Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 200 (2008) 87–102
www.elsevier.com/locate/entcs

Constructing Formally Verified Reasoners for the $\mathcal{ALC}$ Description Logic

M. J. Hidalgo, J. A. Alonso, F. J. Martín and J. L. Ruiz [1,2]

Department of Computer Sciencs
University of Seville
Sevilla, Spain

Abstract

Description Logics are a family of logics used to represent and reason about conceptual and terminological knowledge. Recently, its importance has been increased since they are used as a basis for the Ontology Web Language (OWL) used for the Semantic Web. In previous work, we have developed in PVS a generic framework for reasoning in the $\mathcal{ALC}$ description logic, proving its termination, soundness and completeness. In this paper we present the construction, from the generic framework, of a formally verified generic tableau–based algorithm for checking satisfiability of $\mathcal{ALC}$–concepts. We do it using a methodology of refinements to transfer the properties from the framework to the algorithm. We also obtain some verified reasoners from the algorithm by a process of instantiation.

Keywords: Semantic Web, Description Logics, Verification, Formal Methods.

1 Introduction

For processing knowledge in the Semantic Web, reasoners of Description Logics (DLs) such as RACER, Pellet and FaCT++ [6,11,12] are being used. Description Logics [3] are a family of logics used to represent conceptual and terminological knowledge. Among these, the $\mathcal{ALC}$ logic is a ground logic, which can be extended to the more expressive logic $\mathcal{SHOIN}$, which corresponds to the Ontology Web Language.

Formally verifying the reasoners for DLs could increase their reliability and so that of the Semantic Web. However, formal verification of properties of reasoners for DLs is a time and resource consuming task. Moreover, if we carry out the formal verification of different reasoners for a logic, we will probably have to solve analogous problems for each one.

[1] This research was partially funded by Spanish Ministry of Education and Science under grant TIN2004–03884 and Feder funds

[2] Email: mjoseh,jalonso,fjesus,jruiz@us.es

doi:10.1016/j.entcs.2008.04.094

In [1], we have formalized a generic framework for checking satisfiability of $\mathcal{ALC}$–concepts in the PVS verification system [9]. The goal of this work is to construct reasoners for the $\mathcal{ALC}$–description logic from a generic framework in such a way that the verified properties of the generic framework are transferred to the reasoners. For this, in order to transfer the correctness of the framework to the reasoners we apply the type and operator refinement techniques shown in [2]. The main phases of the process we have followed, whose details we explain below, are shown in Figure 1.

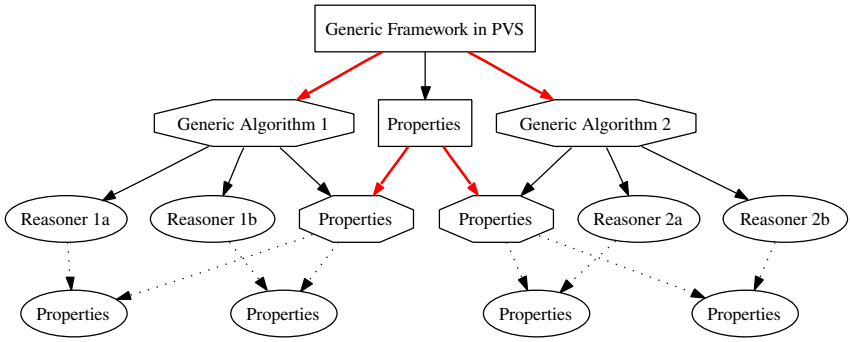

Fig. 1. The roadmap

(i) *Generic framework* (in the figure, the boxes). In this phase, which will be summarized in Section 3, we formalize in PVS a generic specification (using generic types) for checking satisfiability of $\mathcal{ALC}$–concepts. We prove termination, soundness and completeness. The features of the specification make it feasible that the proofs of its properties are close to the same ones in the usual literature. It should be noted that, in general, the most difficult proof is that of termination.

(ii) *Generic algorithm* (in the figure, the octagons). In this second phase, described in Section 5, we implement in PVS a generic algorithm corresponding to the previous specification, in such a way that the correctness properties of this algorithm are based on the same properties already proved for the generic framework. For this purpose, we use the methodology of refinements described in Section 4.

In this step, the algorithm is generic in the sense that the strategy of application of completion rules is not determined. That is, the specified algorithm depends on a selection function coding the strategy. The correctness of the algorithm has been proved, assuming some generic hypotheses about the non-determined selection function.

