10

Conceptual Modeling with Description Logics

Alex Borgida

Ronald J. Brachman

Abstract

The purpose of the chapter is to help someone familiar with DLs to understand the issues involved in developing an ontology for some universe of discourse, which is to become a conceptual model or knowledge base represented and reasoned with using Description Logics.

We briefly review the purposes and history of conceptual modeling, and then use the domain of a university library to illustrate an approach to conceptual modeling that combines general ideas of object-centered modeling with a look at special modeling/ontological problems, and DL-specific solutions to them.

Among the ontological issues considered are the nature of individuals, concept specialization, non-binary relationships, materialization, aspects of part-whole relationships, and epistemic aspects of individual knowledge.

10.1 Background

Information modeling is concerned with the construction of computer-based symbol structures that model some part of the real world. We refer to such symbol structures as information bases, generalizing the term from related terms in Computer Science, such as databases and knowledge bases. Moreover, we shall refer to the part of a real world being modeled by an information base as its *universe of discourse (UofD)*. The information base is checked for consistency, and sometimes queried and updated through special-purpose languages. As with all models, the advantage of information models is that they abstract away irrelevant details, and allow more efficient examination of both the current, as well as past and projected future states of the UofD.

An information model is built up using some language, and this language influences (more or less subtly) the kinds of details that are considered. For example, early information models (e.g., relational data model) were built on conventional programming notions such as records, and as a result focused on the implementation aspects of the information being captured, as opposed to the representational aspects. *Conceptual models* offer more expressive facilities for modeling applications *directly and naturally* [Hammer and McLeod, 1981], and for *structuring* information bases. These languages provide semantic terms for modeling an application, such as entity and relationship (or even activity, agent and goal), as well as means for organizing information.

Conceptual models play an important part in a variety of areas. The following is a brief summary of these areas, as reviewed in [Mylopoulos, 1998]:

- Artificial intelligence programs turned out to require the representation of a great deal of human knowledge in order to act "intelligently." As a result, they relied on conceptual models built up using knowledge representation languages, such as semantic networks—directed graphs labeled with natural language identifiers. DLs are the historical descendants of attempts to formalize semantic networks.
- The design of database systems was seen to have as an important initial phase the construction of a "*conceptual level schema*," which determined the information needs of the users, and which was eventually converted to a physical implementation schema. Chen's Entity-Relationship model [Chen, 1976], and later semantic data models [Hull and King, 1987] were the result of efforts in this direction.
- More generally, the development of all software has an initial *requirements acquisition* stage, which nowadays is seen to consist of a *requirements model* that describes the relationship of the proposed system and its environment. The environment in this case is likely to be a conceptual model.
- Independently, the object-oriented software community has also proposed viewing software components (classes/objects) as models of real-world entities. This was evident in the features of Simula, the first object-oriented programming language, and became a cornerstone of most object-oriented techniques, including the current leader, UML [Rumbaugh *et al.*, 1998].

One interesting aspect of conceptual modeling in the database context has been the identification of a number of *abstraction mechanisms* that support the development of large models by abstracting details initially, and then introducing them in a step-wise and systematic manner. Among the important abstractions are the following:

- thinking of objects as wholes, not just a collection of their attributes/components ("aggregation");
- abstracting away the detailed differences between individuals, so that a class can represent the commonalities ("classification"[1]);

[1] This term is used in a completely different way than in DL terminology, where it refers to the DL-KBMS service of finding the lowest subsumers of a concept or individual.

- abstracting the commonalities of several classes into a superclass ("generalization").

An important claim regarding the benefits of abstraction in conceptual modeling is that it results in a *structured* information model, which is easier to build and maintain. Interestingly, DLs further this goal by supporting the *automatic* classification of concepts with respect to others, thereby revealing generalizations that may not have been recognized by the modeler.

10.2 Elementary Description Logics modeling

Most conceptual models, including DLs, subscribe to an *object-centered* view of the world. Thus, their ontology includes notions like individual objects, which are associated with each other through (usually binary) relationships, and which are grouped into classes. In this chapter we use freely the notation and concrete syntax of Description Logics (see Appendix), and extend it with additional constructs that make it more suitable for modeling.

