Formally Verified Tableau-Based Reasoners for a Description Logic

M. J. Hidalgo-Doblado, J. A. Alonso-Jiménez, J. Borrego-Díaz, F. J. Martín-Mateos, J. L. Ruiz-Reina

Abstract Description Logics are a family of logics used to represent and reason about conceptual and terminological knowledge. One of the most basic description logics is $\mathcal{ALC}$, used as a basis from which to obtain others. Description logics are particularly important to provide a logical basis for the web ontology languages (such as OWL) used in the Semantic Web. In order to increase the reliability of the Semantic Web, formal methods can be applied, and in particular formal verification of its reasoning services can be carried out. In this paper, we present the formal verification of a tableau-based satisfiability algorithm for the logic $\mathcal{ALC}$. The verification has been completed in several stages. First, we develop an abstract formalization of satisfiability-checking of $\mathcal{ALC}$-concepts. Secondly, we define and formally verify a tableau-based algorithm in which the order of rule application and branch selection can be flexibly specified, using a methodology of refinements to transfer the main properties from the $\mathcal{ALC}$ abstract formalization. Finally, we obtain verified and executable reasoners from the algorithm via a process of instantiation.

Partially supported by TIN2009-09492 Project of *Ministerio de Ciencia e Innovación* and Excellence Project TIC-06064 of *Junta de Andalucía*, co-financed with FEDER founds.

M. J. Hidalgo-Doblado (✉) · J. A. Alonso-Jiménez · J. Borrego-Díaz · F. J. Martín-Mateos · J. L. Ruiz-Reina
Computational Logic Group, Department of Computer Science and Artificial Intelligence, University of Seville, E.T.S. Ingeniería Informática, Avda. Reina Mercedes s.n., 41012 Sevilla, Spain
e-mail: mjoseh@us.es

J. A. Alonso-Jiménez
e-mail: jalonso@us.es

J. Borrego-Díaz
e-mail: jborrego@us.es

F. J. Martín-Mateos
e-mail: fjesus@us.es

J. L. Ruiz-Reina
e-mail: jruiz@us.es

Keywords Semantic web · Description logics · Verification · Formal methods

1 Introduction

Description Logics [6] are a family of logics used to represent conceptual and terminological knowledge. Recently, Description Logics have gained importance, since they are the logical basis for web ontology languages (OWL DL, OWL Full,...), used in the Semantic Web. Description Logics reasoners such as RACER, Pellet and FaCT++ [13, 29, 31] are being used to process knowledge in the Semantic Web. The logic $\mathcal{ALC}$ is a basic one, which can be extended to more expressive description logics, such as $\mathcal{SHOIN}$, that are used for reasoning in web ontology languages.

The formal verification of reasoning systems for the Semantic Web would increase the reliability of the Semantic Web itself, and thus is an interesting area for the application of formal methods. In this regard, our main goal in the long term is the formal verification of reasoning systems for Description Logics. We plan to tackle this task incrementally, approaching to the description logic $\mathcal{SHOIN}$ from the logic $\mathcal{ALC}$. As a first step, in this paper we show how to construct formally verified tableau-based reasoners for the description logic $\mathcal{ALC}$. The formalization follows the first chapters of [30].

To formally verify reasoning systems for Description Logics, several possible approaches were considered. One approach could be to concentrate on a specific reasoner and prove its correctness. In this case, we would have to formally develop the full logical theory necessary for its specification and verification. However, this theory will essentially be determined by the reasoner and it might not be suitable for other reasoners. An alternative approach, which we have taken, is to start from an abstract formalization of the logic $\mathcal{ALC}$ and gradually refine the specification for different formally verified reasoners. In every refinement step, some generic features are set, preserving the main properties already verified.

Our formalizations are carried out in the PVS system [22]. This system combines an expressive specification language with an interactive theorem prover. 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 [20].

Before presenting the details of the developed formalization, we are going to explain the methodology we have followed, illustrated in Fig. 1.

1. *Abstract formalization of $\mathcal{ALC}$ tableaux* (in the figure, the square in the root of the tree). We first formalize an abstract specification in PVS to check satisfiability of $\mathcal{ALC}$ -concepts, based on a set of transformation rules (also called *completion* rules). This transforms the satisfiability problem to an equivalent one, and we prove its properties of termination, soundness and completeness. For this reason, we use abstract, not evaluable, PVS types. The features of this specification make it feasible to obtain proofs close to the corresponding ones usually found in the literature of Description Logics. Termination turned out to be the most difficult property to prove.
2. *Generic tableaux* (in the figure, octagons). Then we implement a generic tableaux-based satisfiability checking algorithm in PVS corresponding to the abstract specification, in such a way that the correctness properties of this algorithm

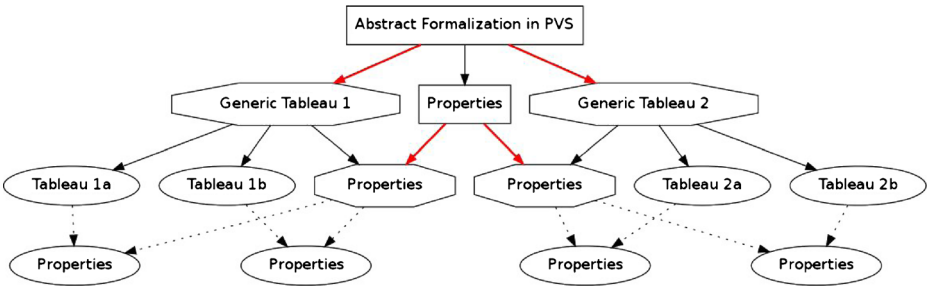

Fig. 1 The roadmap

are based on the same properties already proved for the abstract formalization. For this purpose, we use the methodology of refinements [2, 12]. The tableaux algorithm is based on the application of a set of transformation rules and it is generic in the sense that we leave some strategies of application of these rules unspecified. The correctness of this algorithm has been proved, assuming some generic hypotheses about the functions that implement the strategies of rule application.

3. *Concrete tableaux* (in the figure, ellipses). Finally, we develop concrete executable tableaux algorithms for the logic $\mathcal{ALC}$ in PVS, which are considered as instances of the generic tableaux algorithm. In order to do this, it its sufficient to instantiate the rule application strategies by concrete heuristic functions. We emphasize that in order to prove the correctness of these reasoners, we only have to provide instances with the assumed hypotheses, for each concrete heuristic function.

During the development of this work, two issues emerged, each one important in itself. The first one was addressed when we tackled the termination proof. It is known that the multiset ordering well-foundedness is an useful result needed for many termination proofs, as was proved by Dershowitz and Manna in [11]. Thus, we have extended the multiset library of PVS to include the well-foundedness of the multiset order. We use this property and a measure function, as it is described in [21], to prove the termination of the satisfiability algorithm developed for the logic $\mathcal{ALC}$. The second issue is the use of a methodology of refinements, motivated by the choice to move from an abstract formalization to specific reasoners, preserving the main properties.

The paper is structured as follows: Section 2 is devoted to presenting an overview of the PVS system. Section 3 shows a PVS formalization of an abstract tableau-based satisfiability checking of $\mathcal{ALC}$ -concepts, with a description of proofs of termination, soundness and completeness. In Section 4, we present a sketch of the type refinement and operator refinement techniques developed in PVS. Section 5 is devoted to describing the construction of a generic algorithm corresponding to the specification of the $\mathcal{ALC}$ abstract formalization, presented in Section 3. In Section 5, a particular reasoner is constructed by choosing an application strategy of transformation rules and by instantiating the non-interpreted types used in the former specification. Finally, in the last section we draw some conclusions and suggest some lines of future work.

This paper is an extended and revised version of the work previously reported in [3, 15], presented in a unified way. Compared to these papers, the way in which we introduce the rule application strategy is extended, not only permitting us to define the next completion rule to be applied, but also the order in which to explore the disjunctive branches.

The entire formalization consists of a large collection of PVS definitions and theorems. We encourage the interested reader to consult all the PVS theories that have been developed, available at http://www.glc.us.es/theories/alc.

2 Overview of PVS

In this section, we present a brief description of the PVS language and prover, introducing some of the notions used in this paper.

The *Prototype Verification System* PVS is a general-purpose environment for developing specifications and proofs. The PVS specification language is based on a classical typed higher-order logic with predefined types including booleans, integers, reals,.... New types can be obtained by means of the function type constructor `[D -> R]`, the product type constructor `[A, B]` or record types `[#...#]`.

