Reasoning on UML Class Diagrams in Description Logics

Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
lastname@dis.uniroma1.it

Abstract. In this paper[1] we formalize UML class diagrams in terms of a logic belonging to Description Logics, which are subsets of First-Order Logic that have been thoroughly investigated in Knowledge Representation. The logic we have devised is specifically tailored towards the high expressiveness of UML information structuring mechanisms, and allows one to formally model important properties which typically can only be specified by means of qualifiers. The logic is equipped with decidable reasoning procedures which can be profitably exploited in reasoning on UML class diagrams. This makes it possible to provide computer aided support during the application design phase in order to automatically detect relevant properties, such as inconsistencies and redundancies.

1 Introduction

The *Unified Modeling Language* (UML) is the de facto standard formalism for object-oriented modeling [2, 14]. There is a vast consensus on the need for a precise semantics for UML [12, 17], in particular for UML class diagrams. Indeed, several types of formalization of UML class diagrams have been proposed in the literature [11–13, 9]. Many of them have been proved very useful with respect to the task of establishing a common understanding of the formal meaning of UML constructs. However, to the best of our knowledge, none of them has the explicit goal of building a solid basis for allowing automated reasoning techniques, based on algorithms that are sound and complete wrt the semantics, to be applicable to UML class diagrams.

In this paper, we propose a new formalization of UML class diagrams in terms of a particular formal logic of the family of Description Logics (DLs). DLs[2] have been proposed as successors of semantic network systems like KL-ONE, with an explicit model-theoretic semantics. The research on these logics has resulted in a number of automated reasoning systems [18, 19, 15, 16], that have been successfully tested in various application domains (see e.g., [21, 22, 20]). Our goal is to exploit the deductive capabilities of DL systems, and show

[1] A full version of this paper can be found in [3].
[2] See http://dl.kr.org for the home page of Description Logics.

that effective reasoning can be carried out on UML class diagrams, so as to provide support during the specification phase of software development.

In DLs, the domain of interest is modeled by means of *concepts* and *relations*, which denote classes of objects and relation between objects, respectively. Generally speaking, a DL is formed by three basic components:

- A *description language*, which specifies how to construct complex concept and relationship expressions (also called simply concepts and relationships), by starting from a set of atomic symbols and by applying suitable constructors,
- a *knowledge specification mechanism*, which specifies how to construct a DL knowledge base, in which properties of concepts and relationships are asserted, and
- a set of *automatic reasoning procedures*, which are sound, complete and terminating.

The set of allowed constructors characterizes the expressive power of the description language. Various languages have been considered by the DL community, and numerous papers investigate the relationship between expressive power and computational complexity of reasoning (see [10] for a survey).

Several works point out that DLs can be profitably used to provide both formal semantics and reasoning support to formalisms in areas such as Natural Language, Configuration Management, Database Management, Software Engineering. For example, [7, 8] illustrates the use of DLs for database modeling. However, DLs have not been applied to the Unified Modeling Language (UML) (with the exception of [5]). In this work we concentrate on UML class diagrams for the conceptual perspective. Hence, we do not deal with those features that are relevant for the implementation perspective, such as public, protected, and private qualifiers for methods and attributes. For such UML class diagrams we present a formalization of UML in terms of DLs. In particular, we show how to capture the constructs of UML class diagrams by using a Description Logic that is equipped with n-ary relations. The DL we have adopt is specifically tailored towards the high expressiveness of UML information structuring mechanisms, and allows one to formally model important additional properties, such has disjointness of classes, or partitions of classes into subclasses, that are typically specified by means of constraints in UML class diagrams. In spite of the expressiveness required, the logic proposed admits decidable reasoning procedures. Overall, the formalization in DLs of UML class diagrams provides us with a rigorous logical framework for representing and automatically reasoning on UML class specifications. Such a formalization can be considered as the basic steps towards developing intelligent tools that provide computer aided reasoning support during the application design phase, in order to automatically detect relevant properties, such as inconsistencies and redundancies.

The paper is organized as follows: in Section 2 we give an overview of the Description Logic we use, called $\mathcal{DLR}$. In Sections 3, 4, 5 and 6, we illustrate the formalization of UML class diagrams in terms of $\mathcal{DLR}$, focusing on classes, associations, generalization, and constraints, respectively. In Section 7 we discuss the use of the reasoning procedures associated to $\mathcal{DLR}$ in order to support the specification of UML class diagrams. Section 8 concludes the paper.