In the domain of a university library, we might encounter a particular person, GIANNI, or a particular book, BOOK23. Most of the information about the state of the world is captured by the inter-relationships between individuals, such as GIANNI having borrowed BOOK23. Binary relationships are modeled directly in DLs using *roles* and *attributes*: either GIANNI is a filler of the lentTo role for BOOK23, or BOOK23 is the filler of the hasBorrowed role for GIANNI. Note that lentTo and hasBorrowed are converse relationships, and this should be captured in a model, since frequently one wants to access information about associations in either direction. In DLs, this is accomplished using the role constructor **inverse**:

$$\mathsf{hasBorrowed} \quad \equiv \quad (\textbf{inverse}\ \mathsf{lentTo})$$

Note that in order to avoid inadvertent errors during modeling due to confusion between a role and its converse, or between a role and the kind of values filling it, one heuristic is to use a natural language name that is asymmetric, and adopt the convention that the relationship $R(a, b)$ should be read as "a R b"; therefore in the above case lentTo(BOOK23,GIANNI) reads "BOOK23 lentTo GIANNI," while lentTo(GIANNI,BOOK23) reads "GIANNI lentTo BOOK23," which makes it clear that the first but not the second is the proper way to use the role lentTo in the model. On the other hand, loan would be a poor choice of a role identifier because one could equally well imagine loan as a role of books or of persons, so that neither loan(GIANNI,BOOK23) nor loan(BOOK23,GIANNI) "read" properly.

In addition, it is always important to distinguish functional relationships, like lentTo (a book can be loaned to at most one borrower at any time) from non-functional ones, like hasBorrowed. This is done most cleanly if the particular DL being used allows the declaration of functional relationships, sometimes called "attributes" or "features." Attributes themselves come in two flavors: total and partial. Thus lentTo is a partial attribute because a book can only be loaned to one person, but may not be on loan at some point of time; on the other hand, every book has to have an ISBN-Nr. It is important to check which interpretation of attributes is offered by the particular DL being used. In the rest of this chapter we assume that attributes are total, and the concept constructor **the** will be used as an abbreviation, so that (**the** p C) is equivalent to the conjunction of (**all** p C), (**at-most** 1 p) and (**at-least** 1 p).

Individuals are grouped into classes; for example, Book might be a natural class in our domain. Classes usually abstract out common properties of their instances, e.g., every book in the library has a call number. Classes are modeled by concepts in DLs, and usually the common properties are expressed as subsumption axioms about the concept. These conditions usually involve super-concepts, as well as the kinds of values that can fill roles, and limits on the number of (various kinds of) role fillers. By design, these are exactly the kinds of things that can be expressed using DL constructors:

```
/* Books are materials, whose callNr is an integer */
Book  ⊑  (and Material
              (the callNr Integer)
              ...)
```

As mentioned in earlier chapters, one of the fundamental properties of DLs is support for the distinction between primitive/atomic concepts—for which instances can only be declared explicitly—and defined concepts—which offer necessary and sufficient conditions for membership. So, for example, we can distinguish between the notion of "borrower" as someone who *can* borrow a book (an approved customer of the library)

```
/* Borrower is previously declared as a primitive concept.
   Here it is indicated what restrictions on borrowing are in force for this concept */
Borrower  ⊑  (all hasBorrowed Book)
```

from the notion of "borrower" as someone who has actually borrowed a book from the library

```
/* Borrower is defined as someone who has borrowed books */
Borrower  ≡  (and (all hasBorrowed Book)
                  (at-least 1 hasBorrowed))
```

We now turn to considering a variety of more subtle issues that arise when modeling a domain. Almost all of these issues arise independent of the modeling language used; what we emphasize here is the range of possible solutions in the DL framework.

10.3 Individuals in the world

Some individuals are quite concrete, like a particular person, Gianni, or a particular copy of a book. Some are more abstract, like the subject matter covered by a book. The important property of most individuals is that they have an *identity*, which allows them to be distinguished from one another and to be *counted*.

Modeling of individuals is therefore made easier if they have unique identifiers. Unfortunately, this may not always be the case. For example, if one sees on a bookshelf two brand new copies of a book, which may not be distinguishable by any property known to us, one can still say that they are different copies of the book. In information management systems, and sometimes in the real world, this leads us to devise some kind of "extrinsic" identification scheme. For example, books on the library shelf are assigned a copy number. In this paper, as in object-oriented software systems, we will tend to assign arbitrary internal identifiers to objects, such as GIANNI or BOOK23.