(iii) *Reasoners* (in the figure, the ellipses). In the last phase, described in Section 6, we develop reasoners for the $\mathcal{ALC}$ logic in PVS, considered as instances of the generic algorithm. For this, it suffices to instantiate the strategy of application of completion rules by a selection function. It should be emphasized that in order to prove the correctness of these reasoners, we only have to prove instances of the assumed hypotheses, for each concrete selection function.

To develop our work, we have chosen the PVS system. This system combines an expressive specification language with an interactive theorem prover. Also, although the PVS specification language has been designed to be expressive rather than executable, a wide fragment of PVS is executable by generating Common Lisp code from PVS, that can be evaluated through the PVSio environment [8].

2 Overview of PVS

PVS (Prototype Verification System) [9] 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 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*. 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.

3 A generic framework for checking satisfiability of $\mathcal{ALC}$ –concepts

The goal of this section is to present a summary of a PVS formalization of a generic framework for checking satisfiability of $\mathcal{ALC}$ –concepts, in which different tableaux–based algorithms can be placed. A more detailed description of this formalization, can be seen in [1] and the whole formalization is available at http://www.cs.us.es/~mjoseh/alc/.

Since one of our goals in the development of the formalization of the generic framework is to obtain a high degree of generality, we have specified the relations in a declarative way, instead of by the corresponding function. Another goal of our development is to obtain PVS proofs closely resembling the proofs that we can find

in the usual literature, in such a way that the complexity of a PVS proof stems from the proof itself, and not from the additional complexity introduced by the use of some specific data structure. For that reason, we have mainly used the type of finite sets.

We first describe the basic components of the $\mathcal{ALC}$ logic, and we show below how we have formalized in PVS a generic framework for tableau–based algorithms for this logic. We present the $\mathcal{ALC}$ logic along with the corresponding description of its specification 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 names as described by the following grammar, where $A \in \mathsf{NC}$ and $R \in \mathsf{NR}$

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

The set of $\mathcal{ALC}$–concepts can be represented in PVS as a recursive datatype, using the mechanism for defining abstract datatypes [10], and specifying the constructors, the accessors and the recognizers.

To introduce the assertional knowledge, let NI be a set of *individual names*. Given individual names $x, y \in \mathsf{NI}$, a concept C and a role name R, the expressions $x\colon C$ and $(x, y)\colon R$ are called *assertional axioms*. An *ABox* $\mathcal{A}$ is a finite set of assertional axioms. We specify in PVS the assertional axioms by a datatype and the ABox by a type

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

ABox: TYPE = finite_set[assertional_ax]
```

The semantics of description logics is defined in terms of interpretations. 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 in PVS 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)] #]
```

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

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

The interpretation $\mathcal{I}$ is a *model* of a concept C if $C^{\mathcal{I}} \neq \emptyset$. Thus, a concept 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)
```

The interpretation $\mathcal{I}$ *satisfies* the assertional axiom $x{:}C$ if $x^{\mathcal{I}} \in C^{\mathcal{I}}$ and satisfies $(x,y){:}R$ if $(x^{\mathcal{I}}, y^{\mathcal{I}}) \in R^{\mathcal{I}}$. It *satisfies* the ABox $\mathcal{A}$ if it satisfies every axiom in $\mathcal{A}$. In that case, $\mathcal{A}$ is called *satisfiable* and $\mathcal{I}$ is called a *model* of $\mathcal{A}$.

We have made the PVS formalization of the above definitions, in a generic way using the PVS set theory and its capability of managing the existential and universal quantifiers.

In order to decide the satisfiability of an $\mathcal{ALC}$–concept, a tableau algorithm tries to prove the 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. This does not impose any restriction since it is easy to specify a PVS function such that, for each $\mathcal{ALC}$–concept, computes another equivalent in NNF form.

An ABox $\mathcal{A}$ contains a *clash* if, for some individual name $x \in \mathsf{NI}$ and 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 Aa: member(Aa,AB) AND instanceof?(Aa) AND
            alc_atomic?(right(Aa)) AND
            member(instanceof(left(Aa), alc_not(right(Aa))), AB)
```

To test the satisfiability of an $\mathcal{ALC}$–concept C in NNF, the $\mathcal{ALC}$–*algorithm* works starting from the initial ABox $\{x_0{:}C\}$ and iteratively applying the following *completion rules*:

$\rightarrow_{\sqcap}$: 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 $\mathcal{A} \rightarrow_{\forall} \mathcal{A} \cup \{y{:}D\}$

It stops when a clash has been generated or when no rule is applicable. In the latter case, the ABox is *complete* and a model can be derived from it. The algorithm answers "C is satisfiable" if a complete and clash–free ABox has been generated.