2 The Description Logic $\mathcal{DLR}$

In this paper we adopt a DL, here called $\mathcal{DLR}$, presented in [6], which is a variant of logic originally introduced in [4]. The basic elements of $\mathcal{DLR}$ are *concepts* (unary relations), and *n-ary relations*. We assume to deal with a finite set of atomic relations and atomic concepts, denoted by P and A, respectively. Arbitrary relations (of given arity between 2 and n_{max}), denoted by R, and arbitrary concepts, denoted by C, are built according to the following syntax:

$$\begin{aligned} R &::= \top_n \mid P \mid (i/n : C) \mid \neg R \mid R_1 \sqcap R_2 \\ C &::= \top_1 \mid A \mid \neg C \mid C_1 \sqcap C_2 \mid (\leq k\,[i]R) \end{aligned}$$

where i denotes a component of a relation, i.e., an integer between 1 and n_{max}, n denotes the *arity* of a relation, i.e., an integer between 2 and n_{max}, and k denotes a non-negative integer. We consider only concepts and relations that are *well-typed*, which means that (i) only relations of the same arity n are combined to form expressions of type $R_1 \sqcap R_2$ (which inherit the arity n), and (ii) $i \leq n$ whenever i denotes a component of a relation of arity n.

We also make use of the following abbreviations:

$$\begin{array}{rcl} C_1 \sqcup C_2 & \text{for} & \neg(\neg C_1 \sqcap \neg C_2) \\ C_1 \Rightarrow C_2 & \text{for} & \neg C_1 \sqcup C_2 \\ (\geq k\,[i]R) & \text{for} & \neg(\leq k{-}1\,[i]R) \\ \exists[i]R & \text{for} & (\geq 1\,[i]R) \\ \forall[i]R & \text{for} & \neg\exists[i]\neg R \end{array}$$

Moreover, we abbreviate $(i/n : C)$ with $(i : C)$, when n is clear from the context.

A *$\mathcal{DLR}$ knowledge base* (KB) is constituted by a finite set of *inclusion assertions*, where each assertion has one of the forms:

$$R_1 \sqsubseteq R_2 \qquad\qquad\qquad C_1 \sqsubseteq C_2$$

with R_1 and R_2 of the same arity.

Besides inclusion assertions, $\mathcal{DLR}$ KBs allow for assertions expressing identification constraints and functional dependencies.

$$
\begin{array}{rlrl}
\top_n^{\mathcal{I}} & \subseteq (\Delta^{\mathcal{I}})^n & \top_1^{\mathcal{I}} & = \Delta^{\mathcal{I}} \\
P^{\mathcal{I}} & \subseteq \top_n^{\mathcal{I}} & A^{\mathcal{I}} & \subseteq \Delta^{\mathcal{I}} \\
(i/n\!:C)^{\mathcal{I}} & = \{t \in \top_n^{\mathcal{I}} \mid t[i] \in C^{\mathcal{I}}\} & (\neg C)^{\mathcal{I}} & = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(\neg R)^{\mathcal{I}} & = \top_n^{\mathcal{I}} \setminus R^{\mathcal{I}} & (C_1 \sqcap C_2)^{\mathcal{I}} & = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\
(R_1 \sqcap R_2)^{\mathcal{I}} & = R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}} & (\leq k\,[i]R)^{\mathcal{I}} & = \{a \in \Delta^{\mathcal{I}} \mid \sharp\{t \in R_1^{\mathcal{I}} \mid t[i] = a\} \leq k\}
\end{array}
$$

Fig. 1. Semantic rules for $\mathcal{DLR}$ (P, R, R_1, and R_2 have arity n)

An *identification assertion* on a concept has the form:

$$(\mathbf{id}\ C\ [i_1]R_1, \ldots, [i_h]R_h)$$

where C is a concept, each R_j is a relation, and each i_j denotes one component of R_j. Intuitively, such an assertion states that no two different instances of C agree on the participation to $R_1, \ldots, R_h$. In other words, if a is an instance of C that is the i_j-th component of a tuple t_j of R_j, for $j \in \{1, \ldots, h\}$, and b is an instance of C that is the i_j-th component of a tuple s_j of R_j, for $j \in \{1, \ldots, h\}$, and for each j, t_j agrees with s_j in all components different from i_j, then a and b coincide.

A *functional dependency assertion* on a relation has the form:

$$(\mathbf{fd}\ R\ i_1, \ldots, i_h \to j)$$

where R is a relation, $h \geq 2$, and $i_1, \ldots, i_h, j$ denote components of R. The assertion imposes that two tuples of R that agree on the components $i_1, \ldots, i_h$, agree also on the component j.

Note that unary functional dependencies (i.e., functional dependencies with $h = 1$) are ruled out in $\mathcal{DLR}$, since these lead to undecidability of reasoning [6]. Note also that the right hand side of a functional dependency contains a single element. However, this is not a limitation, because any functional dependency with more than one element in the right hand side can always be split into several dependencies of the above form.

