aa): bool =
  CASES Aa OF
   instanceof(ni,C)     : member(int_names_ind(I)(ni),int_concept(C,I)),
   related(ni1,R,ni2)   : int_names_roles(I)(R)(int_names_ind(I)(ni1),
                                                int_names_ind(I)(ni2)),
   different_ni(ni1,ni2): int_names_ind(I)(ni1) /= int_names_ind(I)(ni2)
  ENDCASES

is_model_ABox(I,AB): bool =
  FORALL Aa: member(Aa,AB) IMPLIES satisfies_aax(I,Aa)

abox_satisfiable(AB:ABox): bool = EXISTS I: is_model_ABox(I,AB)
```

Finally, an interpretation $\mathcal{I}$ *satisfies* a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ if $\mathcal{I} \models \mathcal{T}$ and $I \models \mathcal{A}$. In this case, $\mathcal{K}$ is called *satisfiable* and $\mathcal{I}$ is called a *model* of $\mathcal{K}$.

```
is_model_kb(I,KB): bool =
  LET (T,AB) = KB IN is_model_TBox(I,T) AND is_model_ABox(I,AB)

kb_consistent(KB): bool = EXISTS I: is_model_kb(I,KB)
```

The previous definitions naturally pose some standard inference problems for DL systems, such as concept satisfiability, concept subsumption, knowledge base satisfiability or instance checking, which are all related. For example, it can be proved (and we have done it in PVS) that concept satisfiability can be reduced to ABox satisfiability (i.e., C is satisfiable iff for all $x \in \mathsf{NI}$, $\{x{:}C\}$ is satisfiable):

```
concept_satisfiable_equiv_abox_satisfiable_all: LEMMA
  concept_satisfiable?(C) IFF
  FORALL x: abox_satisfiable(singleton(instanceof(x,C)))
```

We have also proved in PVS a number of similar reduction results for $\mathcal{ALC}$.

4 Tableau Reasoning for $\mathcal{ALC}$–satisfiability

A tableau algorithm for $\mathcal{ALC}$ tries to prove satisfiability of a concept C by attempting to explicitly construct a model of C. This is done considering an individual name x_0 and manipulating the initial ABox $\{x_0{:}C\}$, applying a set of completion rules. In this process, we consider concepts in negation normal form (NNF), a form in which negations appear only in front of concept names.

4.1 Deciding concept satisfiability for $\mathcal{ALC}$

An ABox $\mathcal{A}$ contains a *clash* if, for an individual name $x \in \mathsf{NI}$ and a concept name $A \in \mathsf{NC}$, $\{x{:}A, x{:}\neg A\} \subseteq \mathcal{A}$. Otherwise, $\mathcal{A}$ is called *clash–free*.

```
contains_clash(AB): bool =
  EXISTS x,A: member(instanceof(x,alc_a(A)), AB) AND
              member(instanceof(x,alc_not(alc_a(A))), AB)
```

To test the satisfiability of an $\mathcal{ALC}$–concept C in NNF, the $\mathcal{ALC}$–*algorithm* works as follows. Starting from the initial ABox $\{x_0{:}C\}$ it applies the *completion rules* from Figure 1 which modify the ABox. It stops when a clash has been generated or when no rule is

$\rightarrow_\sqcap$:	if	$x{:}C \sqcap D \in \mathcal{A}$ and $\{x{:}C, x{:}D\} \not\subseteq \mathcal{A}$
	then	$\mathcal{A} \rightarrow_\sqcap \mathcal{A} \cup \{x{:}C, x{:}D\}$
$\rightarrow_\sqcup$:	if	$x{:}C \sqcup D \in \mathcal{A}$ and $\{x{:}C, x{:}D\} \cap \mathcal{A} = \emptyset$
	then	$\mathcal{A} \rightarrow_\sqcup \mathcal{A} \cup \{x{:}E\}$ for some $E \in \{C, D\}$
$\rightarrow_\exists$:	if	$x{:}\exists R.D \in \mathcal{A}$ and there is no y with $\{(x,y){:}R, y{:}D\} \subseteq \mathcal{A}$
	then	$\mathcal{A} \rightarrow_\exists \mathcal{A} \cup \{(x,y){:}R, y{:}D\}$ for a fresh individual y
$\rightarrow_\forall$:	if	$x{:}\forall R.D \in \mathcal{A}$ and there is a y with $(x,y){:}R \in \mathcal{A}$ and $y{:}D \notin \mathcal{A}$
	then	$\mathcal{A} \rightarrow_\forall \mathcal{A} \cup \{y{:}D\}$

Figure 1: The completions rules for $\mathcal{ALC}$

applicable. In the latter case, the ABox is *complete* and a model can be derived of it. The algorithm answers "C is satisfiable" if a complete and clash–free ABox has been generated.

We have formalized the completions rules in PVS as relations between ABoxes. For example, the specification of $\mathcal{A}_1 \rightarrow_\exists \mathcal{A}_2$ is