We have formalized these completion rules following a declarative style, defining them in PVS as binary relations between ABoxes. For example, $\mathcal{A}_1 \rightarrow_{\sqcup_1} \mathcal{A}_2$ if there exists an assertional axiom $x{:}\, C \sqcup D$ in $\mathcal{A}_1$ such that $x{:}\, C \notin \mathcal{A}_1$, $x{:}\, D \notin \mathcal{A}_1$ and $\mathcal{A}_2 = \mathcal{A}_1 \cup \{x{:}C\}$

```
or_step_1(AB1, AB2): bool =
 EXISTS Aa: member(Aa, AB1) AND
            instanceof?(Aa) AND
            alc_or?(right(Aa)) AND
            NOT member(instanceof(left(Aa),conc1(right(Aa))),AB1) AND
            NOT member(instanceof(left(Aa),conc2(right(Aa))),AB1) AND
            AB2 = add(instanceof(left(Aa),conc1(right(Aa))),AB1)
```

Once the rules have been specified in this way, we define the successor relation on the ABoxes type: $\mathcal{A}_1 \rightarrow \mathcal{A}_2$ if $\mathcal{A}_1$ does not contain a clash and $\mathcal{A}_2$ is obtained from $\mathcal{A}_1$ by the application of a 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))
```

It should be noted that we have specified the non-deterministic rule $\rightarrow_{\sqcup}$ by two binary relations (`or_step_1` and `or_step_2`), one for each component.

Taking into account that the completion process can be seen as a closure process, we say that the ABox $\mathcal{A}_2$ is an *expansion* of the ABox $\mathcal{A}_1$ if $\mathcal{A}_1 \stackrel{*}{\rightarrow} \mathcal{A}_2$, where $\stackrel{*}{\rightarrow}$ is the reflexive and transitive closure of $\rightarrow$.

To illustrate the completion process, the following example shows the application of some completion rules to an initial ABox $\{x_0{:}C\}$

Example 3.1 Let C be the concept $\forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F))$. Then,

$$\mathcal{A}_0 := \{x_0{:}\forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F))$$
$$\xrightarrow{*} \mathcal{A}_1 := \mathcal{A}_0 \cup \{x_0{:}\forall R.D, x_0{:}\exists R.(D \sqcup E), x_0{:}\exists R.(D \sqcup F)\}$$
$$\rightarrow \mathcal{A}_2 := \mathcal{A}_1 \cup \{(x_0, x_1){:}R, x_1{:}D \sqcup E\}$$
$$\rightarrow \mathcal{A}_3 := \mathcal{A}_2 \cup \{x_1{:}D\}$$

Once defined the expansion relation, we use it to specify the notions of completeness and consistency. An ABox $\mathcal{A}$ is *complete* if it has not any successor and is *consistent* if it has a complete and clash–free expansion. Similarly, a concept C is *consistent* if the initial ABox $\{x_0{:}C\}$ is consistent

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

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

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

where `complete_clash_free`($\mathcal{A}$) holds if the ABox $\mathcal{A}$ is both complete and clash–free.

This definition is the PVS specification of a generic framework for deciding satisfiability of $\mathcal{ALC}$–concepts. It should be pointed out the two types of non–determinism in it: the way in which the rule $\rightarrow_{\sqcup}$ is applied ("don't know" non-determinism); and the choice of which rule to apply in each step and to which axiom ("don't care" non-determinism). We have established in [1] the correctness of this specification, proving its termination, soundness and completeness.

4 Methodology of refinements

In this section we present a sketch of the type and operator refinement techniques developed in PVS in order to relate different specifications of the same notion. The point is that if we want to prove properties about a program, the development of the formal proof will strongly depend of the used datatypes as well as the concrete implementation of it. Thus, the idea is to verify the desired properties for a generic specification of this program and that the verified properties can be transferred to it.

For this, based on the idea of refinement of data types used by A. Dold [5] and C. B. Jones [7] we have built in PVS a theory establishing the general notions of refinements and its main properties (see [2]). Given types T and R, we say that a *data refinement* of the type T by the type R is a surjective application $f : R \rightarrow T$

