Reifying Concepts in Description Logics

Liviu Badea
AI Research Lab
Research Institute for Informatics
8-10 Averescu Blvd., Bucharest, Romania
e-mail: badea@roearn.ici.ro

Abstract

Practical applications of description logics (DLs) in knowledge-based systems have forced us to introduce the following features which are absent from existing DLs:

- allowing a concept to be regarded at the same time as an individual (the instance of some other meta-level concept)
- allowing an individual to represent a collection (set) of other individuals.

The first extension, called *concept reification*, is more general and thus can cover the second one too. We argue that the absence of these features from existing DLs is an important reason for the lack of a unified approach to description logics and object-oriented databases.

We also show that concept reification cannot be dealt with by the standard DL semantics and propose a slightly modified semantics that takes care of the inherent higher-order features of reification in a first-order setting. A sound and complete inference algorithm for checking consistency in reified $\mathcal{ALCO}_{\in}$ knowledge bases is subsequently put forward.

1 Introduction

Description logics (DLs) are descendants of the famous KL-ONE system [Brachman and Schmolze, 1985] and can be viewed as formalizations of the frame-based knowledge representation languages.

Systems based on DLs are hybrid systems which separate the described knowledge in two distinct categories: *terminological* and *assertional* knowledge. The terminological knowledge is generic and refers to classes of objects and their relationships, while the assertional knowledge describes particular instances (individuals) of these classes. These two levels are completely disjoint since a given object cannot be at the same time a concept *and* an instance. (Description logics further distinguish between two kinds of terminological knowledge, namely concepts and roles. Concepts are essentially unary predicates interpreted as sets of individuals, while roles represent binary predicates interpreted as (binary) relations between individuals.)

An important limitation of current description logics is the clear-cut separation between the terminological (intensional) and the assertional (extensional) level. For example, concepts (representing intensional descriptions of sets of individuals) and their instances are stored at different levels and cannot be mixed under any circumstances.

In certain applications, however, it may be useful to be able to regard a given concept (class) as the instance of a higher level meta-concept (meta-class). This would allow us to *reuse* terminologies by constructing a unique generic terminology which could then be instantiated to produce several particular terminologies.

This paper presents an extension of description logics in which a given concept can be regarded as an individual (i.e. an instance of some other meta-level concept). This process, called *concept reification*, has not been extensively studied in the framework of description logics[1], mainly since it mixes the terminological and assertional levels and therefore spoils the simplicity of the currently used reasoning techniques. Also, reification introduces a form of *higher-order constructs* in description logics thereby complicating the issue of defining a proper semantics of the logic as well as the associated inference services.

In spite of these difficulties, reification is absolutely necessary whenever we want to achieve *reusability* in a knowledge-based system. The following example, taken from [Badea and Ţilivea, 1996], deals with allocating the staff of some research institution. In such a setting we may want to introduce concepts like *manager*, *secretary*, *researcher* and instances like *Tom*, *Joan*, *Mary*, *Peter*, *Fred*, etc: $Tom \in manager$, $Joan \in secretary$, $Mary \in secretary$, $Peter \in researcher$, $Fred \in researcher$.

But now note that the concepts *manager*, *secretary* and *researcher* represent positions in the research institute. They are therefore not only concepts, but also *instances* of the (metal-level) concept

[1] The CLASSIC system [Brachman *et al.*, 1991] already included meta-individuals (a pre-theoretic form of concept reification), but these are not taken into account in DL inferences.

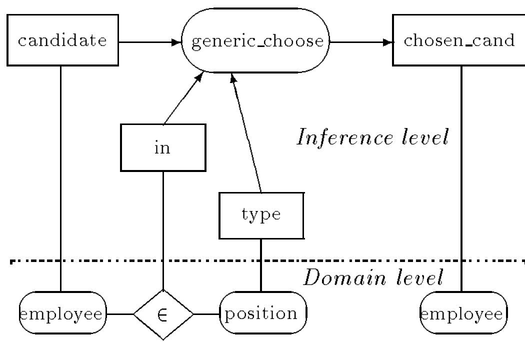

Figure 1: A generic inference using the relation ∈.

position: *manager* ∈ *position*, *secretary* ∈ *position*, *researcher* ∈ *position*.

