A Formal Framework for Reasoning on UML Class Diagrams

Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and 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 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

There is a vast consensus on the need for a precise semantics for UML [9,12], in particular for UML class diagrams. Indeed, several types of formalization of UML class diagrams have been proposed in the literature [8,9,10,6]. 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 (DL). DLs[1] 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 [13,14,11], which have been successfully tested in various application domains (see e.g., [17,18,16]). Our long term goal is to exploit the deductive capabilities of DL systems, and show 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 *relationships*, which denote classes of objects and relations, respectively. Generally speaking, a DL is formed by three basic components:

[1] See http://dl.kr.org for the home page of Description Logics.

M.-S. Hacid et al. (Eds.): ISMIS 2002, LNAI 2366, pp. 503–513, 2002.

- 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* provided by the DL.

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 [7] 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, [5] illustrates the use of DLs for database modeling. However, to the best of our knowlegde, DLs have not been applied to the Unified Modeling Language (UML) (with the exception of [3]). The goal of this work is to present a formalization of UML class diagrams in terms of DLs. In particular, we show how to map the constructs of a class diagram onto those of the EXPTIME decidable DL $\mathcal{DLR}$ [2,4]. The mapping provides us with a rigorous logical framework for representing and automatically reasoning on UML class specifications. 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 constraints. 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.

2 Classes

In this paper we concentrate on 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.

A *class* in an UML class diagram denotes a *sets of objects* with common features. The specification of a class contains the *name* of the class, which has to be unique in the whole diagram, and 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. The specification contains also the *operations* of the class, i.e., the operations associated to the objects of the class. An operation definition has the form:

operation-name(*parameter-list*): (*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 diagram, 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. Observe that for attributes with multiplicity $[0..*]$ we omit the whole assertion, and that, when the multiplicity is missing the above assertion becomes:

$$C \sqsubseteq \exists[1]a \sqcap (\leq 1\,[1]a)$$

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:

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

Fig. 1. Binary association and aggregation in UML

– 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 $\mathcal{DLR}$ allows one to have operations with the same name or even with the same signature in two different classes.

3 Associations and Aggregations

An *association* in UML 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. A binary association A between the instances of two classes C_1 and C_2 is graphically rendered as in the left hand side of Figure 1, where the class A is the association class related to the association, r_1 and r_2 are the *role names* of C_1 and C_2 respectively, i.e., they specify the role that each class plays within the relation R, and where the *multiplicity* $a..b$ specifies that each instance of class C_1 can participate at least a times and at most b times to relation A; $c..d$ has an analogous meaning for class C_2.

An *aggregation* in UML is a binary association 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. An aggregation is graphically rendered as shown in the right hand side of Figure 1, where the diamond indicates the *containing class*, opposed to the *contained class*. The multiplicity has the same meaning as in associations. As for associations, also for aggregation it is possible to define role names which denote the role each class plays in the aggregation.

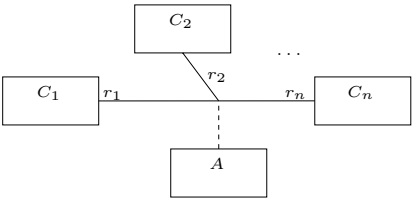

Fig. 2. Association in UML

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.

Next we turn to the formalization in $\mathcal{DLR}$. An aggregation A as depicted in Figure 1, without considering multiplicities, is formalized in $\mathcal{DLR}$ by means of a binary relation A on which the following assertion is enforced:

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

Note that, to distinguish between the contained class and the containing class, we simply use the convention that *the first argument of the relation is the containing class*. To express the multiplicity $n_l..n_u$ on the participation of instances of C_2 for each given instance of C_1, we use the assertion

$$C_1 \sqsubseteq (\geq n_l\,[1]A) \sqcap (\leq n_u\,[1]A)$$

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. Let us consider the association shown in Figure 2. We represent it in $\mathcal{DLR}$ by introducing a concept A and n *binary* relations $r_1, \ldots, r_n$, one for each component of the association A [2]. Then we enforce the following assertion:

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

[2] 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.