```
some_step(AB1,AB2): bool =
  EXISTS Aa: member(Aa,AB1) AND
             instanceof?(Aa) AND
             alc_some?(right(Aa)) AND
             LET R = role(right(Aa)), D = conc(right(Aa)) IN
               empty?(set_ni(R,D,AB1,left(Aa))) AND
               LET y = choose(complement_ni(AB1)) IN
```

```
                AB2 = add(instanceof(y,conc(right(Aa))),
                          add(related(left(Aa),role(right(Aa)),y), AB1))
```

where `empty?(set_ni(R,D,AB1,left(Aa)))` means that there is no y with $\{(x,y)\colon R, y\colon D\} \subseteq \mathcal{A}_1$ and, by the function `choose` of PVS, we pick an individual that not occurs in $\mathcal{A}_1$. In order to guarantee the existence of such individual, we have included the following assumption on the type that represents the individuals in the parameters of the PVS theory.

```
ax1: ASSUMPTION FORALL (S:finite_set[NI]): EXISTS (x:NI): NOT member(x,S)
```

Once the rules have been specified in this way, we define the successor relation on the type of ABoxes: $\mathcal{A}_1 \to \mathcal{A}_2$ if $\mathcal{A}_1$ not contains clash and $\mathcal{A}_2$ is obtained from $\mathcal{A}_1$ by the application of one completion rule.

```
successor(AB2,AB1): bool =
  (NOT contains_clash(AB1)) AND
  (and_step(AB1,AB2)  OR or_step_1(AB1,AB2) OR or_step_2(AB1,AB2) OR
   some_step(AB1,AB2) OR all_step(AB1,AB2))
```

The completion process can be seen as a closure process. The ABox $\mathcal{A}_2$ is an *expansion* of the ABox $\mathcal{A}_1$ if $\mathcal{A}_1 \stackrel{*}{\to} \mathcal{A}_2$, where $\stackrel{*}{\to}$ is the reflexive and transitive closure of $\to$. In PVS, we define the relation $\stackrel{*}{\to}$ as

```
is_expansion: pred[[ABox,ABox]] = rtr_cl(successor)
```

where `rtr_cl(r)` specifies the reflexive and transitive closure of a relation `r`. In order to define `rtr_cl`, we have developed a PVS theory about closures of binary relations, using inductive sets.

Example 4.1 *Let C be the concept $\forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F))$. Then,*

$$\begin{array}{rl} & \mathcal{A}_0 := \{x_0\colon \forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F))\} \\ \stackrel{*}{\to} & \mathcal{A}_1 := \mathcal{A}_0 \cup \{x_0\colon \forall R.D, x_0\colon \exists R.(D \sqcup E), x_0\colon \exists R.(D \sqcup F)\} \\ \to & \mathcal{A}_2 := \mathcal{A}_1 \cup \{(x_0,x_1)\colon R, x_1\colon D \sqcup E\} \\ \to & \mathcal{A}_3 := \mathcal{A}_2 \cup \{x_1\colon D\} \end{array}$$

An ABox $\mathcal{A}$ is *complete* if there is no $\mathcal{A}'$ such that $\mathcal{A} \to \mathcal{A}'$, and it is *consistent* if it has a complete and clash–free expansion.