The concept *position* should not be confused with the concept *employee*, which is a super-concept of *manager*, *secretary* and *researcher*: *manager* ⊂ *employee*, *secretary* ⊂ *employee*, *researcher* ⊂ *employee*.

Now consider an employee allocation problem. In a typical knowledge-based system, we would have to write separate inferences for instance for retrieving managers, secretaries and researchers. This would amount to writing three separate pieces of code that are extremely similar (though not identical, as at least the positions of the employees to be retrieved would be different).

If we would like to avoid writing three separate pieces of code (inferences), we would have to write a *generic* inference that would be parametrized by the *position* of the employee we would like to choose. This can be accomplished by using an input parameter, called *type*, linked to the concept *position* and which is supposed to specify the position of the person to be chosen. Figure 1 is a graphical representation of such a generic inference in the KADS-based $E_x\mathcal{C}laim$ system (see [Badea, 1996] for more details on $E_x\mathcal{C}laim$). Note that we are making use of the built-in membership role ∈, which links an instance X with a concept C whenever X is an instance of C.

The predefined role ∈ links a concept (*employee*) with a meta-concept (*position*) and allows therefore a kind of generic inferences which are essential for developing *domain-independent and reusable models*.[2]

Note that domain-independence and reusability of models could not have been achieved without concept reification and the role ∈.

In order to be usable in real-life applications, description logics will also have to allow for an individual (instance) to represent a *collection* (set) of other individuals. However, existing (implemented) DL systems usually lack constructors for sets or lists of objects[3] and we therefore have to represent such collections outside the DL thereby affecting the completeness of the DL reasoning services.

Using concept reification, we can obtain individuals representing sets of other individuals by reifying concepts of the form $one_of(i_1, \ldots, i_n)$. This observation allows us to concentrate in the following on "concept reification".

Note that the role ∈ allows us to regard the assertional component of the DL (the ABox) as consisting of role tuples only, since instance assertions of the form $X : C$ can be viewed as tuples of the role ∈.

Concept reification, as studied in this paper, is different from Kobsa's role reification implemented in SB-ONE [Kobsa, 1991]. More precisely, while the reification of a concept is an individual, Kobsa's reification of a role is a concept. (Kobsa's approach has been motivated by natural language applications in which a verb, for example, is regarded in some contexts as a role and in other contexts as a concept.) Therefore, while we are concerned with mixing the TBox and the ABox of a DL, Kobsa has dealt with mixing concepts and roles within the TBox (while keeping TBox and ABox disjoint).

As previously mentioned, concept reification involves a form of higher-order logic since the interpretations of value and existential restrictions on the ∈-role employ a form of quantification over *concept-valued* variables. Therefore, whereas in ordinary DLs concepts exist only as named, terminological (*TBox*) level elements, in reified DLs concepts may be individuals ("data") as well.

Reification and the related higher-order features are also essential in object oriented databases (OODBs) [Beeri, 1990]. In classical database systems there are two distinct levels: *data* and *schema* (similar to ABox and respectively TBox in description logics).

In OODBs, meta-data (such as classes and functions), are frequently treated as data. Class objects acquire thereby a dual nature: on the one hand they are data and can be manipulated by the system; on the other hand, they are schema-level objects and thus part of the schema.

This situation is similar to concept reification as introduced in this paper. In fact, it is our opinion that the lack of a unified approach to description logics and OODBs is mainly due to the clear-cut separation of TBox and ABox (i.e. to the absence of reification) in DLs.

However, introducing concept reification in DLs is significantly harder than in OODBs since we have to modify the DL inference services (consistency and subsumption tests) to cope with the new construct. Since no analog inference services exist in OODBs and as long as we

[2]Not only is it cumbersome to have three identical pieces of code, but these pieces of code would depend on the domain level (the types of positions – *director*, *secretary* and *researcher* are domain-dependent; we cannot change the domain level, for example by introducing a new position, without having to modify the inference level too, since we would have to add a new inference for the new position type).

[3]Some description logics provide the $one_of(i_1, \ldots, i_n)$ construct which denotes the concept whose extension is given by the set of instances $\{i_1, \ldots, i_n\}$. However, what we need is a concept construct whose *instances* denote sets or lists of other instances.

deal with reification in an explicit manner alone, there seem to be no complications in reasoning with the new construct in OODBs.