The semantics of $\mathcal{DLR}$ is specified through the notion of interpretation. An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of a $\mathcal{DLR}$ KB $\mathcal{K}$ is constituted by an *interpretation domain* $\Delta^{\mathcal{I}}$ and an *interpretation function* $\cdot^{\mathcal{I}}$ that assigns to each concept C a subset $C^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ and to each relation R of arity n a subset $R^{\mathcal{I}}$ of $(\Delta^{\mathcal{I}})^n$, such that the conditions in Figure 1 are satisfied. (In the figure, $t[i]$ denotes the i-th component of tuple t.) We observe that $\top_1$ denotes the interpretation domain, while $\top_n$, for $n > 1$, does *not* denote the n-Cartesian product of the domain, but only a subset of it, that covers all relations of arity n. It follows, from this property, that the "$\neg$" constructor on relations is used to express difference of relations, rather than complement.

To specify the semantics of a KB we first define when an interpretation satisfies an assertion as follows:

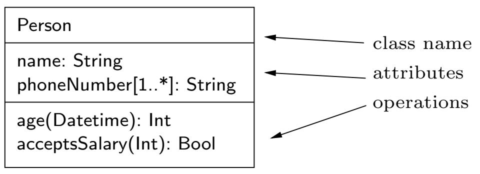

Fig. 2. Representation of a class in UML

- An interpretation $\mathcal{I}$ *satisfies* an inclusion assertion $R_1 \sqsubseteq R_2$ (resp. $C_1 \sqsubseteq C_2$) if $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$ (resp. $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$).
- An interpretation $\mathcal{I}$ *satisfies* the assertion (**id** C $[i_1]R_1, \dots, [i_h]R_h$) if for all $a, b \in C^{\mathcal{I}}$ and for all $t_1, s_1 \in R_1^{\mathcal{I}}, \dots, t_h, s_h \in R_h^{\mathcal{I}}$ we have that:

$$\left.\begin{array}{l} a = t_1[i_1] = \cdots = t_h[i_h], \\ b = s_1[i_1] = \cdots = s_h[i_h], \\ t_j[i] = s_j[i], \text{ for } j \in \{1, \dots, h\}, \text{ and for } i \neq i_j \end{array}\right\} \text{ implies } a = b$$

- An interpretation $\mathcal{I}$ *satisfies* the assertion (**fd** R $i_1, \dots, i_h \to j$) if for all $t, s \in R^{\mathcal{I}}$, we have that:

$$t[i_1] = s[i_1], \ \dots, \ t[i_h] = s[i_h] \quad \text{implies} \quad t[j] = s[j]$$

An interpretation that satisfies all assertions in a KB $\mathcal{K}$ is called a *model* of $\mathcal{K}$.

Several reasoning services are applicable to $\mathcal{DLR}$ KBs. The most important ones are KB satisfiability and logical implication. A KB $\mathcal{K}$ is *satisfiable* if there exists a model of $\mathcal{K}$. A concept C is *satisfiable* in a KB $\mathcal{K}$ if there is a model $\mathcal{I}$ of $\mathcal{K}$ such that $C^{\mathcal{I}}$ is nonempty. A concept C_1 is *subsumed by* a concept C_2 in a KB $\mathcal{K}$ if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{K}$. An assertion α is *logically implied* by $\mathcal{K}$ if all models of $\mathcal{K}$ satisfy α. One can easily verify that logical implication and KB unsatisfiability are mutually reducible.

One of the distinguishing features of $\mathcal{DLR}$ is that it is equipped with reasoning algorithms that are sound and complete wrt to the semantics. Such algorithms allow one to decide all the above reasoning tasks in deterministic exponential time [6]. Indeed, the proposed algorithms are computationally optimal, since reasoning in $\mathcal{DLR}$ is EXPTIME-complete [4].

3 Classes

A *class* in an UML class diagram denotes a *sets of objects* with common features. A class is graphically rendered as a rectangle divided into three parts, as shown for example in Figure 2. The first part contains the *name* of the class, which has to be unique in the whole diagram. The second part contains the *attributes* of the class, each denoted by a name (possibly followed by the *multiplicity*, between square brackets) and with an associated *class*, which indicates the domain of the attribute values. For example, the attribute phoneNumber[1..*]: String means that each instance of the class has at least one phone number, and possibly more, and that each phone numbers is an instance of String. If not otherwise specified, attributes are *single-valued*. The third part contains the *operations* of the class, i.e., the operations associated to the objects of the class. An operation definition has the form:

$$\textit{operation-name}(\textit{parameter-list})\text{: }(\textit{return-list})$$

Observe that an operation may return a *tuple* of objects as result.

An UML class is represented by a $\mathcal{DLR}$ concept. This follows naturally from the fact that both UML classes and $\mathcal{DLR}$ concepts denote *sets of objects*.