The type system is also augmented with *subtypes*, *dependent types* and *abstract data types*, among others. In particular, a predicate subtype `{x:T | p(x)}` consists of all the elements of type `T` verifying the predicate `p`. If x is already declared to be a variable of type T, this subtype is also denoted as `(p(x))`. Predicate subtypes are used to constrain domains and ranges of functions in a specification and, therefore to define partial functions.

In general, type-checking with predicate subtypes is undecidable. Therefore, the type-checker generates proof obligations, called *type correctness conditions* (TCCs). Those TCCs are either discharged by specialized proof strategies or have to be proved by the user before the theory can be considered type checked. In particular, to define a recursive function, it must be ensured that the function terminates. For this purpose, the user has to provide a *measure function* in the definition of a recursive function. This generates a TCC stating that the measure function, applied to the arguments decreases in every recursive call with respect to a well-founded ordering (i.e. an ordering with no infinitely descending chain).

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* them. A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts for a large number of theories.

In the following sections, we will explain some more features of PVS, as we describe the formalization. For complete information on PVS we refer the reader to [22] and to the documentation available at http://www.pvs.csl.sri.com.

3 Abstract Formalization of Tableau-Based Satisfiability Checking of $\mathcal{ALC}$-Concepts

In this section we present an abstract formalization in PVS of satisfiability checking of $\mathcal{ALC}$-concepts, into which different tableau-based algorithms may fit. These

algorithms are usually defined by a set of transformation rules that are applied to sets of assertional axioms, until some "simple form" is reached. Since one of our goals in the development of this abstract formalization is to achieve a high degree of generality, we have specified the transformations in a declarative way. Thus, we do not specify how a rule is applied, but rather the relation between the sets of assertional axioms before and after the application of a rule. This relation holds independently of any concrete and evaluable subsequent rule specifications. Another goal of our development is to obtain PVS proofs that closely resemble the proofs found in the 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. Since sets allow more abstract reasoning than lists, we have mainly used the type of finite sets. Thus, the proof development is done over sets and it is not determined by the datatype used to represent finite sets.

We now describe the basic components of the logic $\mathcal{ALC}$, and how we have formalized satisfiability checking for this logic. We present the logic $\mathcal{ALC}$ along with its formalization in PVS, as well as the proofs of its termination, soundness and completeness.

3.1 Syntax and Semantics of the $\mathcal{ALC}$ Logic

Let NC be a non-empty 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 $R \in \mathsf{NR}$):

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

For example, if Animal, Mammal and Female are concept names, then the expression ¬Mammal ⊔ (Mammal ⊓ Female) represents those individuals that are non-mammals or female mammals. If haveBaby is a role name then Animal ⊓ ∀haveBaby.Mammal denotes those animals all of whose babies are mammals.

The set of $\mathcal{ALC}$ -concepts can be represented in PVS as a recursive datatype, using the mechanism for defining abstract datatypes [23], and specifying 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 induction principles for this datatype. In particular, this theory contains the built-in relation `subterm` (specifying the notion of *subconcept*) and the well-founded relation <<, that will be useful to make recursive definitions on concepts.

Description Logics systems organize knowledge in two categories: *terminological knowledge*, which establishes relationships between concepts, and *assertional*

knowledge, which establishes the relationship between individuals, roles and concepts. A *knowledge base* consists of a finite set of terminological axioms (called TBox) and a finite set of assertional axioms (called ABox).

Terminological axioms have the forms $C \sqsubseteq D$, $C \equiv D$, where C and D are concepts. Axioms of the first kind are called *inclusions*, while axioms of the second kind are called *equations*. In PVS, we specify terminological axioms and TBoxes as follows:

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

TBox: TYPE = finite_set[alc_gtax]
```

Let us now introduce 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 $x{:}\,C$ and $(x, y){:}\,R$ are called *assertional axioms*. The first kind of assertional axiom is intended to represent that x is in C (or that x is an *instance* of C); and the second kind is intended to represent that x and y are related by R. In PVS:

```
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. Specifically, we use a record type of PVS (record types have the form $[\#a_1 : t_1, ..., a_n : t_n\#]$, where the a_i are the accessors and the t_i are their types).

```
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 we consider a non-empty type `U` and we require the domain of an interpretation to be a non empty set of elements in `U`. In general, `PRED[T]` represents the type of predicates defined on `T`, so in this case `PRED[[(int_domain),(int_domain)]]` represents the type of binary relations on `(int_domain)`. Note also that some of the type components can depend on previous components: this is possible in PVS due to its ability to deal with dependent types.

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

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

We have specified $C^{\mathcal{I}}$ in PVS, as usual by recursion on C, using a well-founded relation <<. This PVS specification is close to the definition of interpretation above, due mainly to the fact that we have chosen sets for our abstract specifications.

```
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 <<
```

Note that `int_names_concept(I)` is the function that assigns to every concept name the corresponding subset of the domain of `I`, according to the interpretation `I`. Similarly for `int_names_roles(I)` and role names.

Regarding the terminological knowledge, we say that an interpretation $\mathcal{I}$ *satisfies* $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and it *satisfies* $C \equiv D$ if $C^{\mathcal{I}} = D^{\mathcal{I}}$. We say that $\mathcal{I}$ *satisfies* a TBox $\mathcal{T}$ (or it is a *model* of $\mathcal{T}$) if it satisfies every axiom of $\mathcal{T}$. As for the assertional axioms, $\mathcal{I}$ *satisfies* $x{:}C$ if $x^{\mathcal{I}} \in C^{\mathcal{I}}$ and it satisfies $(x, y){:}\, R$ if $(x^{\mathcal{I}}, y^{\mathcal{I}}) \in R^{\mathcal{I}}$. Analogously, an interpretation *satisfies* the ABox $\mathcal{A}$ (or is a *model* of $\mathcal{A}$) if it satisfies every axiom in $\mathcal{A}$. In that case, we say that $\mathcal{A}$ is called *satisfiable*.

The previous notions naturally bring up some standard inference problems for Description Logics systems. For example, interesting problems include subsumption checking or satisfiability checking: given a TBox $\mathcal{T}$ and two concepts C and D, we say that C is *subsumed* by D with respect to $\mathcal{T}$ (denoted as $C \sqsubseteq_{\mathcal{T}} D$), if for every model $\mathcal{I}$ of $\mathcal{T}$, $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; we say that C is *satisfiable* with respect to $\mathcal{T}$ if there exists a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}}$ is nonempty (in that case, we also say that $\mathcal{I}$ is a *model* of C with respect to $\mathcal{T}$). If $\mathcal{T}$ is the empty box, we simply say that C is *satisfiable*.

Several results regarding how some inference problems can be reduced to other ones can be proved [5, 30], and we have done it in PVS. For example, we have the following results:

– Reducing unsatisfiability to subsumption: a concept C is unsatisfiable with respect to a *TBox* $\mathcal{T}$ iff $C \sqsubseteq_{\mathcal{T}} \bot$ (where $\bot$ is the *bottom* concept).

– Reducing subsumption to unsatisfiability: $C \sqsubseteq_{\mathcal{T}} D$ iff $C \sqcap \neg D$ is unsatisfiable with respect to *TBox* $\mathcal{T}$.

Normally, reasoning tasks are reduced to checking satisfiability with respect to a TBox. Moreover, it can be proved that, under certain conditions, this can be reduced to an equivalent satisfiability problem where the TBox is empty [5, 30]. Thus, it is possible to reduce some reasoning tasks to check satisfiability of a concept with respect to the empty TBox. This is our PVS specification of the notion of concept satisfiability:

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

3.2 Deciding Concept Satisfiability for $\mathcal{ALC}$

A tableau algorithm for $\mathcal{ALC}$ attempts 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 only consider concepts in negation normal form (NNF), a form in which negations appear only in front of concept names. This does not imply a restriction, since it is easy to specify a PVS function computing for each $\mathcal{ALC}$ -concept an equivalent one (i.e., with the same models) in NNF.

An ABox $\mathcal{A}$ contains a *clash* if, for some individual name $x \in \mathsf{NI}$ and concept's names $A \in \mathsf{NC}$, $\{x{:}\, A, x{:}\, \neg A\} \subseteq \mathcal{A}$. Otherwise, $\mathcal{A}$ is called *clash-free*. To test the satisfiability of an $\mathcal{ALC}$ -concept C in NNF, the main idea is to start from the initial ABox $\{x_0{:}\, C\}$ and iteratively apply 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	$\mathcal{A} \rightarrow_{\forall} \mathcal{A} \cup \{y{:}\, D\}$

We say that an ABox is *complete* when none of the above rules can be applied to it. Rule application stops when reaching a complete ABox or when a clash has been generated. Note that the rules preserve satisfiability and that it is easy to derive a model from a complete and clash-free ABox. Thus the algorithm answers "C is satisfiable" if such an ABox has been generated. Otherwise, if no complete and clash-free ABox can be obtained by applying the rules, then it answers "C is unsatisfiable".

These completion rules can be seen as a production system. Nevertheless, we have not formalized them in a functional way, but by following a more declarative style, defining the completion rules in PVS as binary relations between ABoxes. This declarative style allows us to clearly separate logic from control. We can concentrate first on the properties that only depend on the relation between boxes, and later deal with specific strategies for rules application. As we will see, this will benefit our refinement approach.

For example, $\mathcal{A}_1 \rightarrow_{\sqcap} \mathcal{A}_2$ if there exists an assertional axiom $x{:}\, C \sqcap D$ in $\mathcal{A}_1$ such that $\{x{:}\, C, x{:}\, D\} \not\subseteq \mathcal{A}_1$ and $\mathcal{A}_2 = \mathcal{A}_1 \cup \{x{:}\, C, x{:}\, D\}$