Another somewhat related formalism is *F-logic* [Kifer *et al.*, 1995], which attempts to provide sound logical foundations of object-oriented as well as frame-based languages and can be considered as a declarative approach to deductive object-oriented databases.

F-logic provides a form of explicit reification, but it lacks the $\in$-role and the related DL inference services. Therefore, we can easily represent F-logic object models in reified DLs. For instance, an F-logic *non-inheritable property* $object[property \rightarrow value]$ would be represented in reified DLs as a tuple of the role *property* involving the individual *object*: $(object, value) : property$, while an *inheritable* property of the form $object[property \bullet\!\!\rightarrow value]$ would be captured by a DL terminological axiom imposing a restriction on the fillers of the role *property* for the *instances* of *object*: $object \subset \exists property.\{value\}$. F-logic signature (typing) expressions of the form $object[property \Rightarrow type]$ can also be represented in DLs as value restrictions $object \subset \forall property.type$. Of course, set-valued attributes in F-logic correspond to DL roles, while normal F-logic attributes would be represented in DLs as functional roles (attributes). It is our opinion that the DL representation makes the distinction between the assertional and terminological properties of objects clearer.

It is worth noting that many frame-based knowledge representation systems (for example KEE) allow a class to be at the same time an instance, but this feature has usually no associated formal semantics.

2 Reifying concepts in description logics - semantical considerations

Traditional description logics separate the terminological (TBox) and assertional (ABox) levels completely by not allowing a concept to be regarded at the same time as an individual. This simplifies the semantics and corresponding reasoning algorithms.

In this paper we consider an extension to DLs that eliminates this restriction. Concept reification amounts to associating with each concept C an individual C'.

Additionally, we allow the membership role $\in$ and its inverse $\ni$. The role $\in$ links an individual X with some other reified concept C' whenever X is an instance of C (regarded as a concept).

As already mentioned, concept reification involves a form of higher-order logic. For example, in defining the interpretation of $\forall \ni .C$:

$$(\forall \ni .C)^{\mathcal{I}} = \{x \in \Delta \mid \forall y \,.\, x \ni^{\mathcal{I}} y \rightarrow y \in C^{\mathcal{I}}\}$$

we quantify over *all concepts y*, not just the ones that are explicitly given. Since higher-order logics lack even a sound a complete axiomatization and in order to preserve the desired computational properties, we will restrict the semantics of the logic to a first-order semantics. This amounts to interpreting quantified concept variables as ranging over (explicitly given) reified individuals, or *intensions*, rather than over all concepts that potentially exist (or their *extensions*). Therefore, we will allow reification of *concept names* only.

Actually, we can drop the explicit reification construct $\cdot'$ (and use C instead of C' for all concept names C) because we can determine the type of an object (whether it is a concept or an individual) from the context in which it is used. For example, C in the concept term $\forall R.C$ is regarded as a concept, while if we use it in the assertional axiom $C : D$, then C represents the reification of the corresponding concept (i.e. an individual). As previously mentioned, since we don't have an explicit reification operator, we will allow reification of concept names only. For instance, we will not allow ABox assertions like $(\forall R.C) : D$.

Also, since we interpret concept-valued variables over intensions rather than extensions, it may be that the reified counterparts of two equivalent concepts, C and D, represent two different individuals.

We can use the membership role $\in$ to express all instance assertions $X : C$ (for concept *names* C) as tuple assertions of $X \in C$. Therefore, we can regard the ABox as consisting of tuple assertions only.

Reified description logics are quite expressive.

For example, it is easy to check that $\forall \ni .C$ represents, roughly speaking, the power-set $\mathcal{P}(C)$ of C (the set of subset of C), while $\exists \in .C$ denotes the union $\bigcup C$ of the instances of C, regarded as sets.

Although reified DLs are capable of representing concepts like $\forall \ni .\top$ denoting "the set of all sets", we do not run into logical paradoxes, since $\forall \ni .\top$ is equivalent, as expected, to the top concept $\top$.