An UML *attribute* a of type C' for a class C associates to each instance of C, zero, one, or more instances of a class C'. An optional *multiplicity* $[i..j]$ for a specifies that a associates to each instance of C, at least i and most j instances of C'. When the multiplicity is missing, $[1..1]$ is assumed, i.e., the attribute is *mandatory* and *single-valued*.

To formalize attributes we have to think of an attribute a of type C' for a class C as a binary relation between instances of C and instances of C'. We capture such a binary relation by means of a binary relation a of $\mathcal{DLR}$. To specify the type of the attribute we use the assertion:

$$C \sqsubseteq \forall[1](a \Rightarrow (2 : C'))$$

Such an assertion specifies precisely that, for each instance c of the concept C, all objects related to c by a, are instances of C'. Note that an attribute name is not necessarily unique in the whole schema, and hence two different classes could have the same attribute, possibly of different types. This situation is correctly captured by the formalization in $\mathcal{DLR}$.

To specify the multiplicity $[i..j]$ associated to the attribute we add the assertion:

$$C \sqsubseteq (\geq i\,[1]a) \sqcap (\leq j\,[1]a)$$

Such an assertion specifies that each instance of C participates at least i times and at most j times to relation a via component 1. If $i = 0$, i.e., the attribute is *optional*, we omit the first conjunct, and if $j = *$ we omit the second one.

An operation of a class is a function from the objects of the class to which the operation is associated, and possibly additional parameters, to tuples of objects. In class diagrams, the code associated to the operation is not considered and typically, what is represented is only the signature of the operation.

In $\mathcal{DLR}$, we model operations by means of $\mathcal{DLR}$ relations. Let

$$f(P_1, \ldots, P_m) : (R_1, \ldots, R_n)$$

be an operation of a class C that has m parameters belonging to the classes $P_1, \ldots, P_m$ respectively and n return values belonging to $R_1, \ldots, R_n$ respectively. We formalize such an operation as a $\mathcal{DLR}$ relation, named $\mathtt{op}_{f(P_1,\ldots,P_m):(R_1,\ldots,R_n)}$, of arity $m+n+1$ among instances of the $\mathcal{DLR}$ concepts $C, P_1, \ldots, P_m, R_1, \ldots, R_n$. On such a relation we enforce the following assertions:

- An assertion imposing the correct types to parameters and return values:

$$\begin{aligned} C \;\sqsubseteq\; & \forall[1](\mathtt{op}_{f(P_1,\ldots,P_m):(R_1,\ldots,R_n)} \Rightarrow \\ & \quad ((2\!:\!P_1) \sqcap \cdots \sqcap (m+1\!:\!P_m) \sqcap (m+2\!:\!R_1) \sqcap \cdots \sqcap (m+n+1\!:\!R_n)) \end{aligned}$$

- Assertions imposing that invoking the operation on a given object with given parameters determines in a unique way each return value (i.e., the relation corresponding to the operation is in fact a function from the invocation object and the parameters to the returned values):

$$(\mathbf{fd}\ \mathtt{op}_{f(P_1,\ldots,P_m):(R_1,\ldots,R_n)}\ 1,\ldots,m+1 \rightarrow m+2)$$
$$\cdots$$
$$(\mathbf{fd}\ \mathtt{op}_{f(P_1,\ldots,P_m):(R_1,\ldots,R_n)}\ 1,\ldots,m+1 \rightarrow m+n+1)$$

  These functional dependencies are determined only by the number of parameters and the number of result values, and not by the specific class for which the operation is defined, nor by the types of parameters and result values.

The *overloading* of operations does not pose any difficulty in the formalization since an operation is represented in $\mathcal{DLR}$ by a relation having as name the whole signature of the operation, which consists not only the name of the operation but also the parameter and return value types. Observe that the formalization of operations in $\mathcal{DLR}$ correctly allows one to have operations with the same name or even with the same signature in two different classes.

4 Associations and Aggregations

An *association* in UML, graphically rendered as in Figure 3, is a relation between the instances of two or more classes. An association often has a related *association class* that describes properties of the association such as attributes, operations, etc. An *aggregation* in UML, graphically rendered as in Figure 4, is a binary relation between the instances of two classes, denoting a part-whole relationship, i.e., a relationship that specifies that each instance of a class is made up of a set of instances of another class.

Observe that names of associations and names of aggregations (as names of classes) are *unique*. In other words there cannot be two associations/aggregations with the same name.