```
complete(AB): bool = FORALL AB1: NOT successor(AB1,AB)

complete_clash_free(AB): bool = complete(AB) AND NOT contains_clash(AB)

is_consistent_abox(AB): bool =
  EXISTS AB1: is_expansion(AB)(AB1) AND complete_clash_free(AB1)
```

Finally, a concept C is consistent if the initial ABox $\{x_0\colon C\}$ is consistent.

```
is_consistent_concept(C): bool =
  is_consistent_abox(singleton(instanceof(x_0,C)))
```

This definition is the PVS specification of the $\mathcal{ALC}$–algorithm for deciding satisfiability of $\mathcal{ALC}$–concepts. To prove that the algorithm is correct, we have to establish its termination, soundness and completeness. The following subsections describe the PVS proofs of these three properties.

4.2 Soundness

We have proved in PVS that the $\mathcal{ALC}$–algorithm is sound, that is:

```
alc_soundness: THEOREM
  is_consistent_concept(C) IMPLIES concept_satisfiable?(C)
```

The proof is based on the following lemmas.

Lemma 4.2 If $\mathcal{A}$ is a complete and clash-free expansion of an initial ABox $\mathcal{A}_0$, then $\mathcal{A}$ is satisfiable.

```
alc_soundness_L1: LEMMA
  FORALL (AB:expansion_abox(ABI)):
  complete(AB) AND NOT contains_clash(AB) IMPLIES abox_satisfiable(AB)
```

We have proved this lemma by specifying in PVS the canonical interpretation $\mathcal{I}_{\mathcal{A}}$ associated with $\mathcal{A}$, and proving that $\mathcal{I}_{\mathcal{A}}$ is a model of $\mathcal{A}$.

Lemma 4.3 If $\mathcal{A}_1 \rightarrow \mathcal{A}_2$ and $\mathcal{A}_2$ is satisfiable, then $\mathcal{A}_1$ is satisfiable too.

```
abox_satisfiable_successor: LEMMA
  successor(AB2,AB1) AND abox_satisfiable(AB2) IMPLIES abox_satisfiable(AB1)
```

To prove the final soundness theorem, note that the fact of defining the expansion relation by means of the inductive definitions of closure suggests the following induction scheme:

```
expansion_abox_induct: THEOREM
  FORALL (P: pred[expansion_abox(ABI)]):
    (P(ABI) AND
     FORALL (AB1,AB2: expansion_abox(ABI)):
            successor(AB2,AB1) AND P(AB1) IMPLIES P(AB2))
    IMPLIES FORALL (AB: expansion_abox(ABI)): P(AB)
```

This induction scheme (very useful for proving results about expansions), together with the above lemma allows us to deduce the satisfiability of the $\{x_0{:}C\}$ from the satisfiability of any of its expansions and, so, the satisfiability of C.

4.3 Termination

To verify the termination of the $\mathcal{ALC}$ –algorithm it suffices to prove that the successor relation, defined on the inductive set $\mathcal{E}(C)$ of the expansions of the initial ABox, is well founded:

```
well_founded_succesor: THEOREM
  well_founded?[expansion_abox_concept(C)](successor)
```

The proof is based on the existence of a type T, a relation $<$ well founded in T and a function

$$\mu_C : \mathcal{E}(C) \to T \tag{1}$$

such that

$$(\forall \mathcal{A}_1, \mathcal{A}_2)[\mathcal{A}_1 \to \mathcal{A}_2 \Rightarrow \mu_C(\mathcal{A}_2) < \mu_C(\mathcal{A}_1)] \tag{2}$$

We name the functions with these properties as *measure functions*. In order to formalize the existence of measure functions, we have included T and $<$ in the parameters, and (1) and (2) in the body of the PVS theory.