Since the union of all subsets of a set C is equal to C, $\bigcup \mathcal{P}(C) = C$, we obtain the identity $\exists \in .\forall \ni .C = C$ which could be regarded as an axiom in reified DLs. However, only the "$\leftarrow$" direction is specific to $\in$, the "$\rightarrow$" direction being an instance of the axiom for role inverses.

Additionally, note that C is an instance of $\exists \ni .D$ iff $C \sqcap D$ is consistent, and that C is an instance of $\forall \ni .D$ iff $C \subset D$. In reified DLs, subsumption is therefore reducible (within the language) to instance checking.

The following identities specific to reified DLs can also be easily checked[4]:

$$\exists \in .\top = \top \qquad (\textit{or its dual}:\ \forall \in .\bot = \bot)$$
$$\forall \ni .\bot = \{\bot\} \qquad \exists \in .\{C\} = C.$$

Also, since C is an instance of $\exists \ni .\{X\}$ iff $X \in C$, we can regard $\exists \ni .\{X\}$ as denoting the set of concepts C for which X is an instance. This observation shows that the "realization problem"[5] for X has a solution expressible in the language, namely $\exists \ni .\{X\}$.

[4] $\{X\}$ is the singleton concept whose extension has only one element, X.

[5] retrieving the set of concepts C that admit a given individual X as an instance.

Observe that X is an instance of $\forall \in .C$ iff $\exists \ni .\{X\} \subset C$.

The examples and observations above give a flavour of the intricate ways in which reification and the $\in$-role interact with one another and the other DL constructors.

After having informally presented concept reification in DLs, let us now try to formalize it. For reasons of simplicity, we are going to discuss reification in the $\mathcal{ALCO}_\in$ language ($\mathcal{ALC}$ of Schmidt-Schauß and Smolka [1991] extended with the *one-of* construct), but the results are easily extensible to more expressive languages.

The *syntax* of reified $\mathcal{ALCO}_\in$ deals with the following three sets of (syntactic) objects:

- *Names* denoted by $X, Y, \dots$ (including $\bot$)
- *Concept-Terms* denoted by $C, D, \dots$
- *Role-Terms* denoted by $R, Q, \dots$

The set *Names* contains individuals and concept names occurring in the DL knowledge base. Since we want to allow for the reification of concept names, individuals and concept names will have to belong to a single syntactic category (*Names*) (as opposed to traditional DLs where they fall under syntactically disjoint categories).

Concept-Terms are terms built from concept names (belonging to *Names*) using the following $\mathcal{ALCO}_\in$ concept constructors:

$$C ::= X \mid C \sqcap D \mid C \sqcup D \mid \neg C \mid \forall R.C \mid \exists R.C \mid \{X\}.$$

($\{X\}$ is a singleton concept; general $one_of(X_1, \dots, X_n)$ concepts can be represented as $\{X_1\} \sqcup \dots \sqcup \{X_n\}$.)

Role-Terms are built from role names (which are *not* in *Names*, since we do not reify roles) using the specific DL role constructors. Since $\mathcal{ALCO}$ admits only role names, $\mathcal{ALCO}_\in$ will admit only the following roles:

$$R ::= RN \mid \in \mid \ni .$$

The *semantics* of a DL allowing reification is also a bit different from the usual DL semantics.

Traditional DLs separate the terminological (TBox) and assertional (ABox) levels completely and define a polymorphic interpretation function $\cdot^{\mathcal{I}}$ which interprets concepts as sets of elements of some interpretation domain Δ and individuals as elements of Δ:

$$\mathcal{I} : Concepts \to 2^{\Delta} \qquad \mathcal{I} : Individuals \to \Delta.$$

As long as individuals and concepts are distinct, this works fine. But as soon as we allow concept reification, a given object name X can play both the role of an individual *and* of a concept name and we cannot use the above polymorphic interpretation function any more (because we wouldn't know how to define $X^{\mathcal{I}}$: as an element of Δ, or as a subset of Δ ?).