The following examples concerning books show that what constitutes a relevant individual in a UofD depends very much on what we want to do with the information. In a domain concerning literature courses, one might consider something like Dickens' HARD-TIMES as the kind of individual appearing on an assigned reading list. For an Internet book-seller interface, it is necessary to consider a more concrete level of modeling—that of book editions, since, these may have different prices. Finally, in a library, we need to keep track of actual physical book copies.

In the last two cases, one must then decide whether to model books (as opposed to editions or copies) as individuals, or as concepts that have the other kinds of individuals as instances. A general heuristic is that if we expect certain notions to be counted, then they must be modeled as individuals. Another heuristic is that notions that do not have an inception time are usually modeled as concepts.

Modeling of the particular kind of relationship that exists, for example, between a book and its editions is further examined in Section 10.7.2.

10.3.1 Values vs. objects

It is important to distinguish what we may call *individual objects*, such as GIANNI, from *values*, such as integers, strings, lists, tuples, etc. The former have an associated intrinsic and immutable identity, and need to be created in the knowledge base. The later are "eternal" mathematical abstractions, whose identity is determined by some procedure usually involving the structure of the individual. For example, the two strings "`abc`" and "`abc`" are the same individual value because they have the same sequence of characters; similarly for dates, such as 1925/12/20, which can be considered as 3-tuples.

Many DLs only support reasoning with objects, in which case composite values such as dates need to be modeled as objects with attributes for day, month and year. The danger here is that, for example, multiple date individuals can be created with the same attribute values, in which case they are treated as distinct for the purposes of counting and identity checking, resulting in reasoning anomalies. Implemented DLs such as CLASSIC support values from the underlying programming language (so-called "host values"), and relatively simple concept hierarchies over them. Others, such as $\mathcal{ALC}(\mathcal{D})$ [Baader and Hanschke, 1991a] and $\mathcal{SHOQ}(\mathcal{D})$ [Horrocks and Sattler, 2001] allow attributes to have values from so-called "concrete domains," which can contain entirely new kinds of values. These concrete domains are required to have their own, independent reasoners, which are then coupled with the DL reasoner.

Equally desirable would be mathematical types such as sets, bags, sequences, and tuples, as supported by modern programming languages and certain semantic data models.

Currently, only the highly expressive $\mathcal{DLR}$ languages support notions such as n-tuples and recursive fixed-point structures, from which one can build lists, trees, etc. Even here, one can only provide the description of concepts ("list of Persons"), as opposed to the specification of individuals ("the list [GIANNI,ANNA]").

10.3.2 Individuals vs. references to them

It is important to distinguish an individual from various references to it: Gianni vs. "the person whose first name is the 5 letter string "`Gianni`" vs. "the borrower with library card number 32245" vs. "the chairman of the Psychology Department." This distinction becomes crucial when we express relationships: there is a difference between relating two objects and relating their names, because we usually want objects to remain related, even if names are changed. Thus "GIANNI hasBorrowed BOOK25" is different from "card-holder number 32245 hasBorrowed BOOK25," because if Gianni gets a new card (after losing his old one, say), then the relationship between Gianni and the book is lost. So, in general, one should always deal with the individual objects, unless there is a bijection between a class of objects and a class of referents to them, and this bijection is universal (it always exists) and is unchanging[1]. Kent [Kent, 1979] has eloquently argued the importance of these issues in record-based database systems, and shows that in the real world such bijections are much rarer than assumed. For example, Neumann [Neumann, 1992]

[1] Such bijections are exactly the "keys" used in the database context.

reports that the same US social security number (the prototypical identifier for persons in the USA) has been issued to two people, who even have the same name and birth-date!

Conversely, in some cases one wants to state relationships between intensional references, rather than specific objects. For example, we might want to say that, in general, the director of the library is the head of the book selection committee (COMMITTEE3). If Gianni happens to be the current director of the NBU library, then asserting headOf(GIANNI,COMMITTEE3) is improper because, among others, if Gianni steps down as director, according to the above model he would still be committee chair. One needs the ability to use unnamed expressions as arguments of relationships, along the lines of the predicate logic expression headOf(directorOf(NBU-LIBRARY),COMMITTEE3).