```
[..., T: TYPE+, <: (well_founded?[T])]: THEORY

measure_concept(C): [expansion_abox_concept(C) -> T]

measure_concept_decrease_successor: AXIOM
  FORALL (AB1,AB2: expansion_abox_concept(C)):
    successor(AB2,AB1)
    IMPLIES measure_concept(C)(AB2) < measure_concept(C)(AB1)
```

We initially assume as an axiom the intended properties of such measure function called `measure-concept(C)` and we will later show the existence of a function with these properties. For that purpose, we have made use of the so-called *multiset extension* of a well-founded relation, which can be proved to be also well-founded. The definition of such measure function is quite complex. For that reason (and also because the PVS proof of the well-foundedness of the multiset extension is interesting in its own) we postpone its explanation in more detail to section 5.

4.4 Completeness

We have proved in PVS that the $\mathcal{ALC}$ –algorithm is complete, that is:

```
alc_completeness: THEOREM
  concept_satisfiable?(C) IMPLIES is_consistent_concept(C)
```

The proof is based on the following lemmas.

Lemma 4.4 If $\mathcal{A}$ is a satisfiable ABox, then $\mathcal{A}$ is clash–free.

```
alc_completeness_L1: LEMMA
  abox_satisfiable(AB) IMPLIES NOT contains_clash(AB)
```

Lemma 4.5 If $\mathcal{A}_1$ is a satisfiable and not complete ABox, then there exists a satisfiable ABox $\mathcal{A}_2$, successor of $\mathcal{A}_1$.

```
alc_completeness_L2: LEMMA
  abox_satisfiable(AB1) AND NOT complete(AB1)
  IMPLIES EXISTS AB2: successor(AB2,AB1) AND abox_satisfiable(AB2)
```

Lemma 4.6 If $\mathcal{A} \in \mathcal{E}(C)$ is satisfiable, then there exists in $\mathcal{E}(C)$ a complete and clash–free expansion of $\mathcal{A}$.

```
alc_completeness_L3: LEMMA
  FORALL (AB: expansion_abox_concept(C)):
    abox_satisfiable(AB)
    IMPLIES EXISTS (AB2: expansion_abox_concept(C)):
                   is_expansion(AB)(AB2) AND complete_clash_free(AB2)
```

This is the main lemma for the proof of the completeness theorem. We have proved it by well founded induction on the successor relation.

5 Measure on $\mathcal{ALC}$–expansion of C

In this section we show a measure function on $\mathcal{E}(C)$ verifying the monotonicity condition of section 4.3. For this, we have considered T as the type of the finite multisets of pairs of natural numbers $\mathcal{M}(\mathbb{N} \times \mathbb{N})$, and the well-founded order as the extension to $\mathcal{M}(\mathbb{N} \times \mathbb{N})$ of the lexicographic order of $\mathbb{N} \times \mathbb{N}$, that we denote by $<_{mult}$. We have extracted the idea of this function from [19].

5.1 Wellfounded ordering of multisets

Let $<$ be a relation in T. We define the relation between finite multisets of T, $N <_1 M$, if N can be obtained replacing an element a of M by a multiset K of elements less than a. Also, we said that $N <_{mult} M$ if there exists multisets M_0, K_1, K_2 such that $K_1 \neq \emptyset$, $M = M_0 \uplus K_1$, $N = M_0 \uplus K_2$ and $(\forall a)[a \in K_2 \rightarrow (\exists b)[b \in K_1 \wedge a < b]]$.

In order to guarantee that $<_{mult}$ is a well founded order in $\mathcal{M}(\mathbb{N} \times \mathbb{N})$ we have extended the multisets library of PVS to include the wellfoundedness of the multiset order. The proof we have formalized in PVS is based in the proof made by W. Buchholz [18].

In the prelude of PVS, it is defined that $<$ is a well founded relation in T if every non empty subset of T has a $<$–minimal element. Nevertheless, to extend the PVS library we

have used the characterization of wellfoundedness given by P. Aczel in [1]. This characterization uses the well–founded part of a relation, defined inductively as the smallest set of T, $W(T,<)$, such that

$$(\forall x)[(\forall y < x)[y \in W(T,<)] \rightarrow x \in W(T,<)]$$

The scheme of the PVS formalization is the following:

1. We define inductively the well–founded part of T with respect to $<$, and we prove that $<$ is well founded in T if and only if $W(T,<) = T$.

2. We define inductively the transitive closure, $<^+$, of a binary relation $<$ and we prove that if $<$ is well founded, then $<^+$ is well founded too, using the above characterization.

3. We prove that if $<$ is well founded in T, then $<_1$ is well founded in the finite multisets of T, $\mathcal{M}(T)$. So, $<_1^+$ is well founded too.

4. Finally, we prove that if the binary relation $<$ is transitive, then $<_1^+$ and $<_{mult}$ are the same. So, we have that $<_{mult}$ is well founded in $\mathcal{M}(T)$.

5.2 Definition of a Measure on $\mathcal{ALC}$–expansion

The idea to define the measure function μ_C is to map each expansion $\mathcal{A} \in \mathcal{E}(C)$ to a multiset of pairs, in such way that those pairs represent all possibles rules that can be applied to $\mathcal{A}$.

The first step is to define the notions of *level* and *colevel*. For this, we have used the library of graphs of PVS [7]. We define the *associated graph* to an ABox $\mathcal{A}$, $\mathcal{G}(\mathcal{A})$, as the graph whose vertices are the individuals that occur in $\mathcal{A}$, and whose edges are the subsets $\{x, y\}$, such that $(x, y)\colon R \in \mathcal{A}$ for some role R.