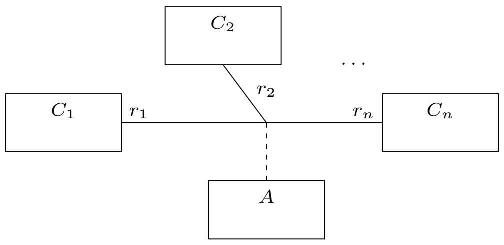

Fig. 3. Association in UML

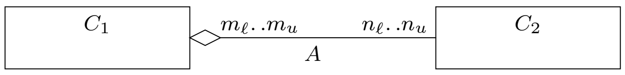

Fig. 4. Aggregation in UML

We first concentrate on the formalization of aggregations, which are simpler to model than general associations. An aggregation A, saying that instances of the class C_1 have components that are instances of the class C_2, is formalized in $\mathcal{DLR}$ by means of a binary relation A together with the following assertion:

$$A \sqsubseteq (1{:}C_1) \sqcap (2{:}C_2).$$

Note that the distinction between the contained class and the containing class is not lost. Indeed, we simply use the following convention: *the first argument of the relation is the containing class.*

As we have seen for class attributes, the multiplicity of an aggregation can be easily expressed in $\mathcal{DLR}$. For example, the multiplicities shown in Figure 4 are formalized by means of the assertions:

$$\begin{array}{lcl} C_1 & \sqsubseteq & (\geq n_\ell\,[1]A) \sqcap (\leq n_u\,[1]A) \\ C_2 & \sqsubseteq & (\geq m_\ell\,[2]A) \sqcap (\leq m_u\,[2]A) \end{array}$$

We can use a similar assertion for a multiplicity on the participation of instances of C_1 for each given instance of C_2.

Observe that, in the formalization in $\mathcal{DLR}$ of aggregation, role names do not play any role. If we want to keep track of them in the formalization, it suffices to consider them as convenient abbreviations for the components of the $\mathcal{DLR}$ relation modeling the aggregation.

Next we focus on *associations*. Since associations have often a related association class, we formalize associations in $\mathcal{DLR}$ by reifying each association A into a $\mathcal{DLR}$ concept A with suitable properties. We represent an association among n classes $C_1, \ldots, C_n$, as shown in Figure 3, by introducing a concept A and n *binary* relations $r_1, \ldots, r_n$, one for each component of the association

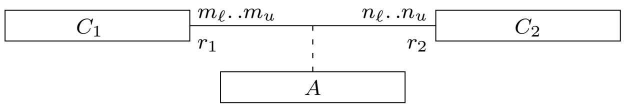

Fig. 5. Binary association in UML

A [3]. Each binary relation r_i has C_i as its first component and A as its second component. Then we enforce the following assertion:

$$\begin{aligned} C \ \sqsubseteq\ & \exists[1]r_1 \sqcap (\leq 1\,[1]r_1) \sqcap \forall[1](r_1 \Rightarrow (2\!:\!C_1)) \sqcap \\ & \exists[1]r_2 \sqcap (\leq 1\,[1]r_2) \sqcap \forall[1](r_2 \Rightarrow (2\!:\!C_2)) \sqcap \\ & \quad \vdots \\ & \exists[1]r_n \sqcap (\leq 1\,[1]r_n) \sqcap \forall[1](r_n \Rightarrow (2\!:\!C_n)) \end{aligned}$$

where $\exists[1]r_i$ (with $i \in \{1,\ldots,n\}$) specifies that the concept A must have all components $r_1,\ldots,r_n$ of the association A, $(\leq 1\,[1]r_i)$ (with $i \in \{1,\ldots,n\}$) specifies that each such component is single-valued, and $\forall[1](r_i \Rightarrow (2\!:\!C_i))$ (with $i \in \{1,\ldots,n\}$) specifies the class each component has to belong to. Finally, we use the assertion

$$(\mathbf{id}\ A\ [1]r_1,\ldots,[1]r_n)$$

to specify that each instance of the concept A indeed represents a *distinct* tuple of the corresponding association.

We can easily represent a multiplicity on a binary UML association, by imposing suitable number restrictions on the $\mathcal{DLR}$ relations modeling the components of the association. Differently from aggregation, however, the names of such relations (which correspond to roles) are unique wrt to the association only, not the entire diagram. Hence we have to state such constraints in $\mathcal{DLR}$ in a slightly different way.

The multiplicities shown in Figure 5 are captured as follows:

$$\begin{aligned} C_1 \ &\sqsubseteq\ (\geq n_\ell\,[1](r_1 \sqcap (2\!:\!A))) \sqcap (\leq n_u\,[1](r_1 \sqcap (2\!:\!A))) \\ C_2 \ &\sqsubseteq\ (\geq m_\ell\,[1](r_2 \sqcap (2\!:\!A))) \sqcap (\leq m_u\,[1](r_2 \sqcap (2\!:\!A))) \end{aligned}$$

5 Generalization and Inheritance

In UML one can use *generalization* between a parent class and a child class to specify that each instance of the child class is also an instance of the parent class. Hence, the instances of the child class inherit the properties of the parent

[3] These relations may have the name of the roles of the association if available in the UML diagram, or an arbitrary name if role names are not available. In any case, we preserve the possibility of using the same role name in different associations.