In DLs, intensional referents can be expressed as roles that are applied to individuals. (The roles may often be complex chains, resulting from the composition of atomic roles, as in "the zipCode of the address of the lentTo.") Assuming that we use the notation NBU-LIBRARY.director to refer to the filler of the director role for the NBU-LIBRARY individual, the above relationship is actually stated as "NBU-LIBRARY.director is identical to COMMITTEE3.head." The concept constructor **same-as**, indicating that two chains of roles have the same value, is used to express exactly such relationships, so the above situation might be modeled, naively, using the concept (**same-as** director head). The problem is that we need a single individual of which to assert this property, yet it is libraries that have directors while committees have heads. In such situations, in DLs one must find or create some chain of attributes relating the two individuals NBU-LIBRARY and COMMITTEE3. The natural relationship in this case is the attribute hasBookSelectionCommittee. Therefore the appropriate way of modeling this situation is

```
/* NBU-LIBRARY has book selection committee COMMITTEE3 */
hasBookSelectionCommittee(NBU-LIBRARY, COMMITTEE3)

/* NBU-LIBRARY.director equals
   NBU-LIBRARY.hasBookSelectionCommittee.head */
(same-as director (hasBookSelectionCommittee ∘ head))(NBU-LIBRARY)
```

10.4 Concepts

For the university library, some obvious classes of individuals include people, institutions, the material that can be loaned by the library, the staff, dates, library cards, and fines. These classes are normally modeled using atomic/primitive concepts in DLs.

It may be worth noting that in DLs the same individual may be an instance of multiple classes, without one being necessarily a subclass of another: some book might be an instance of both hard-cover and science books. This is in contrast with many other object-oriented software systems, where one is forced to create a special subclass for this notion, in order to guarantee a unique "minimal" class for every individual. However, this is not a modeling principle—it is an implementation obstacle.

10.4.1 Essential vs. incidental properties of concepts

As explained in the earlier example involving the two possible meanings for the term "borrower," an important feature of DLs is the ability to distinguish primitive from defined concepts, where the latter have necessary and sufficient conditions for concept membership.

For example, BookOnLoan might naturally be defined as

```
/* A book is on loan if it is borrowed by someone */
BookOnLoan  ≡  (and Book (at-least 1 lentTo))
```

Suppose that we also want to require that only hard-cover books can be loaned out. There seem to be two options for modeling this:

```
/* Option 1 — being hardcover is part of the definition */
BookOnLoan  ≡  (and Book
                    (at-least 1 lentTo)
                    (fills binding 'hardcover))

/* Option 2 — being hardcover is an additional necessary condition */
BookOnLoan  ≡  (and Book (at-least 1 lentTo))
BookOnLoan  ⊑  (fills binding 'hardcover)
```

The first approach is not quite right because being hardcover is an *incidental* property of books on loan, albeit one universally shared by all such objects. Among other things, this means that if the system is to recognize some individual book as being on loan, it is enough to know that it has been lent to someone—one does not also need to know it is hardcover. Hence the second modeling option is the right one, since, one can actually deduce that a book on loan is hardcover, if this was not known ahead of time.

The distinction between definitional and incidental properties is also important if we consider the task of classifying concepts into a taxonomy, since it has been argued that the taxonomy should not depend on contingent facts. This suggests that incidental properties, even universal inclusion assertions like the one for hardcover books in Option 2 above, should appear in the ABox, not the TBox defining the terminology.

Another subtle problem arises when there are multiple sufficient conditions for a concept. For example, suppose we associated a due date with books on loan (in the physical world, this might be recorded as a date stamped in the back of the book). Then encountering a book with a due date in the future would rightly classify it as a book on loan. If we model the due date as an attribute of books, which has a value only as long as the date is in the future, then we would represent this situation as

$$(\textbf{and}\ \mathsf{Book}\ (\textbf{at-least}\ 1\ \mathsf{dueDate})) \quad \sqsubseteq \quad \mathsf{BookOnLoan}$$

and, of course, requiring books on loan to have a due date would lead to

$$\mathsf{BookOnLoan} \quad \sqsubseteq \quad (\textbf{at-least}\ 1\ \mathsf{dueDate})$$

We thus have multiple sufficient conditions for being a book on loan, although one of them appears to be the primary definition.

10.4.2 Reified concepts and meta-roles

In some cases it seems natural to associate information with an entire concept, rather than with each of its individual instances. One situation where this arises is in capturing aggregate information, such as the count of current individual instances of the concept, or the average value of their attributes. In the library example, attributes such as numberOfBooks and mostRequestedBooks would fall into this category.