```
graph_assoc_abox(AB: ABox): graph[NI] =
  (# vert:= occur_ni(AB), edges:= dbl_assoc_abox(AB) #)
```

We have proved that if $\mathcal{A} \in \mathcal{E}(C)$, then $\mathcal{G}(\mathcal{A})$ is a tree with root x_0. This fact allows us to define the level of x in $\mathcal{A}$ as the length of the path from x_0 to x (minus 1), and the colevel of x in $\mathcal{A}$, $|x|_{\mathcal{A}}$, as the difference of $|C|$ and the level of x in $\mathcal{A}$.

```
level(AB)(x:(occur_ni(AB))): nat = l(path_from_root(AB)(x)) - 1
colevel(AB)(x:(occur_ni(AB))): nat = size(C) - level(AB)(x)
```

We have proved that the relation $\rightarrow$ preserves the colevel of individuals, and we have also proved that if y is *successor* of x in $\mathcal{A}$ (i.e., $(x, y)\colon R \in \mathcal{A}$), then $|y|_{\mathcal{A}} = |x|_{\mathcal{A}} - 1$. Both properties are essential to prove the monotonicity of μ_C.

```
successor_preserve_colevel: LEMMA
 occur_ni(AB1)(y) AND successor(AB2,AB1)
 IMPLIES colevel(AB2)(y) = colevel(AB1)(y)

colevel_successor_related: LEMMA
 successor_related(AB)(y,x) IMPLIES colevel(AB)(y) = colevel(AB)(x) - 1
```

As we have already said, the elements of the multiset associated to an expansion $\mathcal{A}$ should represent all possibles applicables rules to $\mathcal{A}$. In some cases, the applicability of a rule is completely determined by an instance axiom of $\mathcal{A}$, but that is not the case for the $\rightarrow_\forall$ rule. Thus, in order to capture the notion of applicability of a rule, we have introduced the type *activation* (`activ`), whose elements are structures consisting in an instance axiom and an individual, that made it applicable. Then, we have specified when an activation is applicable in $\mathcal{A}$ and we have defined the *agenda* of $\mathcal{A}$, $\mathsf{agenda}(\mathcal{A})$, as the set of applicable activations in $\mathcal{A}$.