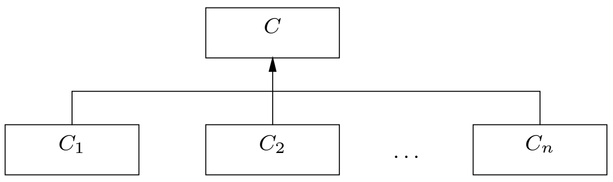

Fig. 6. A class hierarchy in UML

class, but typically they satisfy additional properties that do not hold for the parent class.

Generalization is naturally supported in $\mathcal{DLR}$. If an UML class C_2 generalizes a class C_1, we can express this by the $\mathcal{DLR}$ assertion:

$$C_1 \sqsubseteq C_2$$

Inheritance between $\mathcal{DLR}$ concepts works exactly as inheritance between UML classes. This is an obvious consequence of the semantics of $\sqsubseteq$ which is based on subsetting. Indeed, in $\mathcal{DLR}$, given an assertion $C_1 \sqsubseteq C_2$, every tuple in a relation having C_2 as i-th argument type may have as i-th component an instance of C_1, which is in fact also an instance of C_2. As a consequence, in the formalization, each attribute or operation of C_2, and each aggregation and association involving C_2 is correctly inherited by C_1. Observe that the formalization in $\mathcal{DLR}$ also captures directly inheritance among association classes, which are treated exactly as all other classes, and multiple inheritance between classes (including association classes).

Moreover in UML, one can group several generalizations into a class hierarchy, as shown in Figure 6. Such a hierarchy is captured in $\mathcal{DLR}$ by a set of inclusion assertions, one between each child class and the parent class:

$$C_i \sqsubseteq C \qquad \text{for each } i \in \{1, \ldots, n\}$$

In UML it is possible to override attributes or operations of a superclass. That is, it is possible to specialize an attribute or an operation for the subclass. From the conceptual point of view such a specialization needs to remain compatible with the original definition of the attribute/operation, i.e., the attribute/operation of the subclass can only be a *restriction* of the corresponding attribute/operation belonging to the superclass. For attributes, this means that one can restrict the type of the attribute to be a subclass of the original type, or restrict the multiplicity wrt to the one specified for the superclass. For operations, while keeping the same signature, one may restrict (by means of constraints) the return types and possibly also the argument types to be subclasses of the original ones[4].

[4] Observe that restricting the argument types corresponds, in the implementation of the operation, to restrict the preconditions for the applicability of the operation.

We illustrate by means of an example how one can correctly model such forms of overriding in $\mathcal{DLR}$. Let C be an UML class that has an operation $f(C_1, C_2) : C_3$, and C' be a subclass of C (and hence inherits the operation). In $\mathcal{DLR}$, we model the situation by introducing a concept C and a relation $\mathtt{op}_{f(C_1,C_2):C_3}$ and a concept C' with suitable assertions including $C' \sqsubseteq C$. As a consequence instances of the concept C' inherits the properties that hold for instances of C including the participation in the relation $\mathtt{op}_{f(C_1,C_2):C_3}$. Suppose now that in the UML class diagram C' we override the method $f(C_1, C_2) : C_3$ by requiring that the result value belongs to a subclass C_3' of C_3. We can capture this in $\mathcal{DLR}$ by adding the assertion:

$$C' \;\sqsubseteq\; \forall[1](\mathtt{op}_{f(C_1,C_2):C_3} \Rightarrow (4 : C_3'))$$

6 Constraints

In UML it is possible to add information to a class diagram by using *constraints*. In general, constraints are used to express in an informal way information which cannot be expressed by other constructs of UML class diagrams. We discuss here common types of constraints that occur in UML class diagrams and how they can be taken into account when formalizing class diagrams in $\mathcal{DLR}$.

Generally, in UML class diagrams, unless specified otherwise by a constraint, two classes may have common instances, i.e., they are *not disjoint*. If a constraint imposes the disjointness of two classes, say C and C', this can be formalized in $\mathcal{DLR}$ by means of the assertion

$$C \;\sqsubseteq\; \neg C'$$

Observe that disjointness constraints are often used in class hierarchies. For example, consider a class hierarchy formed by a class C and n subclasses of C, $C_1, \ldots, C_n$. We may want to require that $C_1, \ldots, C_n$ are *mutually disjoint*. In $\mathcal{DLR}$, this can be expressed by the assertions

$$C_i \;\sqsubseteq\; \neg C_j \qquad \text{for each } i, j \in \{1, \ldots, n\} \text{ with } i \neq j$$

Disjointness of classes is just one example of *negative information*. Again, by exploiting the expressive power of $\mathcal{DLR}$, we can express additional forms of negative information, usually not considered in UML, by introducing suitable assertions. For example, we can enforce that no instance of a class C has an attribute a by means of the assertion

$$C \;\sqsubseteq\; \neg\exists[1]a$$

Analogously, one can assert that no instance of a class is involved in a given association or aggregation.