In some object-oriented systems this can be modeled directly because classes are themselves objects, and as such are instances of meta-classes and have meta-properties. Currently, DLs do not have a facility to treat classes as objects. One must therefore create a separate "meta-individual" that is related to the concept by some naming convention, for example. In our example, we would create the individual BOOK-CLASS-OBJECT, and then attach the information regarding numberOfBooks, mostRequestedBooks, etc., as roles of this individual. In the CLASSIC system, given a named concept, this meta-individual can be retrieved using a special, new knowledge base operation.

10.4.3 Concepts dependent on relationships

The following interesting modeling problem arises in many situations: some concepts, such as Book, stand on their own. Others, such as Borrower, rely on the *implied existence of some relation/event* (e.g., lending), which has a second argument, and from which their meaning is derived. It is important to discern this second category of concepts, and explicitly introduce the corresponding binary relationship in the model. In the data modeling literature (e.g., [Albano *et al.*, 1993]) categories of this second type, such as Borrower, are called “roles,” but to avoid confusion with DL roles, we will call them “*relationship-roles*.” The modeling of these will be considered further in Section 10.7.1.

10.5 Subconcepts

For many of the above concepts, there are specialized subconcepts representing subsets of individuals that are also of interest. For example, the concept Material (referring to the holdings of libraries) could be Book, Journal, Videotape, etc. In turn, Book may have subconcepts Monograph, EditedCollection, Proceedings, etc.[1] And Borrowers may be Institutions or Individuals, with the latter being divided into Faculty, Student, Staff.

There are a number of special aspects of the subclass relationship that should be modeled in order to properly capture the semantics of the UofD.

10.5.1 Disjointness of subconcepts

In many cases, subclasses are disjoint from each other. For example, Book and Journal are disjoint subclasses of Material. In DLs that support negation, this is modeled by adding the complement of one concept to the necessary properties of the other concept:

Book $\sqsubseteq$ **not** Journal

Often, entire collections of subclasses are disjoint[2]. For this purpose, some DLs provide the ability to describe disjointness by naming a *discriminator*, and a special declaration operation for primitive subclasses. For example, one might discriminate between various kinds of material on the basis of the medium as follows:

Print $\sqsubseteq$ (**disjointPrim** Material **in group** medium **with discriminant** paper);
Video $\sqsubseteq$ (**disjointPrim** Material **in group** medium **with discriminant** light);
Audio $\sqsubseteq$ (**disjointPrim** Material **in group** medium **with discriminant** sound);

At the same time, one might discriminate between different kinds of material on the basis of the format:

Book $\sqsubseteq$ (**disjointPrim** Material **in group** format **with discriminant** book);
Journal $\sqsubseteq$ (**disjointPrim** Material **in group** format **with discriminant** journal);
...

[1] For this section, we will think of the material to be loaned as physical individuals that can be carried out the door of the library, so to speak.
[2] This is especially the case at the top of the subclass hierarchy: Person, Material, etc.

Two points are worth making here: (i) the advantage of a syntax based on discriminators is that it avoids the multiplicative effect of having to state disjointness for every pair of disjoint concepts; (ii) as in the above example, it is important to allow during modeling for multiple groups of disjoint subconcepts for the same concept.

10.5.2 Covering by subconcepts

In addition to disjointness, it is natural to consider whether some set of subclasses fully covers the superclass. For example, we might want to say that Circulating material must be either short-term or long-term.

For DLs that support concept disjunction, this is easy:

$$\mathsf{Circulating} \quad \sqsubseteq \quad (\textbf{or}\ \mathsf{ShortTerm}\ \mathsf{LongTerm})$$

Note that since ShortTerm, in turn, has Circulating as a superclass, the possibility arises of modeling Circulating as a definition:

$$\mathsf{Circulating} \quad \equiv \quad (\textbf{or}\ \mathsf{ShortTerm}\ \mathsf{LongTerm})$$

However, this approach is not available for languages like CLASSIC, which avoid disjunction in order to gain tractable reasoning. We discuss in the next section an approach to the problem based on subconcept definitions and enumerated values.