```
and_step(AB1, AB2): bool =
 EXISTS Aa: member(Aa,AB1) AND
            instanceof?(Aa) AND
            alc_and?(right(Aa)) AND
            LET x = left(Aa), C = conc1(right(Aa)), D = conc2(right(Aa))
             IN (NOT member(instanceof(x, C), AB1) OR
                 NOT member(instanceof(x, D), AB1)) AND
                 AB2 = add(instanceof(x, C),
                           add(instanceof(x, D), AB1))
```

Once all 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 (defined by `contains_clash`) and $\mathcal{A}_2$ is obtained from $\mathcal{A}_1$ by the application of a completion rule:

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

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))
```

Note 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 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 $\rightarrow$. In PVS, the reflexive and transitive closure of a relation $<$ can be specified by an inductive definition:

```
rtr_cl(<)(x,y): INDUCTIVE bool =
   x = y OR EXISTS z: rtr_cl(<)(x,z) AND z < y
```

Due to this inductive definition, PVS automatically creates two induction schemes (weak and strong [24]) for `rtr_cl` that will be used in subsequent proofs. Furthermore, if we fix an ABox $\mathcal{A}_1$, the relation *expansion* can be currified

```
is_expansion(AB1)(AB2): bool = rtr_cl(successor)(AB2, AB1)
```

To illustrate the completion process, the following example shows the application of some completion rules to an initial ABox $\{x_0{:}\, C\}$ until a complete and clash-free ABox is reached.

Example 1 Let C be the concept $\forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F))$. Then,

$$
\begin{aligned}
& \mathcal{A}_0 := \{x_0{:}\, \forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F)) \\
\stackrel{*}{\rightarrow}\ & \mathcal{A}_1 := \mathcal{A}_0 \cup \{x_0{:}\, \forall R.D, x_0{:}\, \exists R.(D \sqcup E), x_0{:}\, \exists R.(D \sqcup F)\} \\
\rightarrow\ & \mathcal{A}_2 := \mathcal{A}_1 \cup \{(x_0, x_1){:}\, R, x_1{:}\, D \sqcup E\} \\
\rightarrow\ & \mathcal{A}_3 := \mathcal{A}_2 \cup \{x_1{:}\, D\} \\
\rightarrow\ & \mathcal{A}_4 := \mathcal{A}_3 \cup \{(x_0, x_2){:}\, R, x_2{:}\, D \sqcup F\} \\
\rightarrow\ & \mathcal{A}_5 := \mathcal{A}_4 \cup \{x_2{:}\, D\}
\end{aligned}
$$

Once the expansion relation is defined, we use it to specify the notion of consistency: an ABox $\mathcal{A}$ 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. This is the corresponding PVS specification:

```
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)

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

This definition provides the PVS specification for an abstract tableau-based algorithm for deciding the satisfiability of $\mathcal{ALC}$-concepts. The two kinds of non–determinism in it may be pointed out: 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). To prove that such an algorithm would be correct, we have to establish termination, soundness and completeness of this specification. In the following subsections we describe the corresponding PVS proofs.

We end this subsection with a comment on a technical question regarding individual names. First, note that to specify the existential rule, a fresh individual has to be available each time. In PVS we declare NI as a non-empty type, but we have to ensure that this type has all individuals we will need. Thus, we include in the assuming part of the theory the following assumption:

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

When we import this theory, the type-checker will generate the corresponding type correctness condition. In our development, there are some theories in which the type NI is an uninterpreted type. Thus, the TCC generated still cannot be proved. When we will make theories in which we will have evaluable procedures, we will instantiate NI by `nat` and the assumption will be proved.

Second, as we stated in the last subsection, we check satisfiability of a concept C by checking that $\{x_0{:}\,C\}$ is satisfiable, where x_0 is an individual that we fix in the parameters of the PVS theory. Again, this individual will only be instantiated in the theory in which we will specify the evaluable procedure. Then, NI will be instantiated by `nat` and x_0 by 0.

3.3 Soundness

The soundness of the completion process means proving that if C is consistent (i.e., $\{x_0{:}\,C\}$ has a complete and clash-free expansion), then C is satisfiable (i.e., C has a model):

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

The PVS proof is based on the following steps:

1. If $\mathcal{A}$ is a complete and clash-free expansion of an initial ABox $\mathcal{A}_0$, then $\mathcal{A}$ is satisfiable. We have proved this by specifying in PVS the canonical interpretation $\mathcal{I}_\mathcal{A}$ associated with $\mathcal{A}$. That is, $\mathcal{I}_\mathcal{A}$ is defined as follows:
   - The domain of the interpretation consists of the individuals that are in $\mathcal{A}$ together with x_0:
   $$\Delta^{\mathcal{I}_\mathcal{A}} = \{x \in NI \mid x \text{ occurs in } \mathcal{A}\} \cup \{x_0\}$$
   - Interpretation of concept names:
   $$A^{\mathcal{I}_\mathcal{A}} = \{x \mid x : A \in \mathcal{A}\}$$
   - Interpretation of role names:
   $$R^{\mathcal{I}_\mathcal{A}} = \{(x, y) \mid (x, y) : R \in \mathcal{A}\}$$
   - Interpretation of individuals: the individuals that occur in $\mathcal{A}$ are interpreted as themselves, and the other ones are interpreted as x_0.

   Then, we prove that $\mathcal{I}_\mathcal{A}$ is a model of $\mathcal{A}$.
2. If $\mathcal{A}_1 \rightarrow \mathcal{A}_2$ and $\mathcal{A}_2$ is satisfiable, then $\mathcal{A}_1$ is satisfiable as well (see the PVS lemma `abox_satisfiable_successor` in the source code).
3. Now, if C is consistent, by definition the ABox $\{x_0 : C\}$ is consistent and therefore it has a complete and clash-free expansion $\mathcal{A}$; by definition of expansion, this means that $\{x_0 : C\} \xrightarrow{*} \mathcal{A}$. We prove by induction that this implies that $\{x_0 : C\}$ is satisfiable. Steps 1 and 2 imply, respectively, the base case and the inductive case in this proof by induction.
   The following is the PVS proof of this last result, where `rtr_cl_weak_induction` is the weak induction scheme generated by the inductive definition of the reflexive and transitive closure `rtr_cl`:

```
("" 
 (expand "is_expansion")
 (skolem 1 ("_" "AB2"))
 (rewrite "rtr_cl_weak_induction")
 (hide 2)
 (skosimp)
 (ground)
 (skosimp)
 (forward-chain "abox_satisfiable_successor"))
```

3.4 Termination

Given a concept C in negation normal form, we define $\mathcal{E}(C)$ as the set of the expansions of the initial ABox $\{x_0 : C\}$, function `expansion_abox_concept` in PVS:

```
expansion_abox_concept(C:(is_nnf?)): TYPE =
  (is_expansion(singleton(instanceof(x_0, C))))
```

In order to verify the termination of an $\mathcal{ALC}$-algorithm that applies the completion rules above, it suffices to prove that the successor relation, defined on the set $\mathcal{E}(C)$ is well-founded:[1]

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

Our PVS proof of the well-foundedness of the successor relation is based on the following embedding lemma (see [3] for a PVS proof): if $f : (S, <') \to (T, <)$ is monotone and $(T, <)$ is well-founded, then $(S, <')$ is well-founded. Thus, for our purposes it suffices to show the existence of a type T, a well-founded relation $<$ on T, and a *measure function* $\mu_C : \mathcal{E}(C) \to T$ such that:

$$(\forall \mathcal{A}_1, \mathcal{A}_2)[\mathcal{A}_1 \to \mathcal{A}_2 \Rightarrow \mu_C(\mathcal{A}_2) < \mu_C(\mathcal{A}_1)] \tag{1}$$

The formalization of this proof in PVS has been carried out in two stages. In the first one, we have assumed the existence of a measure function μ_C (called `measure_concept`), with T, $<$ and μ_C above as parameters of the PVS theory, and property (1) above as an assumption in its body:

```
th[..., C: (is_nnf?),
        T: TYPE+, <: (well_founded?[T]),
        measure_concept: [expansion_abox_concept(C) -> T]]: THEORY

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

From this and the embedding lemma, we have proved that the successor relation is well-founded on $\mathcal{E}(C)$.