Turning again the attention to generalization hierarchies, by default, in UML a generalization hierarchy is open, in the sense that there may be instances of the superclass that are not instances of any of the subclasses. This allows for extending the schema more easily, in the sense that the introduction of a new subclass does not change the semantics of the superclass. However, in specific situations, it may happen that in a generalization hierarchy, the superclass C is a covering of the subclasses $C_1, \ldots, C_n$. We can represent such a situation in $\mathcal{DLR}$ by simply including the additional assertion

$$C \sqsubseteq C_1 \sqcup \cdots \sqcup C_n$$

The above assertion models a form of *disjunctive information*: each instance of C is either an instance of C_1, or an instance of C_2, ...or an instance of C_n. Other forms of disjunctive information can be modeled by exploiting the expressive power of $\mathcal{DLR}$. For example, that an attribute a is present only for a specified set $C_1, \ldots, C_n$ of classes can be modeled by suitably using union of classes as follows:

$$\exists[1]a \sqsubseteq C_1 \sqcup \cdots \sqcup C_n$$

Keys are a modeling notion that is very common in databases, and they are used to express that certain attributes uniquely identify the instances of a class. We can exploit the expressive power of $\mathcal{DLR}$ in order to associate keys to classes. If an attribute a is a key for a class C this means that there is no pair of instances of C that have the same value for a. We can capture this in $\mathcal{DLR}$ by means of the assertion $(\mathbf{id}\ C\ [1]a)$. More generally, we are able to specify that a *set* of attributes $\{a_1, \ldots, a_n\}$ is a key for C; in this case we use the assertion: $(\mathbf{id}\ C\ [1]a_1, \ldots, [1]a_n)$

As already seen, constraints that correspond to the specialization of the type of an attribute or its multiplicity can be represented in $\mathcal{DLR}$. Similarly, consider the case of a class C participating in an aggregation A with a class D, and where C and D have subclasses C' and D' respectively, related via an aggregation A'. A *subset constraint* from A' to A can be modeled correctly in $\mathcal{DLR}$ by means of the assertion $A \sqsubseteq A'$, involving the two binary relations A and A' that represent the aggregations.

In general, one can exploit the expressive power of $\mathcal{DLR}$ to formalize several types of constraints that allow one to better represent the application semantics and that are typically not dealt with in a formal way. Observe that this allows one to take such constraints fully into account when reasoning on the class diagram.

7 Reasoning on Class Diagrams

Traditional CASE tools support the designer with a user-friendly graphical environment and provide powerful means to access different kinds of repositories that store information associated to the elements of the developed project. However, no support for higher level activities related to managing the complexity of the design is provided. In particular, the burden of checking relevant properties of class diagrams, such as consistency or redundancy, is left to the responsibility of the designer. Thus, the formalization in $\mathcal{DLR}$ of UML class diagrams, and the fact that properties of inheritance and relevant types of constraints are perfectly captured by the formalization in $\mathcal{DLR}$ and the associated reasoning tasks, provide the ability to reason on class diagrams. This represents a significant improvement and it is a first step towards the development of modeling tools that offer an automated reasoning support to the designer in his modeling activity. By exploiting the $\mathcal{DLR}$ reasoning services various kinds of checks can be performed on the class diagram.

A class diagram is *consistent*, if its classes can be populated without violating any of the constraints in the diagram. Observe that the interaction of various types of constraints may make it very difficult to detect inconsistencies. A *class* is *consistent* if it can be populated without violating any of the constraints in the class diagram. The inconsistency of a class may be due to a design error or due to over-constraining. In any case, the designer can be forced to remove the inconsistency, either by correcting the error, or by relaxing some constraints, or by deleting the class, thus removing redundancy from the schema. By exploiting the formalization in $\mathcal{DLR}$, class consistency can be checked by verifying satisfiability of the corresponding concept in the $\mathcal{DLR}$ KB representing the class diagram. Similarly, consistency of the class diagram corresponds to consistency of the $\mathcal{DLR}$ KB.