```
f: VAR [R->T]
is_ refinement?(f): bool = FORALL (t:T): EXISTS (r:R): f(r) = t
```

Intuitively, this function provides the relationship between abstract values (T)

and their representations (R). It is clear that there is at least one representation, not necessarily unique, for any abstract value. Also, we can see that a type can be refined stepwise by sequential composition of refinements. In that sense, we prove in PVS that, if $f : R \to T$ is a data refinement of T by R and $g : Q \to R$ is a data refinement of R by Q, then $f \circ g : Q \to T$ is a data refinement of T by Q.

With respect to the refinements of operators or functions, let us consider an operator $op : T_1 \to T_2$ and let us suppose that we have the data refinements given by the functions $f_1 : R_1 \to T_1$ and $f_2 : R_2 \to T_2$. We say that the operator $op_{ref} : R_1 \to R_2$ is a *refinement* of op if the following diagram commutes:

$$\begin{array}{ccc} T_1 & \xrightarrow{op} & T_2 \\ \uparrow & & \uparrow \\ f_1| & & f_2| \\ | & & | \\ R_1 & \xrightarrow{op_{ref}} & R_2 \end{array}$$

```
op: VAR [T1 -> T2]  op_ref: VAR [R1 -> R2]
is_refinement_op?(op,op_ref): bool =
 FORALL r1: op(f1(r1))=f2(op_ref(r1))
```

In essence, we require that op_{ref} has the same behavior as op. The most important feature of this definition is that it makes possible to transfer the properties of the operator op to the operator op_{ref}. For example, we prove that the well–foundedness of a relation can be transferred through the refinements

```
refinement_preserve_wf: LEMMA
 is_refinement_op?(rel,rel_ref) AND well_founded?(rel) IMPLIES
 well_founded?(rel_ref)
```

In a more general way, let us suppose that we have established a correctness theorem for op, in terms of pre and post conditions. That is, a theorem like

$$(\forall y \in T_1)[\phi(y) \Rightarrow \rho(y, op(y))]$$

where ϕ is the precondition and ρ is the postcondition. Then, if op_{ref}, ϕ_{ref} and ρ_{ref} are refinements of op, ϕ and ρ, respectively, we have proved that

$$(\forall x \in R_1)[\phi_{ref}(x) \Rightarrow \rho_{ref}(x, op_{ref}(x))]$$

which is just the correctness theorem corresponding to op_{ref}. And this has been proved in a general way without no specific assumptions about ϕ, ρ and op.

In order to illustrate the above idea, we show how we have formalized a refinement of finite sets by lists. Let us consider a data refinement $f : R \to T$. From this, we specify a data refinement of type "finite sets with elements in T" by the type "lists with elements in R", by the function `c(f):list[R]` $\to$ `finite_set[T]`, defined as follows

```
c(f)(l: list[R]): RECURSIVE finite_set[T] =
 CASES l OF
   null: emptyset,
   cons(x, l1): add(f(x), c(f)(l1))
 ENDCASES
 MEASURE length(l)
```

To build refinements corresponding to the operations over finite sets we use operations over lists "simulating" the behaviour of analogous operations on finite sets. For example, the built–in operations `null?`, `cons` and `append` are refinements of `empty?`, `add` and `union`, respectively. As for the membership relation, it can be noticed that if f is injective, the predicate `member` for lists is a refinement of the predicate `member` for finite sets.

Finally, regarding the construction of a refinement on a specification, we take into account that a specification of an algorithm is normally built combining some other specifications of operators. Hence, for constructing a refinement of a specification of an algorithm it suffices to construct a refinement of each operator used in it, and to replace it. In that sense, we prove that if $op_{1_{ref}}$ and $op_{2_{ref}}$ are refinements of op_1 and op_2, respectively, then $op_{2_{ref}} \circ op_{1_{ref}}$ is also a refinement of $op_2 \circ op_1$.

5 Generic algorithms for checking satisfiability of $\mathcal{ALC}$ – concepts

This section is devoted to present the construction of a generic algorithm corresponding to the specification of the generic framework that we have described in Section 3. In addition, our purpose is to do it in such way that its termination, soundness and completeness can be deduced from the corresponding properties of the generic framework. For this, we will use the methodology of refinements explained in Section 4.

It should be noted that the specification of the generic framework cannot be transformed into an algorithm by composition of refined operators for each of the operators composing this specification. The main reason is that in the generic framework, the searching process that the algorithm has to carry out in the space $\mathcal{E}(C)$ of the expansions of the initial ABox $\{x_0 : C\}$ is not specified. So, in the process of construction of the algorithm we have to concretize how to carry out the search. Also, we have to consider the following facts:

(i) It is necessary to use evaluable data types. For this, we have to refine, among others, the type used to represent ABoxes (finite sets) by another evaluable type (in this case, lists).

(ii) It is necessary to define evaluable specifications of the predicates used for recognizing if an ABox is complete and clash-free.

(iii) Since we will construct a recursive algorithm, it is necessary to have a well–foundness relation in $\mathcal{E}(C)$, that provides a measure function for proving its termination.

(iv) Finally, due to the non–determinism of the generic framework, it is necessary to determine the completion rule that will be applied in each step. Also, we need a function that applies a completion rule to an ABox.

In the following subsections, we describe the most significant features of each one of these points.

5.1 Refinements of data types

Firstly, let us note that in the specification of the generic framework we have used the same name (for example, `left`) to denote different accessor functions of the data types used to represent concepts and assertional axioms. However, although overloading of names does not present any problem for reasoning about the specifications, this fact is problematic for the PVS evaluator, since they cannot be distinguished by their type. This problem has been easily solved specifying new types that refine the previous ones, with different names for each function. For example, the type used to refine the datatype `assertional_ax` is the following