In the second stage, we proved the existence of a measure function, following a definition given in [21]. The main idea is to take 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[2] of the lexicographic order on $\mathbb{N} \times \mathbb{N}$, that we will denote by $<_{\text{mult}}$, to $\mathcal{M}(\mathbb{N} \times \mathbb{N})$. In order to formalize that construction in PVS, we needed to carry out two main subtasks:

1. First, we extended the PVS multiset library, proving the following well-known theorem by Dershowitz and Manna [11]: if $<$ is a transitive and well-founded relation on T, then $<_{\text{mult}}$ is a well-founded relation on $\mathcal{M}(T)$ (see [3]). This means that in order to prove the well-foundedness of the extension to $\mathcal{M}(\mathbb{N} \times \mathbb{N})$ of the lexicographic order on $\mathbb{N} \times \mathbb{N}$ it suffices to instantiate, in the previous

[1] In PVS, $p[T](R)$ means that the relation R, restricted to type T, verifies the predicate p. In this case, the above theorem states that the relation `successor`, restricted to $\mathcal{E}(C)$, verifies `well_founded?`

[2] In general, given a set A, we denote $\mathcal{M}(A)$ the set of finite multisets (or *bags*) of elements of A; if $<$ is an ordering on A, we denote as $<_{\text{mult}}$ its *multiset extension* on $\mathcal{M}(A)$, defined as $M <_{\text{mult}} N$ iff there exists X and Y such that $\emptyset \neq X \subseteq M$, $N = (M \setminus X) \uplus Y$ and $\forall y \in Y \exists x \in X$ such that $x > y$.

theorem, T by $\mathbb{N} \times \mathbb{N}$ and $<$ by the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$ (which we also proved in PVS to be well-founded).
2. Second, we have defined a function μ_C mapping each expansion $\mathcal{A} \in \mathcal{E}(C)$ to a multiset of pairs, such that $\mu_C(\mathcal{A}_2) <_{\text{mult}} \mu_C(\mathcal{A}_1)$ when $\mathcal{A}_1 \to \mathcal{A}_2$. For the sake of clarity, we devote Section 3.6 to a detailed explanation of this result.

3.5 Completeness

Completeness means to prove that if a concept C is satisfiable, then C is consistent (i.e. $\{x_0{:}\,C\}$ has a complete and clash-free expansion):

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

The PVS proof is achieved by means of three main results:

1. If $\mathcal{A}$ is a satisfiable ABox, then $\mathcal{A}$ is clash-free.

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

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

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

   This result is proved by showing that for every satisfiable and not complete ABox $\mathcal{A}$, for every rule r, and for every axiom of $\mathcal{A}$ to which r can be applied, there exists an axiom such that adding that axiom to $\mathcal{A}$, we obtain a satisfiable ABox.
3. If C is satisfiable, then the ABox $\{x_0{:}\,C\}$ is satisfiable. So the completeness theorem will be a corollary of the following main lemma: if $\mathcal{A} \in \mathcal{E}(C)$ is satisfiable, then it has a complete and clash-free expansion.

   ```
   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 proved by well-founded induction on the successor relation. Note that in the PVS prelude, the following induction scheme is proved for every well-founded relation < on a type T:

   ```
   wf_induction: LEMMA
     (FORALL (p:pred[T]):
       (FORALL (x:T):
         (FORALL (y:T): y<x IMPLIES p(y)) IMPLIES p(x))
       IMPLIES (FORALL (x:T): p(x)))
   ```

That is, in our case it suffices to prove that for every satisfiable $\mathcal{A}$ such that all its successors have a complete and clash-free expansion, then $\mathcal{A}$ has also a complete and clash-free expansion. This can be proved by distinguishing two cases: if $\mathcal{A}$ is complete, then by 1 above $\mathcal{A}$ is itself its complete and clash-free expansion. Otherwise, by 2, there is a satisfiable ABox $\mathcal{A}'$, which is a successor of $\mathcal{A}$; by induction hypothesis, $\mathcal{A}'$ has a complete and clash-free expansion, which is also an expansion of $\mathcal{A}$. A fragment of the PVS proof of this main lemma in which the main interactions with the prover can be observed is:

```
1  ("" 
2   (induct "AB" :name
3    "wf_induction[expansion_abox_concept(C), successor]")
4  (("1"
5     (skosimp*)
6     (case "complete(x!1)")
7     ((("1"
8       (forward-chain "abox_satisfiable_implies_clash_free")
9       ...
10     ("2"
11      (forward-chain "abox_satisfiable_successor_vice")
12      ...
13    ("2" (rewrite "well_founded_succesor")))
```

3.6 Measure on $\mathcal{ALC}$-Expansions of a Concept

As we said in Section 3.4, we now explain in detail the definition of the measure function μ_C on the set of expansions of C, and the proof of its monotonicity with respect to the successor relation and $<_{\text{mult}}$ (property (1) of Section 3.4). Following [21], the main idea is that given $\mathcal{A} \in \mathcal{E}(C)$ we construct a multiset of pairs of natural numbers; each of these pairs is a measure of a possible *activation* (i.e. a representation of a possible application of a completion rule to $\mathcal{A}$).

Let us present an intuitive idea of the measure. First, note that the existential rule is the only rule that introduces new individuals in the ABox. These individuals constitute a tree, considering that y is a successor of x in the tree when (x, y): $R \in \mathcal{A}$ for some role R. Thus, for each individual y in an expansion of an initial ABox, there exists a path from x_0 to y. This allows us to define the notion of *level* (or depth) of y in the tree. Furthermore, we define the *size* of a concept D (denoted as $|D|$) as the number of symbols that it needs to be constructed (it is easily defined by recursion on the structure of a concept). Notice that if y: $D \in \mathcal{A}$ ($\mathcal{A}$ being an expansion of $\{x_0 : C\}$), then, $level(y) + |D| \leq |C|$. Therefore, we can define the *colevel* of y as $|y|_{\mathcal{A}} = |C| - level(y)$. Moreover, if z is a successor of y in the ABox $\mathcal{A}$ then $|z|_{\mathcal{A}} = |y|_{\mathcal{A}} - 1$.

We assign a multiset of pairs of natural numbers to an expansion $\mathcal{A} \in \mathcal{E}(C)$ as follows:

- for every axiom y: D in $\mathcal{A}$, such that D has the form $\sqcap$, $\sqcup$ or $\exists$ and if the corresponding rule is applicable, we put the pair $(|y|_{\mathcal{A}}, |D|)$ into the multiset.
- for every axiom y:$\forall R.D$ in $\mathcal{A}$ and each z such that (y, z): $R \in \mathcal{A}$ and $\mathcal{A}$ does not contain z: D, we put the pair $(|z|_{\mathcal{A}}, |\forall R.D|)$ into the multiset.

Example 2 The following table shows the measures associated with each of the expansions in Example 1, where $C = \forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F))$:

	$\mathcal{A}$	measure($\mathcal{A}$)
	$\mathcal{A}_0 := \{x_0{:}\forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F))\}$	$\dot{\{}(15, 15)\dot{\}}$
$\overset{*}{\rightarrow}$	$\mathcal{A}_1 := \mathcal{A}_0 \cup \{x_0{:}\forall R.D, x_0{:}\exists R.(D \sqcup E), x_0{:}\exists R.(D \sqcup F)\}$	$\dot{\{}(15, 5), (15, 5)\dot{\}}$
$\rightarrow_\exists$	$\mathcal{A}_2 := \mathcal{A}_1 \cup \{(x_0, x_1){:}\, R, x_1{:}\, D \sqcup E\}$	$\dot{\{}(14, 3), (14, 3), (15, 5)\dot{\}}$
$\rightarrow_\forall$	$\mathcal{A}_3 := \mathcal{A}_2 \cup \{x_1{:}\, D\}$	$\dot{\{}(15, 5)\dot{\}}$
$\rightarrow_\exists$	$\mathcal{A}_4 := \mathcal{A}_3 \cup \{(x_0, x_2){:}\, R, x_2{:}\, D \sqcup F\}$	$\dot{\{}(14, 3), (14, 3)\dot{\}}$
$\rightarrow_{\sqcup_1}$	$\mathcal{A}_5 := \mathcal{A}_4 \cup \{x_2{:}\, D\}$	$\dot{\{}\dot{\}}$

Let us explain the first three rule applications in this example. In the initial ABox $\mathcal{A}_0 := \{x_0{:}C\}$, only the rule $\rightarrow_\sqcap$ can be applied to $\mathcal{A}_0$. So, since $|C| = 15$ and $level(x_0) = 0$, the associated pair is $(15, 15)$ and the measure of $\mathcal{A}_0$ is the multiset $\dot{\{}(15, 15)\dot{\}}$. When the rule $\sqcap$ is applied twice, the following instance axioms $\{x_0{:}\forall R.D, x_0{:}\exists R.(D \sqcup E), x_0{:}\exists R.(D \sqcup F)\}$ are added to the ABox, obtaining $\mathcal{A}_1$.

Then, the $\rightarrow_\sqcap$ rule is no longer applicable but the $\rightarrow_\exists$ rule is applicable to two axioms.[3] Therefore, the pair $(15, 15)$ is replaced by the pairs $(15, 5)$ and $(15, 5)$ in the multiset. If we apply the $\rightarrow_\exists$ rule to the axiom $x_0{:}\exists R.(D \sqcup E)$, obtaining $\mathcal{A}_2$, a new individual x_1 is introduced, a successor of x_0 in the associated tree. Thus, the colevel of x_1 in $\mathcal{A}_2$ is 14. Now, there are three applicable rules, corresponding to the axioms $x_0{:}\exists R.(D \sqcup F)$, $x_1 \in D \sqcup E$ and $x_0{:}\forall R.D$ (using x_1 as witness), and thus the associated multiset is $\dot{\{}(14, 3), (14, 3), (15, 5)\dot{\}}$.

Applying the $\rightarrow_\sqcup$ rule, we obtain $\mathcal{A}_3$, adding $x_1 \in D$. This means that now the $\rightarrow_\sqcup$ rule and even the $\rightarrow_\forall$ rule are no longer applicable and thus the associated multiset is $\dot{\{}(15, 5)\dot{\}}$ (note how sometimes the application of a rule not only disables its own applicability but also that of other rules).

Let us explain now how we define the measure in PVS. First we define the notions of level and colevel. In order to do this, we used the PVS library of graphs [8], in which a graph is specified as a record type with two components: the vertices and the edges. These are represented by sets with two different elements (type `doubleton[T]`):