Two classes are *equivalent* if they denote the same set of instances whenever the constraints imposed by the class diagram are satisfied. Determining equivalence of two classes allows for their merging, thus reducing the complexity of the schema. A class C_1 is *subsumed by* a class C_2 if, whenever the constraints imposed by the class diagram are satisfied, the extension of C_1 is a subset of the extension of C_2. Such a subsumption allows one to deduce that properties for C_1 hold also for C_2. It is also the basis for a *classification* of all the classes in a diagram. Such a classification, as in any object-oriented approach, can be exploited in several ways within the modeling process [1]. Class equivalence, subsumption, and hence classification, can be checked by verifying equivalence and subsumption in $\mathcal{DLR}$.

A property is a *logical consequence* of a class diagram if it holds whenever all constraints specified in the diagram are satisfied. As an example, consider a class C generalizing classes $C_1, \ldots, C_n$, and assume that a constraint specifies that it is complete. If an attribute a is defined as mandatory for all classes $C_1, \ldots, C_n$, then it follows logically that the same attribute is mandatory also for class C, even if not explicitly present in the schema. Determining logical consequence is useful on the one hand to reduce the complexity of the schema by removing those constraints that logically follow from other ones, and on the other hand it can be used to make properties explicit that are implicit in the schema, thus enhancing its readability. Logical consequence can be captured by logical implication in $\mathcal{DLR}$, and determining logical implication is at the basis of all types of reasoning that a $\mathcal{DLR}$ reasoning system can provide. In particular, observe that all reasoning tasks we have considered above can be rephrased in terms of logical consequence.

8 Conclusions

We have proposed a new formalization of UML class diagrams in terms of a particular formal logic of the family of Description Logics. Notably such a logic has sound, complete and decidable reasoning procedures. These reasoning procedures can be favorably exploited for developing intelligent system that support automated reasoning on UML class diagrams, so as to provide support during the specification phase of software development. We have already started experimenting such systems. In particular, we have represented UML diagrams in $\mathcal{DLR}$ and used DL reasoners, specifically FACT [18] and RACER [16], for reasoning on UML class diagrams. Although such DL reasoners do not yet incorporate all features required by our formalization (e.g., support for identifiers), the first results are encouraging.

References

1. Sonia Bergamaschi and Bernhard Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Applied Intelligence*, 4(2):185–203, 1994.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Publ. Co., Reading, Massachussetts, 1998.
3. Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. A formal framework for reasoning on UML class diagrams. Submitted for publication, 2001.
4. Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *Proc. of the 17th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'98)*, pages 149–158, 1998.
5. Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI'99)*, pages 84–89, 1999.
6. Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Identification constraints and functional dependencies in description logics. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001)*, 2001. To appear.
7. Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Description logics for conceptual data modeling. In Jan Chomicki and Günter Saake, editors, *Logics for Databases and Information Systems*, pages 229–264. Kluwer Academic Publisher, 1998.
8. Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Unifying class-based representation formalisms. *J. of Artificial Intelligence Research*, 11:199–240, 1999.

9. Tony Clark and Andy S. Evans. Foundations of the Unified Modeling Language. In David Duke and Andy Evans, editors, *Proc. of the 2nd Northern Formal Methods Workshop*. Springer-Verlag, 1997.
10. Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Reasoning in description logics. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, Studies in Logic, Language and Information, pages 193–238. CSLI Publications, 1996.
11. Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Proc. of the OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.
12. Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-modelling semantics of UML. In H. Kilov, editor, *Behavioural Specifications for Businesses and Systems*, chapter 2. Kluwer Academic Publisher, 1999.
13. Andy S. Evans. Reasoning with UML class diagrams. In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT'98)*. IEEE Computer Society Press, 1998.
14. Martin Fowler and Kendall Scott. *UML Distilled – Applying the Standard Object Modeling Laguage*. Addison Wesley Publ. Co., Reading, Massachussetts, 1997.
15. Volker Haarslev and Ralf Möller. High performance reasoning with very large knowledge bases: A practical case study. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001)*, 2001.
16. Volker Haarslev and Ralf Möller. RACER system description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, 2001.
17. David Harel and Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Rehovot, Israel, 2000.
18. Ian Horrocks. Using an expressive description logic: FaCT or fiction? In *Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.
19. Ian Horrocks and Peter F. Patel-Schneider. Optimizing description logic subsumption. *J. of Logic and Computation*, 9(3):267–293, 1999.
20. Thomas Kirk, Alon Y. Levy, Yehoshua Sagiv, and Divesh Srivastava. The Information Manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, pages 85–91, 1995.
21. D. McGuinness and J. Wright. Conceptual modelling for configuration: A description logic-based approach. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing Journal*, 12:333–344, 1998.
22. Ulrike Sattler. *Terminological Knowledge Representation Systems in a Process Engineering Application*. PhD thesis, LuFG Theoretical Computer Science, RWTH-Aachen, 1998.