```
assertional_ax_ref: DATATYPE
 BEGIN
  r_instanceof(left_i: nat, right_i: r_alc_concept): r_instanceof?
  r_related(left_r: nat, role: NR, right_r: nat)   : r_related?
 END assertional_ax_ref
```

Also, we refine the types `ABox` and `is_expansion` as we show in the following table:

Notion	Type	Refined type	Refinement function
$\mathcal{ALC}$–concept	`alc_concept`	`r_alc_concept`	`alc_f`
Assertional ax.	`assertional_ax`	`assertional_ax_ref`	`alc_f_aax`
ABox	`ABox`	`LABox`	`c(alc_f_aax)`
Expansion	`is_expansion`	`is_expansion_l`	`c(alc_f_aax)`

where `LABox: TYPE = list[assertional_ax_ref]`

It should be observed that, for our purposes, it is not necessary to refine the specification used to represent the notion of interpretation, since this will not be used directly by the decision procedure. In the same way, it is not necessary for the notions of satisfiability to be evaluable; they only have to be equivalent to that defined in the generic framework. Thus, in this case we define the semantic notions for the refined types through the types refinements

```
r_concept_satisfiable?(C): bool = concept_satisfiable?(alc_f(C))
r_abox_satisfiable(L: LABox): bool = abox_satisfiable(c(alc_f_aax)(L))
```

5.2 Evaluable refined predicates

In order to define an evaluable refinement of the predicate `contains_clash`, we note that the range of the existential quantifier is the ABox $\mathcal{A}$ itself. Therefore, we can construct a refinement of this predicate using the PVS predicate `some` [3]

```
be_clash(Aa,L): bool =
 r_instanceof?(Aa) AND r_alc_atomic?(right_i(Aa)) AND member(Aa,L)
 AND member(r_instanceof(left_i(Aa), r_alc_not(right_i(Aa))), L)

contains_clash_l(L): bool = some(lambda(Aa): be_clash(Aa,L))(L)
```

However, in the case of the predicate `complete` the generic specification cannot be evaluable, not even in the case when the `successor` relation is it. Let us see how we have constructed a refinement of this predicate. Firstly, we define predicates that recognize when an assertional axiom is expansive in an ABox, according to each of the completion rules. For example, $x{:}C \sqcup D \in L$ is expansive in L if $x{:}C \notin L$ and $x{:}D \notin L$

```
is_or_expansive(Aa,L): bool =
 member(Aa,L) AND r_instanceof?(Aa) AND r_alc_or?(right_i(Aa)) AND
 NOT member(r_instanceof(left_i(Aa), conc1_or(right_i(Aa))), L) AND
 NOT member(r_instanceof(left_i(Aa), conc2_or(right_i(Aa))), L)
```

This makes possible to specify the notions of expansive axiom with respect to an ABox L, and the following specification of the predicate `complete_l`

```
is_expansive(Aa,L): bool =
 is_and_expansive(Aa,L) OR is_or_expansive(Aa,L) OR
 is_some_expansive(Aa,L) OR is_all_expansive(Aa,L)

complete_l(L): bool = NOT some(lambda(Aa): is_expansive(Aa,L))(L)
```

Regarding to the predicate `complete_l` it should be observed that it is not, exactly, a refinement of the predicate `complete`, although it can be seen as a refinement in conjunction with the predicate `not_contains_clash_l`