```
pregraph:TYPE = [# vert:finite_set[T], edges:finite_set[doubleton[T]] #]

graph:TYPE = {g:pregraph | (FORALL (e:doubleton[T]): edges(g)(e) IMPLIES
                                (FORALL (x:T): e(x) IMPLIES vert(g)(x)))}
```

[3] Note that the $\rightarrow_\forall$ rule cannot still be applied to $\{x_0{:}\forall R.D\}$, since there are no individuals related by R to x_0.

We define the *graph associated* 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 sets $\{x, y\}$, such that (x, y): $R \in \mathcal{A}$ for some role R, being x and y different individuals:

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

Here `occur_ni` specifies the set of individuals appearing in an ABox and `dbl_assoc_abox` the doubletons of individuals related in it. The notation `(# .. #)` in PVS is for defining specific instances of a record type (in this case, the `pregraph` type).

From this definition, we prove the well-known result stating 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 its colevel $|x|_{\mathcal{A}}$ as the difference between the size of the concept 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)
```

In addition, we prove that the colevel of an individual in $\mathcal{A}$ remains invariant after the application of completion rules:

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

and that if y is a *successor of x with respect to R* in $\mathcal{A}$ (i.e., (x, y): $R \in \mathcal{A}$), then $|y|_{\mathcal{A}} = |x|_{\mathcal{A}} - 1$:

```
successor_related(AB)(y,x):bool = EXISTS R: member(related(x,R,y),AB)

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

Both properties are essential for proving the monotonicity of μ_C.

As we have already mentioned, the elements of the multiset associated to an expansion $\mathcal{A}$ should measure all rules applicable to $\mathcal{A}$. Note that the applicability of a rule is completely characterized by the axiom of $\mathcal{A}$ to which it is applied, except in the case of the $\rightarrow_\forall$ rule, in which in addition we have to specify the individual used as well. Thus, in order to capture the notion of applicability of a rule, we introduce the type *activation* (`activ`), whose elements are structures consisting of an instance axiom and an individual, called the *witness* (actually, this individual will only make sense for $\rightarrow_\forall$ rules). We also specify when an activation is applicable to an ABox $\mathcal{A}$ and we define the *agenda* of $\mathcal{A}$, agenda($\mathcal{A}$), as the set of activations applicable to $\mathcal{A}$.

```
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)}
```

The measure of the expansion $\mathcal{A}$, $\mu_C(\mathcal{A})$ (function `expansion_measure`), is constructed by recursion on the agenda of $\mathcal{A}$, adding the pair $(|y|_{\mathcal{A}}, |D|)$ (defined by `bag_assoc_activ`), for each applicable activation $[x\colon D, y]$.

```
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 3 We illustrate the evolution of agendas (and their associated measures) in Example 1.

$\mathcal{A}$	agenda($\mathcal{A}$)	measure($\mathcal{A}$)
$\mathcal{A}_0$	$\{[x_0\colon \forall R.D \sqcap (\exists R.(D \sqcup E) \sqcap \exists R.(D \sqcup F)), x_0]\}$	$\dot{\{}(15, 15)\dot{\}}$
$\mathcal{A}_1$	$\{[x_0\colon \exists R.(D \sqcup E), x_0], [x_0\colon \exists R.(D \sqcup F), x_0]\}$	$\dot{\{}(15, 5), (15, 5)\dot{\}}$
$\mathcal{A}_2$	$\{[x_0\colon \exists R.(D \sqcup F), x_0], [x_0\colon \forall R.D, x_1], [x_1\colon D \sqcup E, x_1]\}$	$\dot{\{}(14, 3), (14, 3), (15, 5)\dot{\}}$
$\mathcal{A}_3$	$\{[x_0\colon \exists R.(D \sqcup F), x_0]\}$	$\dot{\{}(15, 5)\dot{\}}$
$\mathcal{A}_4$	$\{[x_0\colon \forall R.D, x_2], [x_2\colon D \sqcup F, x_2]\}$	$\dot{\{}(14, 3), (14, 3)\dot{\}}$
$\mathcal{A}_5$	$\{\}$	$\dot{\{}\dot{\}}$

Finally, we prove the theorem that states the monotonicity of μ_C:

Theorem 1 *Let* $\mathcal{A}_1, \mathcal{A}_2 \in \mathcal{E}(C)$. *If* $\mathcal{A}_1 \to \mathcal{A}_2$ *then* $\mu_C(\mathcal{A}_2) <_{\text{mult}} \mu_C(\mathcal{A}_1)$

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

where `less_mult` defines $<_{\text{mult}}$, the extension to $\mathcal{M}(\mathbb{N} \times \mathbb{N})$ of the lexicographic order on $\mathbb{N} \times \mathbb{N}$.

The formalization of the proof of this theorem in PVS follows the hand proof in [21].[4] First, note that if we have $\mathcal{A}_1 \rightarrow \mathcal{A}_2$, then there exists an activation $Ac_1 = [x{:}\, D, y] \in \mathsf{agenda}(\mathcal{A}_1)$, corresponding to the applied rule. In addition, $Ac_1 \notin \mathsf{agenda}(\mathcal{A}_2)$ and, for each activation Ac_2 introduced in $\mathsf{agenda}(\mathcal{A}_2)$ as a result of the rule application, its associated pair is smaller (lexicographically) than $(|y|_{\mathcal{A}}, |D|)$. In fact, one of the following cases may occur:

1. $Ac_2 = [x{:}\, E, z]$, where z is successor of y (with respect to a relation) in $\mathcal{A}_2$. Then, $|z|_{\mathcal{A}_2} < |y|_{\mathcal{A}_2} = |y|_{\mathcal{A}_1}$. So, $(|z|_{\mathcal{A}_2}, |E|) < (|y|_{\mathcal{A}_1}, |D|)$.
2. $Ac_2 = [x{:}\, E, y]$, being E a subconcept of D. In this case, $|y|_{\mathcal{A}_2} = |y|_{\mathcal{A}_1}$ and $|E| < |D|$. So, $(|y|_{\mathcal{A}_2}, |E|) < (|y|_{\mathcal{A}_1}, |D|)$.