10.5.3 Defined vs. primitive subconcepts

In the case of material that is either circulating or non-circulating, the name of the second class provides a hint: after introducing Material and Circulating as primitives, NonCirculating should be *defined*:

$$\begin{array}{lcl} \mathsf{Circulating} & \sqsubseteq & \mathsf{Material} \\ \mathsf{NonCirculating} & \equiv & (\textbf{and}\ \mathsf{Material}\ (\textbf{not}\ \mathsf{Circulating})) \end{array}$$

In this case, the DL can deduce both the disjointness of Circulating and NonCirculating, and the fact that Material is the union of Circulating and NonCirculating, without having stated anything explicitly about either. This shows clearly the power of a reasoning system that is capable of supporting definitions.

By joining covering and disjointness one gets the partitioning of a class by some group of subclasses. In some DLs—those supporting the constructor **one-of**—it is possible to simulate the effect of declaring concepts as partitioned into subconcepts through the use of a special attribute. For example, we could add the attribute format to Books, with an enumerated set of possible values:

$$\mathsf{Book} \quad \sqsubseteq \quad (\textbf{the}\ \mathsf{format}\ (\textbf{one-of}\ \texttt{'monograph 'journal 'editedCollection}))$$

and then define the corresponding subclasses:

$$\begin{array}{lcl}
\text{Monograph} & \equiv & (\textbf{and}\ \text{Book}\ (\textbf{fills}\ \text{format}\ \texttt{'monograph})) \\
\text{Journal} & \equiv & (\textbf{and}\ \text{Book}\ (\textbf{fills}\ \text{format}\ \texttt{'journal})) \\
\text{EditedCollection} & \equiv & (\textbf{and}\ \text{Book}\ (\textbf{fills}\ \text{format}\ \texttt{'editedCollection}))
\end{array}$$

These concepts will be disjoint because format can have at most one value, and they cover the original class Book, because format must have (at least) one value from among the set enumerated.

10.5.4 Dynamics of (sub)concept membership

When changes in the model are allowed, there is a distinction between concepts that represent inherent properties of objects that do not change over time (called "rigid" in [Guarino and Welty, 2000]) such as Book, and concepts that represent more transient properties, such as MisplacedBook. Note that while it is possible for a transient property to be a subconcept of rigid one, the converse does not make sense.

Standard DLs have not developed modeling tools for issues involving the dynamics of the world, and hence usually cannot represent such distinctions. DLs extended with the notion of time, such as [Artale and Franconi, 1998], are of course well suited to express them.

10.5.5 The structure of the subconcept hierarchy

Recent work by Guarino and Welty (e.g., [Guarino and Welty, 2000]) has presented several interesting ontological dimensions along which a concept can be positioned.

The dimensions are related to many of the topics we discuss elsewhere in this chapter, including the existence or absence of criteria for identifying individuals (viz. Section 10.3), the rigid vs. non-rigid nature of concept membership (viz. Section 10.5.4), the nature of the part-whole relationship (viz. Section 10.7.3), and aspects resembling relationship-roles (viz. Section 10.7.1).

The significance of these dimensions is that they can be used to both clarify the intended meaning of concepts in an ontology, and to better organize the taxonomy of primitive concepts. The conditions for proper taxonomies are based on observations such as "*a concept some of whose current instances may cease to be instances at some point in the future (e.g., Student) cannot subsume a concept whose membership cannot change (e.g., Person).*"

We refer the reader to the original paper for further details.

10.6 Modeling relationships

As mentioned earlier, binary relationships are modeled in DLs using roles and attributes. Just as with subclasses, there are a number of special constraints that are frequently expressed about relationships: cardinality constraints state the minimum and maximum number of objects that can be related via a role; domain constraints state the kinds of objects that can be related via a role; and inverse relationships between roles need to be recorded. For example, a book has exactly one title, which is a string, and exactly one call number, which is some value that depends on the cataloging technique used. On the other hand, there may be zero or more authors for a book:

$$\begin{aligned}\mathsf{Book} \;\sqsubseteq\; & (\textbf{and}\ (\textbf{the}\ \mathsf{title\ String})\\ & \quad (\textbf{the}\ \mathsf{callNr\ MaterialIdentifier})\\ & \quad (\textbf{all}\ \mathsf{author\ Person}))\end{aligned}$$