```
complete_iff_complete_l: THEOREM
 complete_l(L) AND NOT contains_clash_l(L) IFF
 complete(c(alc_f_aax)(L)) AND NOT contains_clash(c(alc_f_aax)(L))
```

5.3 Measure function

It should be pointed out that the role of the `successor` relation is the same, both in the specification of the generic framework and in the algorithm: it is a well founded relation necessary to ensure the termination of both specifications. Thus, it is not essential for the refined specification of the `successor` relation to be evaluable. Then, we could think in a definition of the `successor` relation in the same way that

[3] The predicate `some` is an executable PVS predicate that checks if some element of a list verifies a property.

we have defined the notions of satisfiability. That is

```
successor_l(L2,L1): bool =
  successor(c(alc_f_aax)(L2),c(alc_f_aax)(L1))
```

With this, it would be straightforward that `successor_l` is a refinement of `successor`. Then, we would prove that `successor_l` is a well founded relation, applying properties of refinements. However, due to its necessary relationship with the predicate `complete_l`, we have chosen to refine each one of the relations representing a completion rule. For example, L_2 is a $\sqcup_1$–expansion of L_1 if exists an axiom $x{:}C_1 \sqcup C_2$ expansive in L_1, and L_2 is obtained adding $x{:}C_1$ to L_1

```
r_or_step_1(L1,L2): bool =
 EXISTS Aa:
  is_or_expansive(Aa,L1) AND
  FORALL Aa1:
   member(Aa1,L2) IFF
   (member(Aa1,L1) OR
    Aa1=r_instanceof(left_i(Aa),conc1_or(right_i(Aa))))
```

From these definitions we define the relation `successor_l` and we prove that it is a refinement of `successor`

```
successor_l(L2,L1): bool =
 not_contains_clash_l(L1) AND
 (r_and_step(L1,L2) OR r_or_step_1(L1,L2) OR r_or_step_2(L1,L2)
  OR r_some_step(L1,L2) OR r_all_step(L1,L2))

successor_l_is_refinement: THEOREM
 is_refinement_op?(successor,successor_l)
```

The hardest part of the formalization of the generic framework was proving that the `successor` relation is a well–founded relation on the set $\mathcal{E}(C)$ of the expansions of a concept C in NNF. Now, this property can be transferred to the relation `successor_l` using that the well–foundedness of a relation is preserved through refinements

```
successor_l_is_wf: THEOREM
 well_founded?[expansion_abox_concept_l(C)](successor_l)
```

5.4 Application of completion rules

In order to construct the satisfiability algorithm we firstly specify some functions that, given an instance axiom Aa and an ABox L, compute the ABox corresponding with the application to L of some associated rule to Aa. For example, the result of apply the rule $\rightarrow_{\sqcup_1}$ to Aa and L is the ABox obtained adding $x : C_1$ to L, if $Aa = x{:}C_1 \sqcup C_2$ is or–expansive in L; and L, otherwise

```
or_step_1_ax(Aa,L): LABox =
 IF is_or_expansive(Aa,L)
```

```
THEN cons (r_instanceof(left_i(Aa), conc1_or(right_i(Aa))), L)
ELSE L ENDIF
```

Secondly, it should be taken into account that the applicability of a rule does not only depend on an instance axiom of an ABox L. In order to capture the notion of applicability of a rule, the type *activation* (`activ`) was introduced in the generic framework. An activation is a structure consisting of an instance axiom Aa and a witness x, which made it applicable. Now, we refine in a natural way the type `activ`, and we specify a function computing a list with the ABoxes obtained by application to L of the rules corresponding to an activation Ac

```
apply_activ(Ac: r_activ, L:LABox): list[LABox] =
 IF NOT r_applicable_activ(Ac,L)
 THEN null
 ELSE LET Aa = r_ax(Ac), D = right_i(Aa) IN
      CASES D OF
      r_alc_and(C1,C2): (: and_step_ax(Aa,L) :),
      r_alc_or(C1,C2) : (: or_step_1_ax(Aa,L), or_step_2_ax(Aa,L) :),
      r_alc_all(R,D1) : (: all_step_ax(Aa,L) :),
      r_alc_some(R,D1): (: some_step_ax(Aa,L) :)
     ENDCASES
 ENDIF
```

For example, let L be $(:\ x_0{:}\forall R.D, (x_0,x_1){:}R, x_0{:}D \sqcup E\ :)$. Then,

$\texttt{apply\_activ}([x_0{:}\forall R.D, x_0], L) = (:\ \ :)$

$\texttt{apply\_activ}([x_0{:}\forall R.D, x_1], L) = (:\ (:\ x_1{:}D, x_0{:}\forall R.D, (x_0,x_1){:}R, x_0{:}D \sqcup E\ :)\ :)$

$\texttt{apply\_activ}([x_0{:}D \sqcup E, x_0], L) = (:\ (:\ x_0{:}D, x_0{:}\forall R.D, (x_0,x_1){:}R, x_0{:}D \sqcup E\ :),$
$(:\ x_0{:}E, x_0{:}\forall R.D, (x_0,x_1){:}R, x_0{:}D \sqcup E\ :)\ :)$

With regard to the completion rule to apply in each step, our goal is to specify a decision procedure independent of the strategy followed in the order of application of these rules. For this, we declare the function