Second, as we saw in Example 2, the application of a rule may disable some activations of the agenda and, therefore it may eliminate its associated pairs from the multiset. Taking this into account and denoting as μ_{aux} the function `expansion_measure_aux` above, we define in PVS the multiset $K_1 = \mu_{\text{aux}}(\mathcal{A}_1, \mathsf{agenda}(\mathcal{A}_1) \setminus \mathsf{agenda}(\mathcal{A}_2))$, whose elements are the pairs associated to disabled activations. Also, the multiset $K_2 = \mu_{\text{aux}}(\mathcal{A}_2, \mathsf{agenda}(\mathcal{A}_2) \setminus \mathsf{agenda}(\mathcal{A}_1))$ contains the pairs associated to new enabled activations. Finally, $M_0 = \mu_{\text{aux}}(\mathcal{A}_1, \mathsf{agenda}(\mathcal{A}_1) \cap \mathsf{agenda}(\mathcal{A}_2))$ is the multiset whose elements are the pairs associated with the activations that remain enabled after the application of the rule. We proved the following properties:

1. $K_1 \neq \emptyset$
2. $\mu_C(\mathcal{A}_2) = M_0 \uplus K_2$
3. $\mu_C(\mathcal{A}_1) = M_0 \uplus K_1$
4. $(\forall a \in K_2)(\exists b \in K_1)[a < b]$

Thus, from the definition of the multiset extension, it is clear that $\mu_C(\mathcal{A}_2) <_{\text{mult}} \mu_C(\mathcal{A}_1)$.

Once the measure function has been constructed, the parameters T and $<$, and the signature `measure-concept(C)` of Section 3.4 are instantiated by the appropriate mechanism of PVS, substituting T by $\mathcal{M}(\mathbb{N} \times \mathbb{N})$, $<$ by $<_{\text{mult}}$ and `measure_concept(C)` by the `expansion_measure` function. Note that the function `expansion_measure` is also parametrized by the concept C but, in this case, this parameter is not visible since it is one of the parameters of the PVS theory.

4 Methodology of Refinements

Our purpose is to be able to define and prove properties of concrete tableau-based satisfiability checking algorithms, using the properties proved for the abstract formalization shown in the previous sections. The main idea is that we will *refine* it, specifying the data structures used and the search control, while preserving the essential properties already proved.

In this section we present a sketch of the type and operator refinement techniques we developed in PVS in order to relate different specifications of the same notion.

[4] Although it has some slight differences, because we had to fix some details.

The development is general and it could be applied in any context. In our case, we have applied it to prove the properties of concrete tableau-based satisfiability checking of $\mathcal{ALC}$ -concepts, as we will see in the next section.

Based on the idea of refinements of data types [12, 17] we have built in PVS a theory establishing the general notions of refinements and their 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 $f : R \to T$

```
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.

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

In a similar way as data types are refined, we can also refine 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 $op_{\text{ref}} : R_1 \to R_2$ is a *refinement* of op if the following diagram commutes:

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

```
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))
```

For instance, to build refinements corresponding to the operations over finite sets, we use operations over lists "simulating" the behaviour of analogous operations on finite sets. In particular, the built-in list operations `null?`, `cons` and `append` are refinements of `empty?`, `add` and `union`, respectively. As for the membership relation, it can be observed that if the refinement used for the inner types is injective, then the predicate `member` for lists is also a refinement of the predicate `member` for finite sets.

In general, for an operator op_{ref} to be a refinement, we require it to have the same behaviour as op. The most important feature of operator refinements is that they make it possible to transfer the properties of one operator op to its refined

version op_{ref}. For example, an important property in our formalization, that can be transferred through refinements, is the well-foundedness of a relation:

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

In general, 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

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

which is just the correctness theorem corresponding to op_{ref}. Therefore, once we have proved a correctness theorem for op, the correctness of op_{ref} is already proved. Note that it has been proved in a general way without any specific assumptions about ϕ, ρ and op. Thus, we can use this theory to transfer properties from op to op_{ref}.

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, to construct 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 $op_{1_{\text{ref}}}$ and $op_{2_{\text{ref}}}$ are refinements of op_1 and op_2, respectively, then $op_{2_{\text{ref}}} \circ op_{1_{\text{ref}}}$ is also a refinement of $op_2 \circ op_1$.

$$\begin{array}{ccccc} T_1 & \xrightarrow{op_1} & T_2 & \xrightarrow{op_2} & T_3 \\ {\scriptstyle f_1}\uparrow & & {\scriptstyle f_2}\uparrow & & {\scriptstyle f_3}\uparrow \\ R_1 & \xrightarrow{op_{1_{\text{ref}}}} & R_2 & \xrightarrow{op_{2_{\text{ref}}}} & R_3 \end{array}$$

5 Generic Tableau Algorithms for Checking Satisfiability of $\mathcal{ALC}$-Concepts

This section is devoted to presenting the construction of a generic tableau-based $\mathcal{ALC}$ satisfiability checking algorithm corresponding to the specification of the $\mathcal{ALC}$ abstract formalization that we have described in Section 3. Our purpose is to do it in such a way that its termination, soundness and completeness can be deduced from the corresponding properties of the abstract formalization. In order to do this, we will use the methodology of refinements explained in Section 4.

The algorithm presented in this section is generic in the sense that we will leave unspecified the strategy used to select the rule to apply and the order in which the two branches corresponding to the $\rightarrow_{\sqcup}$ rule are explored. The main idea is that by defining concrete strategies, we will obtain executable and formally verified algorithms for $\mathcal{ALC}$ satisfiability, as we will describe in the next section. In order to obtain executable code, the ground evaluator is able to extract Common Lisp code from functional PVS specifications, that can be evaluated on ground expressions. Moreover, we will use the PVSio package [20], that extends the PVS ground evaluator with a predefined library of imperative programming language features.

Note that the specification of the $\mathcal{ALC}$ abstract formalization cannot be directly transformed into an algorithm by composition of refined operators for each of the operators composing this specification, as sketched at the end of the previous section. The main reason is that, in the abstract formalization, the search process that the algorithm has to carry out in the space $\mathcal{E}(C)$ is not specified. To define an algorithm we have to concretize how to carry out the search. Then, we have to consider the following:

1. It is necessary to use data types whose elements can be evaluated by the PVS evaluator. To do this, we have to refine, among others, the type used to represent ABoxes (finite sets) by an executable type (in this case, lists).
2. It is necessary to define specifications of the predicates used for recognizing if an ABox is complete and clash-free, which have to be executable.
3. Since we will construct a recursive algorithm, it is necessary to have a well-founded relation over $\mathcal{E}(C)$, providing a measure function for proving its termination.
4. Finally, due to the non-determinism of the abstract formalization, it is necessary to introduce both a function that selects the completion rule applied in each step, and another function determining the order in which the branches produced by the union rule are explored. We will also need a function that applies completion rules in a functional way.

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

5.1 Refinements of Data Types

First, let us note that in the PVS specification of the abstract formalization 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 the 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. That is, PVSio does not waive PVS type checking.

This problem has been easily solved by 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
```

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]`

For our purposes, it is not necessary to refine the specification used to represent the notion of interpretation, since it is not used directly by the decision procedure. In the same way, it is not necessary for the notions of satisfiability to be executable; they only have to be equivalent to those defined in the abstract formalization. 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 Executable Refined Predicates

In order to define an executable refinement of the predicate `contains_clash` in the $\mathcal{ALC}$ abstract formalization (see Section 3.2), 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 executable PVS predicate `some`, that checks if some element of a list verifies a property:

```
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)
```

The predicate `complete` defined in the abstract formalization (Section 3.2) is quantified over all the ABoxes. This means that it cannot be executable, even when the `successor` relation was executable. Let us see how we have constructed a refinement of this predicate. First, we define predicates by recognizing when an assertional axiom is *expandable* in an ABox, according to each of the completion rules. For example, $x{:}\, C \sqcup D \in L$ is expandable in L if $x{:}\, C \notin L$ and $x{:}\, D \notin L$.

```
is_or_expandable(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 expandable axiom with respect to an ABox L, and the specification of the predicate `complete_l`:

```
is_expandable(Aa,L): bool =
 is_and_expandable(Aa,L) OR is_or_expandable(Aa,L) OR
 is_some_expandable(Aa,L) OR is_all_expandable(Aa,L)

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

The predicate `complete_l` it is not, strictly speaking, a refinement of the predicate `complete`; nevertheless it can be seen as a refinement in conjunction with the negation of `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))
```

Note that the role of the successor relation is the same, both in the specification of the abstract formalization 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 executable. Then, we could consider a definition of the successor relation in the same way as we have defined the notions of satisfiability. That is, it could be:

```
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 and then, we would prove that successor_l is a well-founded relation, by applying properties of refinements.