```
activ: TYPE = [# ax: (instanceof?), witness: NI #]

applicable_activ(Ac,AB): bool =
 LET Aa = ax(Ac),y = witness(Ac),x = left(Aa),D = right(Aa) IN
 member(Aa,AB) AND
 CASES D OF
  alc_a(A)       : false,
  alc_not(D1)    : false,
  alc_and(C1,C2) : x = y AND (NOT member(instanceof(x,C1),AB) OR
                              NOT member(instanceof(x,C2),AB)),
  alc_or(C1,C2)  : x = y AND NOT member(instanceof(x,C1),AB) AND
                             NOT member(instanceof(x,C2),AB),
  alc_all(R,D1)  : x /= y AND member(related(x,R,y),AB) AND
                              NOT member(instanceof(y,D1),AB),
  alc_some(R,D1) : x = y AND NOT (EXISTS y: member(related(x,R,y),AB) AND
                                            member(instanceof(y,D1),AB))
 ENDCASES

agenda(AB): finite_set[activ] = {Ac | applicable_activ(Ac, AB)}
```

To specify the function μ_C we have found the following difficulty: we can not define a multiset in PVS in a declarative way, as with sets. Thus, the measure of the expansion $\mathcal{A}$, $\mu_C(\mathcal{A})$, is constructed by recursion in the agenda of $\mathcal{A}$, adding for each applicable activation, $[x{:}D, y]$, the element $(|y|_{\mathcal{A}}, |D|)$.

```
bag_assoc_activ(Ac, AB): finite_bag[[nat, nat]] =
  IF NOT applicable_activ(Ac, AB) THEN emptybag
  ELSE LET Aa = ax(Ac), y = witness(Ac), D = right(Aa) IN
```

```
            singleton_bag((colevel(AB)(y), size(D)))
  ENDIF

expansion_measure_aux(AB,(AB1: finite_set[activ])):
                      RECURSIVE finite_bag[[nat, nat]] =
  IF empty?(AB1) THEN emptybag
  ELSE plus(bag_assoc_activ(choose(AB1), AB),
            expansion_measure_aux(AB, rest(AB1)))
  ENDIF
MEASURE card(AB1)

expansion_measure(AB): finite_bag[[nat, nat]] =
  expansion_measure_aux(AB, agenda(AB))
```

Example 5.1 *The agendas and measures of the ABoxes of Example 4.1 are:*

	agenda	measure
$\mathcal{A}_0$	$\{(x_0{:}\forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F)), x_0)\}$	$\dot{\{}(15,15)\dot{\}}$
$\mathcal{A}_1$	$\{(x_0{:}\exists R.(D \sqcup E), x_0), (x_0{:}\exists R.(D \sqcup F), x_0)\}$	$\dot{\{}(15,5),(15,5)\dot{\}}$
$\mathcal{A}_2$	$\{(x_0{:}\exists R.(D \sqcup F), x_0), (x_0{:}\forall R.D, x_1), (x_1{:}D \sqcup E, x_1)\}$	$\dot{\{}(14,3),(14,3),(15,5)\dot{\}}$
$\mathcal{A}_3$	$\{(x_0{:}\exists R.(D \sqcup F), x_0)\}$	$\dot{\{}(15,5)\dot{\}}$

Finally, we prove the theorem that assures the monotony of μ_C.