Therefore, we are forced to introduce two different interpretation functions:

(1) one that interprets object names as elements of the interpretation domain Δ (i.e. regards them as individuals) $\nu : Names \to \Delta$ (the "name function") and

(2) one that associates an *extension* with concept and role terms (the "extension function"):

$$\varepsilon : Concept_Terms \to 2^{\Delta} \qquad \varepsilon : Role_Terms \to 2^{\Delta \times \Delta}.$$

The "name function" maps object names to elements of the interpretation domain Δ. Such elements $x \in \Delta$ can be regarded either as individuals (if we use their names) or as concept names (if we use their extensions).

In order to retrieve the extension of an element $x \in \Delta$ regarded as a concept name, we need an additional function, the *"value function"* $\mathcal{V} : \Delta \to 2^{\Delta}$.

$\mathcal{V}$ associates with each $x \in \Delta$ the extension $\mathcal{V}(x) \subset \Delta$ of the concept name denoted by x, and is therefore uniquely determined by the extension of the $\in$-role:

$$\mathcal{V}(x) = \{y \in \Delta \mid (y, x) \in \varepsilon(\in)\} \qquad (1)$$

(in fact, $\mathcal{V}$ is a functional representation of the extension of the role $\ni$). $\varepsilon(\in)$ must verify the following (first-order) constraint:

$$\forall x \in \Delta . \exists y \in \Delta . [(x, y) \in \varepsilon(\in) \wedge \forall z . ((z, y) \in \varepsilon(\in) \to z = x)]$$

(i.e. $\forall x \in \Delta . \exists y \in \Delta . \mathcal{V}(y) = \{x\}$). This constraint says that each singleton $\{x\}$ must have an *intension* y in Δ (this is needed, for example, to prove $C \to \exists \in . \forall \ni .C$). We also interpret the $\ni$-role as the inverse of the $\in$-role: $\varepsilon(\ni) = \varepsilon(\in)^{-1}$.

Using the "value function" and the "name function", we can now construct the extensions of concept names as

$$\varepsilon(X) = \mathcal{V}(X^{\nu}) \qquad \varepsilon(\bot) = \emptyset. \qquad (2)$$

The extension of concept terms is defined as usual depending on the particular DL concept constructors:

$$\begin{aligned}
\varepsilon(C \sqcap D) &= \varepsilon(C) \cap \varepsilon(D) \\
\varepsilon(C \sqcup D) &= \varepsilon(C) \cup \varepsilon(D) \\
\varepsilon(\neg C) &= \Delta \setminus \varepsilon(C) \qquad (3) \\
\varepsilon(\forall R.C) &= \{x \in \Delta \mid \forall y \in \Delta . (x, y) \in \varepsilon(R) \to y \in \varepsilon(C)\} \\
\varepsilon(\exists R.C) &= \{x \in \Delta \mid \exists y \in \Delta . (x, y) \in \varepsilon(R) \wedge y \in \varepsilon(C)\} \\
\varepsilon(\{X\}) &= \{X^{\nu}\}.
\end{aligned}$$

An interpretation $\mathcal{I}$ is uniquely determined by the "name function" $\nu : Names \to \Delta$ and the "extension function" restricted to role names (including $\in$) $\varepsilon : Role_Names \cup \{\in\} \to 2^{\Delta \times \Delta}$. It can be extended to a full interpretation as follows:

- first, we extend ε from role names to role terms
- then we use $\varepsilon(\in)$ to determine the "value function" $\mathcal{V} : \Delta \to 2^{\Delta}$ according to (1)
- this in turn helps us define the extension of concept names according to (2)
- finally, ε extends naturally to concept terms as in (3).

Having two different interpretation functions ν and ε applicable to a given object X allows us to talk about the interpretation X^{ν} of X as an individual *and* as a concept $\varepsilon(X)$ *at the same time*! This wasn't possible in the old setting, where we had a single polymorphic interpretation function $\cdot^{\mathcal{I}}$.

3 Reasoning in reified $\mathcal{ALCO}_\in$

Like in traditional description logics, the reasoning services in reified $\mathcal{ALCO}_\in$ are reducible to the knowledge base (KB) consistency test [Buchheit *et al.*, 1993]. Therefore, we will concentrate in the following on checking consistency in $\mathcal{ALCO}_\in$ knowledge bases. The algorithm is a non-trivial extension of the algorithm for $\mathcal{ALC}$ and is based on a tableaux-like calculus operating on constraint systems.