However, due to its necessary relationship with the predicate complete_l, we decided to refine each one of the relations representing a completion rule. For example, L_2 is a $\sqcup_1$-expansion of L_1 if there exists an assertion $x{:}\, C_1 \sqcup C_2$ expandable 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_expandable(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 $\mathcal{ALC}$ abstract formalization 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 because of the well-foundedness of a relation is preserved through refinements:

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

This theorem is straightforwardly proved in PVS, as shown by the following proof:

```
1  (skosimp)
2  (lemma "successor_wf")
3  (typepred "C!1")
4  (inst - "alc_f(C!1)")
5  (("1"
6    (lemma
7     "refinement_preserve_wf[expansion_abox_concept(alc_f(C!1)),
8                             expansion_abox_concept_l(C!1),
9                             c(alc_f_aax)]")
```

```
10    (("1"
11      (use "successor_l_is_refinement")
12     ....
```

5.4 Application of Completion Rules

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

```
or_step_1_ax(Aa,L): LABox =
 IF is_or_expandable(Aa,L)
 THEN cons (r_instanceof(left_i(Aa), conc1_or(right_i(Aa))), L)
 ELSE L ENDIF
```

Secondly, take 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 abstract formalization. Recall from Section 3.6 that 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 of the ABoxes obtained by application to L of the rules corresponding to an activation Ac[5]

```
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,

`apply_activ`$([x_0{:}\forall R.D, x_0], L) =$ (: :)

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

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

The final step to complete the specification of the generic algorithm is to introduce the strategies to decide which rule to apply in every step, and the order in which the two alternatives corresponding to the $\rightarrow_{\sqcup}$ rule will be explored. As we have previously mentioned, in this subsection these strategies will be introduced in a generic way, only assuming certain properties about them; in the next section, they will be instantiated by concrete strategies, obtaining executable algorithms.

[5]The PVS notation `(: ... :)` is an abbreviation for lists.

We have introduced those generic functions in the parameters of the theory just like also the initial concept C_0. We also import the theory of the $\mathcal{ALC}$ abstract formalization, in which the type of individuals NI has been instantiated by `nat` and x_0 has been instantiated by 0.

```
IMPORTING alc_completeness[NC,NR,nat,nat,0]
```

As for the strategy to select the completion rule to apply in each step (dealing with the don't-care nondeterminism), we declare the following function f, whose role is to select the rule to apply in each step:

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

The idea is that $f(L)$ selects an activation applicable to an ABox 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, $f(L)$ is the empty list; otherwise, it returns a singleton list with the selected activation.

In order to ensure the correctness of the algorithm, the function f has to verify some properties, which we introduce as PVS assumptions. First, let us observe that if L is not complete, then there are some rules applicable to L and therefore there are some applicable activations. In that case, we require f to select at least one of those activations. Also, we require that f only selects activations applicable to an expansion L of the initial concept C_0:

```
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)
```

As for the order to consider the two alternatives corresponding to the $\rightarrow_{\sqcup}$ rule (the don't-know nondeterminism), note that if the order suggested by the function `apply_activ` were chosen, the left branch will always be considered first. However, our intention is to specify a generic algorithm with the possibility to use different heuristics to decide which branch to explore first. Thus, we declare the function

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

verifying the following property:

```
g_ax_1: ASSUMPTION
  FORALL (Ac:r_activ,L:LABox):permutation?(g(Ac,L),apply_activ(Ac,L))
```

Finally, the generic algorithm we specify below is a tableau-based algorithm, that carries out a depth first search on the set of expansions of C, using the function f to select the rule to apply in every step and the function g to order the disjunction alternatives. It finishes when it finds a complete and clash-free ABox (from which a model of the initial input concept can be constructed), or when all its branches are closed (thus proving the unsatisfiability of the concept).

```
sat_alc_alg_g_aux_i(L: expansion_abox_concept_l(C_0)): RECURSIVE bool =
 IF complete_l(L) AND not_contains_clash_l(L)
 THEN TRUE
  ELSIF NOT not_contains_clash_l(L)
  THEN FALSE
  ELSE LET Ac = car(f(L)), S = g(Ac,L) IN
```

```
        IF null?(cdr (S))
        THEN sat_alc_alg_g_aux_i(car(S))
        ELSE LET L1 = car(S), L2 = car(cdr(S)) IN
              sat_alc_alg_g_aux_i(L1) OR sat_alc_alg_g_aux_i(L2)
        ENDIF
  ENDIF
 MEASURE L BY successor_l

sat_alc_alg_g: bool = sat_alc_alg_g_aux_i((: r_instanceof(0,C_0) :))
```

Note that the input of the algorithm is not explicit, because it is one of the parameters of the theory.

The termination of this algorithm is ensured by the well-foundedness of the `successor_l` relation. The soundness and completeness are easily proved in PVS by well-founded induction in the `successor_l` relation, using the same properties already proved for the specification of the $\mathcal{ALC}$ abstract formalization and the properties required to f and g.

```
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 Concrete Tableaux for Checking Satisfiability of $\mathcal{ALC}$ -Concepts

A particular tableau algorithm can be obtained by defining a rule selection function f and a heuristic function g, both verifying the assumptions above. We also have to instantiate the non-interpreted types used to represent the set of concept names, the set of role names and the set of individuals.

For instance, an usual application strategy of completion rules in functional algorithms for deciding satisfiability of $\mathcal{ALC}$ -concepts is the following (see [5]):

1. Whenever the $\rightarrow_{\sqcap}$ rule can be applied, apply the $\rightarrow_{\sqcap}$ rule;
2. Else, whenever the $\rightarrow_{\sqcup}$ rule can be applied, apply the $\rightarrow_{\sqcup}$ rule;
3. Otherwise, if a $\rightarrow_{\exists}$ rule can be applied, apply the $\rightarrow_{\exists}$ rule and all the $\rightarrow_{\forall}$ rules derived from it.

This is defined in PVS by the following function `fi`:

```
fi(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 unitary list with one of the activations corresponding to the $\rightarrow_*$-rule applicable to L, if there are such activations; or otherwise the empty list.

As for the strategy for exploring the alternatives corresponding to a disjunction $C_1 \sqcup C_2$, for example we can define the following:

1. If $\neg C_1$ is in the Abox, first try the branch corresponding to C_2, and vice-versa.
2. Otherwise, first explore the branch corresponding to the disjunct of lesser size.

The following PVS function `gi` defines this strategy:

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

Thus, the concrete decision procedure is