```
f: [LABox -> list[r_activ[NC,NR]]]
```

whose role is to select the rule to apply in each step. Then, for every different selection function, we will have a different decision procedure.

The idea is that $f(L)$ selects an activation applicable to L. With this activation the algorithm will carry out the next step of the completion process. Due to typing reasons, given an ABox L, $f(L)$ provides a list of activations. Thus, if there is not any activation applicable to L, $f(L)$ should be the empty list.

In order to ensure the correctness of the algorithm, the selection function f has to verify some properties, which we introduce as PVS assumptions. Firstly, let us observe that if L is not complete, then there is some rule applicable to L and therefore there is some activations applicable to f. In that sense, we require f to select at least one activation applicable to L whenever there are such activations.

Also, we require that f only selects activations applicable to L.

Thus, if L, L_1 are expansions of an initial concept C_0, the function f has to verify the following properties

```
f_ax_1: ASSUMPTION  NOT complete_l(L) IMPLIES cons?(f(L))

f_ax_2: ASSUMPTION
 FORALL (Ac:r_activ):member(Ac,f(L)) IMPLIES r_applicable_activ(Ac,L)
```

The generic algorithm we specify below is a tableau–based algorithm, that carries out a depth first search and whose size depends on the selection function f. It finishes when it finds a complete and clash-free ABox (that is, a non closed branch of the tableau from which a model of the initial concept can be constructed); or when all its branches are closed

```
sat_alc_alg_aux_i(L: expansion_abox_concept_l(C_0)): RECURSIVE bool =
 IF complete_l(L) AND not_contains_clash_l(L)
 THEN TRUE
  ELSIF contains_clash_l(L)
  THEN FALSE
  ELSE LET Ac = car(f(L)), S = apply_activ(Ac,L) IN
        IF null?(cdr (S))
        THEN sat_alc_alg_aux_i(car(S))
        ELSE LET L1 = car(S), L2 = car(cdr(S)) IN
              sat_alc_alg_aux_i(L1) OR sat_alc_alg_aux_i(L2)
        ENDIF
  ENDIF
 MEASURE L BY successor_l

sat_alc_alg_i: bool = sat_alc_alg_aux_i((: r_instanceof(0,C_0) :))
```

The termination of this algorithm is ensured by the well–foundedness of the `successor_l` relation. The soundness and completeness are proved by well–founded induction in the `successor_l` relation, using the same properties already proved for the specification of the generic framework and the properties required to f

```
sat_alc_alg_i_soundness: THEOREM
 sat_alc_alg_i IMPLIES r_concept_satisfiable?(C_0)

sat_alc_alg_i_completeness: THEOREM
 r_concept_satisfiable?(C_0) IMPLIES sat_alc_alg_i
```

6 Reasoners for checking satisfiability of $\mathcal{ALC}$–concepts

A particular reasoner can be constructed by defining a selection function verifying the assumptions of subsection 5.4 and instantiating the noninterpreted types used to represent the set of concepts names, the set of role names and the set of individuals. With each concrete function, a different specification of the generic algorithm can

be defined.

An usual application strategy of completion rules in functional algorithms deciding satisfiability of $\mathcal{ALC}$–concepts is the following (see [4]):

(i) Whenever the $\rightarrow_{\sqcap}$ rule can be applied and L is clash–free, apply the $\rightarrow_{\sqcap}$ rule;

(ii) Else, whenever the $\rightarrow_{\sqcup}$ rule can be applied and L is clash–free, apply the $\rightarrow_{\sqcup}$ rule;

(iii) Otherwise, if a $\rightarrow_{\exists}$ rule can be applied, apply the $\rightarrow_{\exists}$ rule and all the $\rightarrow_{\forall}$ rules derived from it.

In order to embed this decision procedure as an instance of the generic algorithm shown in subsection 5.4 it suffices to instantiate f by the following selection function [4]