Starting from an initial constraint system representing the KB, the calculus tries to construct a model of the knowledge base by applying a series of propagation rules. In doing so, it may discover obvious contradictions (clashes) and report the inconsistency of the original KB, or it may come up with a complete clash-free model, thus proving the satisfiability of the knowledge base.

The initial knowledge base to be tested for consistency is represented as a set of constraints of the form:

$$X : C, \quad (X, Y) : R, \quad def(CN, C)$$

where X, Y and CN are names, C is a concept term and R a role term. We also assume that all the concepts and roles occurring in constraints have been previously brought to the *negation normal form*.

The KB consistency checking algorithm applies a series of propagation rules to a given constraint set S, until either an obvious contradiction (or clash) is generated (thereby proving the consistency of S), or no propagation rules are applicable any more (case in which the constraint system is called *complete* and can be used to construct an interpretation of S).

The propagation rules for $\mathcal{ALCO}_\in$, presented in Figure 2, can take the following two forms:

$$\alpha \to \beta \ \ if\ \gamma \qquad\qquad \alpha \Rightarrow \beta \ \ if\ \gamma.$$

Both forms fire only if the condition γ holds and if the current constraint system contains constraints matching α. After execution, the first deletes the constraints matching α from the constraint system, while the second keeps them. Both forms add the constraints from β to the constraint system after firing.

The predicate $individual(X)$ succeeds on constant individuals, variables or singleton constructs $\{X\}$, while $role_name(R)$ succeeds only on role names (excluding $\in$ and $\ni$).

The rule $(\neg\{\bot\})$ can be explained as follows: $C : \neg\{\bot\}$ holds iff $C \neq \bot$, i.e. C is not the "empty" concept. C is therefore consistent and admits an instance $X : C$. The special case $\bot : \neg\{\bot\}$ is avoided by this rule since it is dealt with by the $(clash_{\{\}})$ rule.

Note that the $(\forall_\in)$ rule asserts Y to be an instance (of C) only if it is an individual (since we allow reification of concept names only).

Since X is an instance of the singleton concept $\{X\}$, rules $(\forall_{\in\{\}})$ and $(\forall_{\ni\{\}})$ make sure that this is taken into consideration during constraint propagation.

$(clash_\neg)$	$X : \neg C, X : C \to fail$
$(clash_{\{\}})$	$X : \neg\{X\} \to fail$
$(\bot)$	$X : \bot \to fail$
$(\sqcap)$	$X : C \sqcap D \to X : C, X : D$
$(\sqcup)$	$X : C \sqcup D \to X : C \mid X : D$
$(\exists)$	$X : \exists R.C \Rightarrow (X,Y) : R, Y : C$ $(new\ variable\ Y)$
$(\forall)$	$X : \forall R.C, (X,Y) : R \Rightarrow Y : C$ $if\ role_name(R)$
$(\forall_\in)$	$X : \forall \in .C, X : Y \Rightarrow Y : C$ $if\ individual(Y)$
$(\forall_{\in\{\}})$	$X : \forall \in .C \Rightarrow \{X\} : C$
$(\forall_\ni)$	$X : \forall \ni .C, Y : X \Rightarrow Y : C$
$(\forall_{\ni\{\}})$	$\{X\} : \forall \ni .C \Rightarrow X : C$
$(\{\})$	$X : \{Y\} \to X = Y$
$(\neg\{\bot\})$	$C : \neg\{\bot\} \to X : C$ $if\ C \neq \bot$ $(new\ variable\ X)$
$(\in)$	$(X, C) : \in \to X : C$
$(\ni)$	$(C, X) : \ni \to X : C$
(def)	$def(CN, C), X : CN \Rightarrow X : C$

Figure 2: The propagation rules for reified $\mathcal{ALCO}_\in$

Finally, rule (def) deals with acyclic[6] definitions $def(CN, C)$ of concept names CN (C being a concept term). Such definitions are interpreted semantically as $\varepsilon(CN) \subset \varepsilon(C)$. We shall not address the issue of more complex definitions (like general concept inclusions or equations) in this setting, since $\mathcal{ALCO}_\in$ is *"unstable"* due to the presence of the $\ni$ role (the inverse of $\in$). Instability amounts to the possibility that after having expanded all the constraints for some individual X, at some point in the future new constraints involving X get discovered. For example, if we apply the propagation rules to $x : (c \sqcap \forall \in .\forall \ni .d)$ and expand all the constraints involving x, we obtain the constraints (1)-(3) below. But subsequent applications of propagation rules eventually discover a new constraint involving x (namely (5)), thus proving the instability of $\mathcal{ALCO}_\in$.