```
expansion_measure_decrease_successor: THEOREM
  successor(AB2, AB1)
  IMPLIES less_mult(expansion_measure(AB2), expansion_measure(AB1))
```

The formalization of the proof of this theorem in PVS has turned out to be more difficult than the hand proof presented in [19]. Firstly, we can observe (in Example 5.1) that if $\mathcal{A}_1, \mathcal{A}_2 \in \mathcal{E}(C)$ are such that $\mathcal{A}_1 \to \mathcal{A}_2$, then there exists an activation $A_1 = [x\colon D, y] \in \mathsf{agenda}(\mathcal{A}_1)$, that matches with the applied rule. In addition, we have that $A_1 \notin \mathsf{agenda}(\mathcal{A}_2)$ and, for each activation A_2 introduced in the $\mathsf{agenda}(\mathcal{A}_2)$ as result of the rule application, its associated pair is smaller (lexicographically) than $(|y|_{\mathcal{A}}, |D|)$. Indeed, one of the following cases may occur:

1. $A_2 = [x\colon E, z]$, being z successor of y in $\mathcal{A}_2$. Then, $|z|_{\mathcal{A}_2} < |y|_{\mathcal{A}_2} = |y|_{\mathcal{A}_1}$. So, $(|z|_{\mathcal{A}_2}, |E|) <_{lex} (|y|_{\mathcal{A}_1}, |D|)$.

2. $A_2 = [x\colon E, y]$, being E subconcept of D. Then, $|y|_{\mathcal{A}_2} = |y|_{\mathcal{A}_1}$ and $|E| < |D|$. So, $(|y|_{\mathcal{A}_2}, |E|) <_{lex} (|y|_{\mathcal{A}_1}, |D|)$.

Secondly, we should note that the application of a rule can disable some activations of the agenda and, so, it can eliminate its associated pairs of the multiset. We have

defined in PVS the multiset $K_1 = \mu_{aux}(\mathcal{A}_1, \mathsf{agenda}(\mathcal{A}_1) \setminus \mathsf{agenda}(\mathcal{A}_2))$, whose elements are the pairs associated to disabled activations. On the other hand, the multiset $K_2 = \mu_{aux}(\mathcal{A}_2, \mathsf{agenda}(\mathcal{A}_2) \setminus \mathsf{agenda}(\mathcal{A}_1))$ contains the pairs associated to new enabled activations. Finally, $M_0 = \mu_{aux}(\mathcal{A}_1, \mathsf{agenda}(\mathcal{A}_1) \cap \mathsf{agenda}(\mathcal{A}_2))$ is the multiset whose elements are the pairs associated to the activations that remains enabled after the application of the rule. Regarding these multisets, we prove the following properties: (1) $K_1 \neq \emptyset$, (2) $\mu_C(\mathcal{A}_2) = M_0 \uplus K_2$, (3) $\mu_C(\mathcal{A}_1) = M_0 \uplus K_1$ y (4) $(\forall a \in K_2)(\exists b \in K_1)[a <_{lex} b]$. Thus, we conclude that $\mu_C(\mathcal{A}_2) <_{mult} \mu_C(\mathcal{A}_1)$.

Once the measure function has been constructed, the parameters T and $<$, and the signature `measure-concept(C)` will be interpretated by the appropriated mechanism of PVS.

6 Conclusions and future work

We have formalized the $\mathcal{ALC}$ logic and proved the correctness (soundness, completeness and termination) of a tableau-based algorithm for deciding satisfiability of $\mathcal{ALC}$ –concepts. To our knowledge, this is the first work on formalizing DL reasoning, although related works (see, for example [15, 13, 23]) have been done for other logics.

For this task, it has been essential the use of some available PVS libraries, and others specifically developed for this formalization. In particular, we have developed a library about the reflexive and transitive closure of a relation and we have extended the multisets library, including the multiset extension relation and its well–foundedness. Furthermore, the work has been benefited by the use of PVS inductive definitions, which provide suitable induction schemes for most of the proofs.

The hard part of the formal proof presented has been the termination of the algorithm. In order to be able to formalize the proof given in [19] we needed to introduce new concepts that, in some sense, are implicit in the standard hand proofs. This issue is common in the formalization of mathematical knowledge.

We are considering two lines of possible future work: (1) carry out our formalization to other DLs that extends the $\mathcal{ALC}$ logic, and (2) optimize tableaux decision procedures for DL. As for (1), we intend to formalize OWL DL using the correspondence established in [14]. In summary, we think that our approach provides a good basis to the verification of reasoning in Semantic Web.