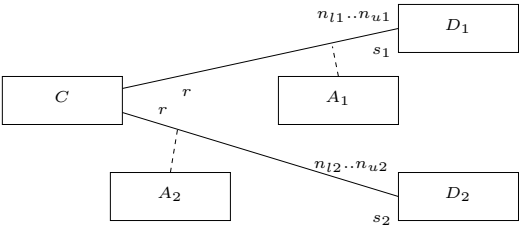

Fig. 3. Multiplicity in aggregation

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 A represents a *distinct* tuple in $C_1 \times \cdots \times C_n$.

We can easily represent in $\mathcal{DLR}$ a multiplicity on a binary association, by imposing a number restriction on the 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 more involved way.

Suppose we have a situation like that in Figure 3. Consider the association A_1 and the constraint saying that for each instance of C there can be at least n_{l1} and at most n_{u1} instances of D_1 related by A_1 to it. We capture this constraint as follows:

$$C \sqsubseteq (\geq n_{l1}\,[2](r \sqcap (1:A_1))) \sqcap (\leq n_{u1}\,[2](r \sqcap (1:A_1)))$$

Observe that nothing prevents C to partecipate to a different association A_2 with the same role r but with different multiplicity $n_{l2}..n_{u2}$. Observe that this is modeled by the totally unrelated assertion:

$$C \sqsubseteq (\geq n_{l2}\,[2](r \sqcap (1:A_2))) \sqcap (\leq n_{u2}\,[2](r \sqcap (1:A_2)))$$

4 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 class, but typically they satisfy additional properties that in general 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:

$$C_1 \sqsubseteq C_2$$

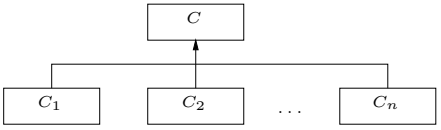

Fig. 4. A class hierarchy in UML

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, given an assertion $C_1 \sqsubseteq C_2$, every tuple in a $\mathcal{DLR}$ 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 4. Such a hierarchy is captured in $\mathcal{DLR}$ by a set of inclusion assertions, one between each child class and the parent class:

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

We discuss in Section 5 how to formalize in $\mathcal{DLR}$ additional properties of a class hierarchy, such as mutual disjointness between the child classes, or covering of the parent class.

5 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 $\mathcal{DLR}$.

Often, when defining generalizations between classes, we need to add additional constraints among the involved classes. For example, for the class hierarchy in Figure 4, a constraint may express that $C_1, \ldots, C_n$ are *mutually disjoint*. In $\mathcal{DLR}$, such a relationship can be expressed by the assertions $C_i \sqsubseteq \neg C_j$, for each $i, j \in \{1, \ldots, n\}$ with $i \neq j$.

In general, in UML, if not otherwise specified 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 $C \sqsubseteq \neg C'$.

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 diagram 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 $C \sqsubseteq C_1 \sqcup \cdots \sqcup C_n$ Such an 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: $\exists[1]a \sqsubseteq C_1 \sqcup \cdots \sqcup C_n$.

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 discussed in Section 4, 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.

More generally, 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.

6 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 (see below), 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 formal-

ization in $\mathcal{DLR}$ and the associated reasoning tasks, provides the ability to reason on class diagrams. This represents a significant improvement and is a first step towards the development of modeling tools that offer an automated reasoning support to the designer in his modeling activity.

We briefly discuss the tasks that can be performed by exploiting the reasoning capabilities of a $\mathcal{DLR}$ reasoner [14,15], and that allow a modeling tool to take over tasks traditionally left to the responsibility of the designer. Such a tool may construct from a class diagram a $\mathcal{DLR}$ knowledge base, and manage it in a way completely transparent to the designer. By exploiting the $\mathcal{DLR}$ reasoning services various kinds of checks can be performed on the class diagram. [3]

Consistency of 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. By exploiting the formalization in $\mathcal{DLR}$, the consistency of a class diagram can be checked by checking the satisfiability of the corresponding $\mathcal{DLR}$ knowledge base.

Class Consistency. 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 diagram. Exploiting the formalization in $\mathcal{DLR}$, class consistency can be checked by checking satisfiability of the corresponding concept in the $\mathcal{DLR}$ knowledge base representing the class diagram.

Class Equivalence. 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 diagram. Again, checking class equivalence amounts to check the equivalence in $\mathcal{DLR}$ of the corresponding concepts.