```
f(L: LABox): list[r_activ] =
 IF cons?(list_first_r_activ_all(L))
 THEN (: car(list_first_r_activ_all(L)) :)
   ELSIF cons?(list_first_r_activ_and(L))
   THEN (: car(list_first_r_activ_and(L)) :)
    ELSIF cons?(list_first_r_activ_or(L))
    THEN (: car(list_first_r_activ_or(L)) :)
     ELSIF cons?(list_first_r_activ_some(L))
     THEN (: car(list_first_r_activ_some(L)) :)
      ELSE null[r_activ]
 ENDIF
```

where `list_first_r_activ_*`(L) is a list with one of the activations corresponding to the $\rightarrow_*$–rule applicable to L, if there are such activations; or the empty list, otherwise. Thus, the concrete decision procedure is

```
sat_alc_i(C): bool = sat_alc_alg_i[string,string,C,f]
```

Finally, in order to ensure the correctness of this reasoner it suffices to prove that f verifies the assumptions required in subsection 5.4, which are automatically generated by the system when the selection function is instantiated. Then, the theorems that ensure the correctness of this reasoner are the following

```
sat_soundness: THEOREM sat_alc_i(C) IMPLIES r_concept_satisfiable?(C)
sat_completeness: THEOREM r_concept_satisfiable?(C) IMPLIES sat_alc_i(C)
```

7 Conclusions and future work

We have presented a formalization of a generic framework for checking satisfiability of $\mathcal{ALC}$–concepts in PVS, for which we have proved its termination, soundness and completeness. From this framework, we have constructed a generic tableau–based algorithm using the methodology of refinements to transfer it its main properties.

[4] It should be noted that when the reasoner is running, f is applied to expansions of an initial ABox $\{x_0{:}C\}$. In this case, the $\rightarrow_{\forall}$–rule only can be applied after the application of a $\rightarrow_{\exists}$–rule.

Finally, we have obtained some concrete reasoners by instantiation of noninterpreted types and of the function which coded the strategy to select the rule to apply in each step.

It is worth pointing out that this makes the formal verification of the fundamental properties of the reasoners easier, since this is reduced to prove the hypotheses assumed on the generic function that codes the selection strategy.

We would like to point out some lines for future work. We plan to continue this work following several research lines. First, we will extend the $\mathcal{ALC}$ –reasoners to manage concept satisfiability with respect to terminological Boxes. Second, we will deal with more specific algorithms which optimize the performing by richer data structures. Also, we are interested in extending the reasoners for the $\mathcal{ALC}$ logic to other description logics, incrementally approaching us to the description logic $\mathcal{SHOIN}$, which is the description logic corresponding to OWL–DL.

References

[1] J. A. Alonso, J. Borrego, M. J. Hidalgo, F. J. Martín and J. L. Ruiz, *A formally verified prover for the ALC description logic*, LNCS **4732**, Springer (2007), 135–150.

[2] 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** (2004), 3–16.

[3] F. Baader, D. McGuiness, D. Nardi and P. Patel–Schneider, *The Description Logic Handbook: Theory, Implementation and Applications*, Cambridge University Press (2003).

[4] F. Baader and B. Hollunder, *A Terminological Knowledge Representation System with Complete Inference Algorithms.*, PDK (1991), 67–86.

[5] A. Dold, "Formal Software Development using Generic Development Steps, " Ph.D. thesis, Ulm University, 2000.

[6] V. Haarslev and R. Möller, *RACER System Description*, LNCS **2083**, Springer (2001), 701–705.

[7] C. B. Jones, *Systematic Software Development using VDM*, Prentice–Hall International, (1990).

[8] C. Muñoz, *PVSio Reference Manual Version 2.b*, National Institute of Aerospace (2005).

[9] S. Owre, J. M. Rushby and N. Shankar, *PVS: A Prototype Verification System*, LNAI **607**, Springer 1992, 748–752.

[10] S. Owre and N. Shankar, *Abstract Datatype in PVS*, Computer Science Laboratory, SRI International, (1997).

[11] E. Sirin and B. Parsia, *Pellet System Description*, Proceedings of the 2006 International Workshop on Description Logics (DL2004), CEUR Workshop Proceedings **189** (2006).

[12] D. Tsarkov and I. Horrocks, *FaCT++ Description Logic Reasoner: System Description*, LNAI **4130**, Springer (2006), 292–297.