References

[1] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North–Holland Publishing Company, 1977.

[2] J. A. Alonso, J. Borrego, A. M. Chávez, and F. J. Martín. Foundational challenges in automated semantic web data and ontology cleaning. *IEEE Intelligent Systems*, 21(1):45–52, 2006.

[3] J. A. Alonso, J. Borrego, M. J. Hidalgo, F. J. Martín, and J. L. Ruiz. Verification of the formal concept analysis. *RACSAM (Revista de la Real Academia de Ciencias), Serie A: Matemáticas*, 98:3–16, 2004.

[4] F. Baader, D. McGuiness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[5] T. Berners-Lee. WWW past & future. Available at `http://www.w3.org/2003/Talks/0922-rsoc-tbl/`.

[6] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientic American*, 284(5):34–43, 2001.

[7] R. W. Butler and J. Sjogren. A PVS graph theory library. NASA Langley Technical Report Server, 1998.

[8] J. Dong, Y. Feng, and Y. Li. Verifying OWL and ORL ontologies in PVS. In *Theoretical Aspects of Computing ICTAC 2004*, volume 3407 of *Lecture Notes in Computer Science*, pages 265–279. Springer–Verlag, 2005.

[9] F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford, California, 1996.

[10] R. Fikes, D. McGuinness, and R. Waldinger. A first–order logic semantics for semantic web markup languages. Technical report, Knowledge Systems Laboratory, January 2002.

[11] A. Gómez-Pérez. Evaluation of ontologies. *Int. J. Intell. Syst.*, 16:391–409, 2001.

[12] V. Haarslev and R. Möller. RACER system description. In *International Joint Conference on Automated Reasoning, IJCAR'2001*, volume 2083 of *Lecture Notes in Computer Science*, pages 701–705. Springer–Verlag, 2001.

[13] J. Harrison. Formalizing basic first order model theory. In *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *Lecture Notes in Computer Science*. Springer–Verlag, 1998.

[14] I. Horrocks and P. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *J. of Web Semantics*, 1(4):345–357, 2004.

[15] F. J. Martín, J. A. Alonso, M. J. Hidalgo, and J. L. Ruiz. Formal verification of a generic framework to synthesize SAT–provers. *Journal of Automated Reasoning*, 32(4):287–313, 2004.

[16] I. Medina, F. Palomo, and J. A. Alonso. A certified polynomial-based decision procedure for propositional logic. In *Theorem Proving in Higher Order Logics TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 297–312. Springer–Verlag, 2001.

[17] I. Medina, F. Palomo, J. A. Alonso, and J. L. Ruiz. Verified computer algebra in ACL2. Gröbner bases computation. In *Artificial Intelligence and Symbolic Computation, AISC 2004*, volume 3249 of *Lecture Notes in Computer Science*, pages 171–184. Springer–Verlag, 2004.

[18] T. Nipkow. An inductive proof of the wellfoundedness of the mulitset order. `http://www4.informatik.tu-muenchen.de/~nipkow/misc`, October 1998. A proof due to W. Buchholz.

[19] W. Nutt. *Algorithms dor constraint in deduction and knowledge representation*. PhD thesis, Universität des Saarlandes, 1993.

[20] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer–Verlag, 1992.

[21] S. Owre and N. Shankar. Abstract datatype in PVS. Technical report, Computer Science Laboratory, SRI International, 1997.

[22] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS language reference. Technical report, Computer Science Laboratory, SRI International, 1999.

[23] T. Ridge and J. Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 294–309. Springer–Verlag, 2005.

[24] J. L. Ruiz, J. A. Alonso, M. J. Hidalgo, and F. J. Martín. Formal proofs about rewriting using ACL2. *Ann. Math. Artif. Intell*, 36(3):239–262, 2002.

[25] M. Schmidt–Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[26] E. Sirin and B. Parsia. Pellet system description. In *Proceedings of the 2006 International Workshop on Description Logics (DL2004)*, volume 189 of *CEUR Workshop Proceedings*, 2006.

[27] S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, LuFG Theoretical Computer Science. RWTH-Aachen, Germany, 2001.

[28] D. Tsarkov and I. Horrocks. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, pages 292–297. LNAI, 4130.