(1)	$x : (c \sqcap \forall \in .\forall \ni .d)$	
(2)	$x : c$	$(\sqcap) : (1)$
(3)	$x : \forall \in .\forall \ni .d$	$(\sqcap) : (1)$
(4)	$c : \forall \ni .d$	$(\forall_\in) : (2), (3)$
(5)	$x : d$	$(\forall_\ni) : (2), (4)$

The same kind of problem occurs if we allow role inverses and general inclusions, for which the results in [Buchheit *et al.*, 1993] are no longer applicable (because the stability lemma fails in the presence of role inverses). The problems posed by role inverses are deep and will not be tackled in the present paper since they are orthogonal to the issue of interest here (namely reification).

Note that the ABox assertion $\top : \forall \ni .C$ is equivalent

[6] Cycles "through" the $\in$ (or $\ni$) role are anyhow problematic from a semantical point of view. For example, since $\ni$ should be well-founded ($\neg repeat(\ni)$ in *repeat-PDL* notation), cycles involving $\ni$ should be interpreted probably w.r.t. least fixpoint semantics.

Also, such cycles don't seem to occur in practical applications anyway.

to the TBox axiom stating the validity of C and can therefore be used to express general concept inclusions or equations. In order to avoid the above-mentioned problems with general inclusions, we will not allow $\top$ to be used as a concept name.[7]

The following sequence of constraints illustrates the consistency checking algorithm applied to the KB consisting of constraints (1) and (2) below.

(1)	$X : \forall R.C \sqcap \forall \in .\exists \ni .D$	
(2)	$def(D, \exists R.\neg C)$	
(3)	$X : \forall R.C$	$(\sqcap)$:(1)
(4)	$X : \forall \in .\exists \ni .D$	$(\sqcap)$:(1)
(5)	$\{X\} : \exists \ni .D$	$(\forall_{\in\{\}})$:(4)
(6)	$(\{X\}, Y) : \ni$	$(\exists)$:(5)
(7)	$Y : D$	$(\exists)$:(5)
(8)	$Y : \{X\}$	$(\ni)$:(6)
(9)	$Y = X$	$(\{\})$:(8)
(10)	$X : \exists R.\neg C$	(def):(2),(7),(9)
(11)	$(X, Z) : R$	$(\exists)$:(10)
(12)	$Z : \neg C$	$(\exists)$:(10)
(13)	$Z : C$	$(\forall)$:(11),(3)
(14)	$fail$	$(clash_{\neg})$:(12),(13)

3.1 Soundness and completeness

The *termination* and *soundness* of the algorithm are easy to prove. Its *completeness* is established by constructing a *canonical interpretation* $\mathcal{I}_S$ for each clash-free and complete constraint system S. Note that the propagation rules deal with so-called *"extended" constraints*, i.e. constraints of the form $X : Y$ and $(X, Y) : R$ where X and Y can be not just names, but also "extended" individuals represented by arbitrarily nested singletons like $\{\ldots\{Z\}\ldots\}$ (if we disallow extended individuals, we loose the completeness of the algorithm).

We can extend the "name function" ν to the set $Names^{\{\}}$ of "extended" individuals and thus talk about the name (intension) $\{X\}^{\nu}$ of an "extended" individual $\{X\}$. The extended name function must satisfy the constraint $\mathcal{V}(\{X\}^{\nu}) = \{X^{\nu}\}$.