```
sat_alc_alg_i(C): bool = sat_alc_alg_g[string,string,C,fi,gi]
```

Note that the parameters of the PVS theory in which the generic tableau is specified are the type of concept names, the type of role names, the initial concept, the selection function and the ordering function. In this case, we instantiate with the parameters `string`, `string`, `fi`, `gi` and the concept C. It is worth noting the different positions of C in the previous definition.

Finally, to obtain the correctness of this executable algorithm it suffices to prove that `fi` and `gi` verify the assumptions required in Section 5.4 for `f` and g, respectively. These proof obligations are automatically generated by the PVS system when the functions are instantiated and have to be proved by the user. Once proved, the theorems stating the correctness of this reasoner are proved:

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

In the source code, we include this concrete tableau algorithm, as well as additional instantiations of the generic algorithm with different versions for f and g, with some statistics of the performance obtained by executing these different versions.

7 Conclusions and Future Work

We have presented an abstract formalization for checking satisfiability of $\mathcal{ALC}$-concepts in PVS, which is based on a set of transformation rules, proving its termination, soundness and completeness. From this, we have constructed a generic tableau-based algorithm using the methodology of refinements to transfer its main properties. Finally, we have obtained some concrete reasoners by instantiation of non-interpreted types and of the functions coding the strategies for rule applications.

It should be pointed out that the choice of PVS as our verification system has turned out to be beneficial for our formalization, since PVS allows for definitions of abstract datatypes, inductive sets and dependent types as well as parametrized theories of sets, multisets and graphs.

In Fig. 2 we present a graphical representation of the theories used and developed in our formalization of the $\mathcal{ALC}$ logic, along with their dependencies. It is worth pointing out that we reused some existing libraries of PVS (ovals in the figure). In particular, we used the PVS library of graphs for the definition of the tree associated with the completion process, the PVS library of structures for the formalization of the well-foundedness of multiset relations and PVSio for the evaluation of examples. As for our formalization, it is structured in the following libraries:

- `wf`, a characterization of the well-foundedness of an ordering, based on [1]
- `bags-wf`, for the well-foundedness of multiset relations
- `refinement`, the library with all the general theory of refinements
- `lists-aux` and `sets-aux` extend the theory of lists and sets of PVS with additional properties
- Finally, the `ALC` library contains the main body of the formalization.

To give and idea of the proof effort, we give in Fig. 2 the numbers of theorems proved in each of the libraries we have developed.

It is worth pointing out that the way in which we have approached the task makes the formal verification of the properties of the reasoners easier: once the correctness properties of the generic algorithm are proved, everything is reduced to proving the hypotheses assumed in the generic functions that code the rule application strategies. In general, we think that this example shows that this approach, where the logic of the algorithm is clearly separated from concrete control strategies, could benefit other formalizations where the final goal is the verification of algorithms.

As a byproduct, we have developed a number of PVS formalized theories that could be used in other formalization work. For example, the multiset ordering, used

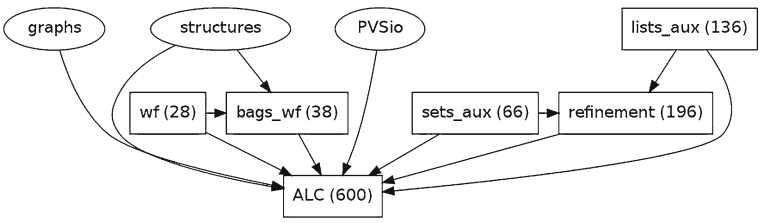

Fig. 2 Libraries used and developed

for proving termination of our tableau algorithm, could be used in other non-trivial termination proofs.

As for related work, formalizations of logical systems and related algorithms have been carried out in many of the main computer aided reasoning system, most of them dealing with propositional and first-order logic. For example [7, pp. 56–86] and [27] in Nqthm, [9] in Nuprl, [14] in HOL, [19] in ACL2, [26] in Isabelle or [10] in Mizar. More recently special attention has been devoted to the formal verification of efficient SAT algorithms for propositional logic: [18] in Isabelle, [4] in Coq and [28] in PVS. To the best of our knowledge, there are no previously reported formalizations of description logics.

We think that the work presented here is a good starting point for further formalizations related to the verification of reasoning algorithms for description logics. We plan to continue this work by following several lines. First, extending the reasoners for the $\mathcal{ALC}$ logic to other description logics, incrementally bringing us closer to the description logic $\mathcal{SHOIN}$, which is the description logic corresponding to OWL-DL. Second, the book [25] introduces Sequent Calculi and Natural Deduction for some Description Logics ($\mathcal{ALC}$, $\mathcal{ALCQ}$); since both are similar to a tableau calculus, we think that the formalization of these logical calculi could be easily carried out from our $\mathcal{ALC}$ abstract framework. Third, note that efficiency has not been our main concern in this work; so we plan to define more efficient algorithms than the one presented here. For example, semantic branching, boolean constraint propagation and more efficient heuristics for the application of rules [16] could also be defined in a generic way as well, and also as a refinement of the abstract framework following the same methodology. Finally, regarding proof development, it would be desirable to design specific strategies for the most common proofs carried out in the formalization, thus reducing user interaction.

References

1. Aczel, P.: An introduction to inductive definitions. In: Barwise, J. (ed.) Handbook of Mathematical Logic, pp. 739–782. North–Holland Publishing Company (1977)
2. Alonso, J.A., Borrego, J., Hidalgo, M.J., Martín, F.J., Ruiz, J.L.: Verification of the formal concept analysis. RACSAM (Revista de la Real Academia de Ciencias), Serie A: Matemáticas **98**, 3–16 (2004)
3. Alonso, J.A., Borrego, J., Hidalgo, M.J., Martín, F.J., Ruiz, J.L.: A formally verified prover for the ALC description logic. In: Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 4732, pp. 135–150. Springer (2007)
4. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Wener, B.: Verifying SAT and SMT in Coq for a fully automated decision. In: PSATTT'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories (2011)
5. Baader, F., Hollunder, B.: A terminological knowledge representation system with complete inference algorithms. In: Proceedings of the First International Workshop on Processing Declarative Knowledge, pp. 67–86 (1991)
6. Baader, F., McGuiness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
7. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press (1979)
8. Butler, R.W., Sjogren, J.: A PVS graph theory library. Technical report, NASA Langley (1998)
9. Caldwell, J.L.: Classical propositional decidability via Nuprl proof extraction. In: Theorem Proving in Higer Order Logics. Lecture Notes in Computer Science, vol. 1479, pp. 105–122. Springer (1998)
10. Caminati, M.B.: Basic first–order model theory in Mizar. J. Formalized Reasoning **3**(1), 49–77 (2010)

11. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Commun. ACM **22**(8), 465–476 (1979)
12. Dold, A.: Formal software development using generic development steps. PhD thesis, Ulm University (2000)
13. Haarslev, V., Möller, R.: RACER system description. In: International Joint Conference on Automated Reasoning, IJCAR'2001. Lecture Notes in Computer Science, vol. 2083, pp. 701–705. Springer (2001)
14. Harrison, J.: Formalizing basic first order model theory. In: Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98. Lecture Notes in Computer Science, vol. 1497. Springer (1998)
15. Hidalgo, M.J., Alonso, J.A., Martín, F.J., Ruiz, J.L.: Constructing formally verified reasoners for the description logic. In: Proceedings of the 3rd international workshop on automated specification and verification of web systems (WWV 2007). Electronic Notes in Theoretical Computer Science, vol. 200, pp. 87–102 (2008)
16. Horrocks, I.: Implementation and optimization techniques. In: Baader, F., McGuiness, D., Nardi, D., Patel-Schneider, P. (eds.) The Description Logic Handbook: Theory, Implementation and Applications, pp. 306–346. Cambridge University Press (2003)
17. Jones, C.B.: Systematic Software Development using VDM. Prentice–Hall International (1990)
18. Marić, F.: Formalization and implementation of modern SAT solvers. J. Autom. Reasoning **43**, 81–119 (2009)
19. Martín, F.J., Alonso, J.A., Hidalgo, M.J., Ruiz, J.L.: Formal verification of a generic framework to synthesize SAT–provers. J. Autom. Reasoning **32**(4), 287–313 (2004)
20. Muñoz, C.: PVSio reference manual version 2.b. Technical report, National Institute of Aerospace (2005)
21. Nutt, W.: *Algorithms for Constraint in Deduction and Knowledge Representation*. PhD thesis, Universität des Saarlandes, 1993.
22. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer (1992)
23. Owre, S., Shankar, N.: Abstract datatype in PVS. Technical report, Computer Science Laboratory, SRI International (1997)
24. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS language reference. Technical report, Computer Science Laboratory, SRI International (1999)
25. Rademaker, A.: A Proof Theory for Description Logics. Springer (2012)
26. Ridge, T., Margetson, J.: A mechanically verified, sound and complete theorem prover for first order logic. In: Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 3603, pp. 249–309. Springer (2005)
27. Shankar, N.: Towards mechanical metamathematics. J. Autom. Reasoning **1**, 407–434 (1985)
28. Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-Based satisfiability solver. Electronic Notes in Theoretical Computer Science **269**, 3–17 (2011)
29. Sirin, E., Parsia, B.: Pellet system description. In: Proceedings of the 2006 International Workshop on Description Logics (DL2004). CEUR Workshop Proceedings, vol. 189 (2006)
30. Tobies, S.: Complexity results and practical algorithms for logics in knowledge representation. PhD thesis, RWTH–Aachen University (2001)
31. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: system description. In: Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006). Lecture Notes in Artificial Intelligence, vol. 4130, pp. 292–297. Springer (2006)