Class Subsumption. 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]. Subsumption, and hence classification, can be checked by verifying subsumption in $\mathcal{DLR}$.

Logical Consequence. 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 the generalization hierarchy depicted in Figure 4 and assume that a constraint specifies that it is complete. If an attribute a is defined as mandatory for all classes $C_1, \ldots, C_n$

[3] A prototype design tool with such a kind of automated reasoning support is available at `http://www.cs.man.ac.uk/~franconi/icom/`.

then it follows logically that the same attribute is mandatory also for class C, even if not explicitly present in the diagram. Determining logical consequence is useful on the one hand to reduce the complexity of the diagram by removing those constraints that logically follow from other ones, and on the other hand it can be used to explicit properties that are implicit in the diagram, 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.

7 Conclusions

We have proposed a new formalization of UML class diagrams in terms of a particular formal logic of the family of Description Logics. Our long term goal is to exploit the deductive capabilities of DL systems, thus showing that effective reasoning can be carried out on UML class diagrams, so as to provide support during the specification phase of software development. As a first step, we have shown in this paper how to map the constructs of a class diagram onto those of Description Logics. The mapping provides us with a rigorous logical framework for representing and automatically reasoning on UML class specifications.

We have already started experimenting our approach. In particular, we have used FACT for representing and reasoning on class diagrams. Although FACT does not yet incorporate all features required by our formalization (e.g., keys), the first results are encouraging. In particular, we have been able to draw interesting, non-trivial inferences on class diagrams containing about 50 classes. More experiments are under way, and we plan to report on them in the near future.

In the future, we aim at extending our formalization in order to capture further aspects of the UML. Our first step in this direction will be to add to our formal framework the possibility of modeling and reasoning on objects and links (i.e., instances of classes and associations).

References

1. S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Applied Intelligence*, 4(2):185–203, 1994.
2. D. Calvanese, G. De Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. In *Proc. of PODS'98*, pages 149–158, 1998.
3. D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In *Proc. of IJCAI'99*, pages 84–89, 1999.
4. D. Calvanese, G. De Giacomo, and M. Lenzerini. Identification constraints and functional dependencies in description logics. In *Proc. of IJCAI 2001*, pages 155–160, 2001.
5. D. Calvanese, M. Lenzerini, and D. Nardi. Description logics for conceptual data modeling. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 229–264. Kluwer Academic Publisher, 1998.

6. T. Clark and A. S. Evans. Foundations of the Unified Modeling Language. In D. Duke and A. Evans, editors, *Proc. of the 2nd Northern Formal Methods Workshop*. Springer-Verlag, 1997.
7. F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation*, Studies in Logic, Language and Information, pages 193–238. CSLI Publications, 1996.
8. A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a formal modeling notation. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proc. of the OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.
9. A. Evans, R. France, K. Lano, and B. Rumpe. Meta-modelling semantics of UML. In H. Kilov, editor, *Behavioural Specifications for Businesses and Systems*, chapter 2. Kluwer Academic Publisher, 1999.
10. A. S. Evans. Reasoning with UML class diagrams. In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT'98)*. IEEE Computer Society Press, 1998.
11. V. Haarslev and R. Möller. Expressive ABox reasoning with number restrictions, role hierarchies, and transitively closed roles. In *Proc. of KR 2000*, pages 273–284, 2000.
12. D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Rehovot, Israel, 2000.
13. I. Horrocks. Using an expressive description logic: FaCT or fiction? In *Proc. of KR'98*, pages 636–647, 1998.
14. I. Horrocks and P. F. Patel-Schneider. Optimizing description logic subsumption. *J. of Log. and Comp.*, 9(3):267–293, 1999.
15. I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proc. of LPAR'99*, number 1705 in LNAI, pages 161–180. Springer-Verlag, 1999.
16. T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, pages 85–91, 1995.
17. D. L. McGuinness and J. R. Wright. An industrial strength description logic-based configuration platform. *IEEE Intelligent Systems*, pages 69–77, 1998.
18. U. Sattler. *Terminological Knowledge Representation Systems in a Process Engineering Application*. PhD thesis, LuFG Theoretical Computer Science, RWTH-Aachen, Germany, 1998.