The canonical interpretation $\mathcal{I}_S$ is defined by

$$(X^{\nu}, Y^{\nu}) \in \varepsilon(R) \quad \text{iff} \quad (X,Y) : R \textit{ is in } S \qquad (4)$$

$$(X^{\nu}, Y^{\nu}) \in \varepsilon(\in) \quad \text{iff} \quad (X : Y \textit{ is in } S) \vee Y = \{X\} \qquad (5)$$

for $X, Y \in Names^{\{\}}$ and $R \neq \in, \ni$. The extension of concept names can thus be obtained as:

$$\begin{aligned} \varepsilon(CN) &= \mathcal{V}(CN^{\nu}) = \{X^{\nu} \in \Delta \mid (X^{\nu}, CN^{\nu}) \in \varepsilon(\in)\} \\ &= \{X^{\nu} \in \Delta \mid (X : CN \textit{ is in } S) \vee CN = \{X\}\}. \end{aligned}$$

ε extends naturally to concept terms according to (3).

Testing the consistency of $\mathcal{ALCO}_{\in}$ knowledge bases w.r.t. the proposed semantics is therefore decidable.

4 Conclusions

Extending description logics with concept reification is essential for developing domain-independent and reusable models. Nevertheless, it has not been extensively studied, mainly due to the semantical problems posed by its inherent higher-order features and because of the complications in the reasoning algorithms.

The semantical problems related to the higher-order features implicit in reification are solved by defining a first-order semantics which ensures the decidability of the main inference services. We have also described sound and complete inference algorithms for the reified terminological language $\mathcal{ALCO}_{\in}$ (but the algorithms can be extended to more expressive languages).

In our view, concept reification represents an essential element for bridging the gap between description logics and (deductive) object-oriented databases.

It also makes description logics expressive enough to be used for developing *generic* problem solving models [Badea and Ţilivea, 1996] and even libraries of such models.

[7] We also disallow iff-definitions since these would enable us to define a concept name $D = E \sqcup \neg E$ equivalent to $\top$, and the above problems with expressing valid concepts C using $D : \forall \ni .C$ reappear.

Acknowledgments

Thanks are due to Doina Ţilivea and Alon Levy for interesting discussions as well as to anonymous reviewers for their helpful comments and especially for pointing out the paper [Franconi, 1993] (which also introduces an explicit membership role, but does not provide a complete reasoning algorithm).

References

[Badea, 1996] Badea L. *E_xClaim: a hybrid language for knowledge representation and reasoning using description logics.* Proc. ECAI'96 Workshop on Validation, Verification and Refinement of KBS, Budapest, 1996.

[Badea and Ţilivea, 1996] Badea L., Ţilivea D. *E_xClaim: a language for operationalizing CommonKADS expertise models using description logics.* PEKADS Report 4.5.1.

[Beeri, 1990] Beeri C. *A formal approach to object-oriented databases.* Data & Knowledge Eng. 5 (1990) 353-382.

[Brachman and Schmolze, 1985] Brachman R.J., Schmolze J.G. *An Overview of the KL-ONE Knowledge Representation System.* Cognitive Science 9(2), 171-216, 1985.

[Brachman *et al.*, 1991] Brachman R.J., e.a. *Living with CLASSIC: When and How to Use a KL-ONE like Language,* in Sowa J.F. (ed) *Principles of Semantic Networks,* Morgan Kaufmann 1991.

[Buchheit *et al.*, 1993] Buchheit M., Donini F.M., Schaerf A. *Decidable Reasoning in Terminological Knowledge Representation Systems.* J. of AI Research 1 (1993), 109-138.

[De Giacomo and Lenzerini, 1995] De Giacomo G., Lenzerini M. *What's in an aggregate: foundations for description logics with tuples and sets.* Proc. IJCAI'95, 801-807.

[Franconi, 1993] Franconi E. *A treatment of plurals and plural quantifications based on a theory of collections.* Minds and Machines, 1993 special issue on KR for NL, 453-474.

[Kobsa, 1991] Kobsa A. *Reification in SB-ONE.* In Proc. Int. Workshop on Terminological Logics, 72-74, DFKI D-91-13.

[Kifer *et al.*, 1995] Kifer M., Lausen G., Wu J. *Logical foundations of object-oriented and frame-based languages.* Journal of the ACM, May 1995.

[Schmidt-Schauß and Smolka, 1991] Schmidt-Schauß M., Smolka G. *Attributive concept descriptions with complements.* Artificial Intelligence 48 (1), 1-26, 1991.