As mentioned in Section 10.2, we can use the attribute lentTo to model when someone borrows a book:

$$\mathsf{Book} \;\sqsubseteq\; (\textbf{all}\ \mathsf{lentTo\ Borrower})$$

Suppose we also want to record that the material in the library may be on loan, available or missing. This can be modeled by adding appropriate roles to the library:

$$\begin{aligned}\mathsf{Library} \;\sqsubseteq\; & (\textbf{and}\ (\textbf{all}\ \mathsf{hasOnLoan\ Material})\\ & \quad (\textbf{all}\ \mathsf{hasAvailable\ Material})\\ & \quad (\textbf{all}\ \mathsf{hasMissing\ Material}))\end{aligned}$$

In such a case we would like to say that these roles are non-overlapping. This could be accomplished through the use of a concept constructor **non-overlapping**, syntactically similar to **same-as**: (**non-overlapping** hasOnLoan hasAvailable). However, if only one library is involved, it would be better to model the situation using an appropriate subclass of Material, such as MissingMaterial, because we already have tools for modeling disjointness of subclasses, and reasoning with them is not inherently hard as is the case of general constructors such as **same-as** and **non-overlapping**.

10.6.1 Reified relationships

It is sometimes useful to be able to give "properties of properties." For example, when some material is lent to a borrower, it is useful to record on what date the loan took place and when the material is due back. In the Entity-Relationship approach this would be modeled by the creation of a relationship class, called Loan, which would have attributes onLoan, lentTo, as well as lentOn and dueOn, describing the loan. This can be thought of as the *reification* of the relationship, and results in the following DL class specification:

```
Loan  ⊑  (and (the lentTo Borrower)
              (the onLoan Material)
              (the lentOn Date)
              (the dueOn Date)
              (the NrOfRenewals (max 3)))
```

Unless the DL supports n-ary relations, reified relationships become essential when modeling associations that involve more than two objects, as would be the case, for example, if we had several libraries (or branches), and we wanted to record from which library the loan was made.

Reified relationships have the disadvantage of requiring the modeler to distinguish somehow the subset of attributes determining the relationship $R(a, b, \ldots)$, from those qualifying it. In the above case, we may imagine that Loan represents a binary relationship Loan(Borrower,Material) between lentTo and onLoan (in which case lentOn is there just to qualify the relation); alternatively, we may interpret Loan as a ternary relationship Loan(Borrower,Material,Date) between lentTo, onLoan and lentOn. The former records loans (a borrower may have a book at most once) while the latter records the *history* of loans. The notion of "keys/unique identifiers" from databases, as adapted to DLs [Borgida and Weddell, 1997] can be used for this task, by marking the collection of attributes that describe the relationship as a key.

We remark that the $\mathcal{DLR}$ description logic can express n-ary relationships directly, so it does not require reification for this purpose.

10.6.2 Role hierarchies

In many applications, two roles on the same concept may be related by the constraint that every filler of the first role must be a filler of the second role. For example, in the library domain, the fillers of the role hasOnShortTermLoan, recording a borrower's materials that need to be returned within a week, are also fillers of hasBorrowed, recording all the materials borrowed (this would be true by definition). Similarly, the editorInChief of a journal would be included in its editorialStaff.

One of the important features of frame knowledge representation schemes, and DLs in particular, is that they encourage the modeler to think of roles as first class citizens. This includes support for the notion of a role taxonomy (subroles). This is all the more reasonable, since once we reify a relationship, we would be allowed to create subconcepts of it at will.

As a result, the above kinds of constraints on the containment of role fillers can be modeled through the use of role hierarchies—a notion supported by most DLs, at least for primitive roles:

$$\mathsf{hasOnShortTermLoan} \sqsubseteq \mathsf{hasBorrowed}$$

10.7 Modeling ontological aspects of relationships

The material in this section deals with some special kinds of relationships and approaches to modeling them. The cognoscenti will recognize these as issues related to the ontological aspects of a UofD (constructs relating to the essence of objects), as opposed to epistemological aspects (constructs relating to the structure of objects), which are captured by notions such as InstanceOf and IS-A. The kinds of relationships to be discussed below do however occur relatively frequently, and pose difficulties to the uninitiated.

10.7.1 Relationship-roles

A subtle, but important distinction can be drawn between objects that *may* participate in a relationship (the domain restrictions on the role) and the objects that actually do take part in one or more relationships. For example, the objects participating in a lending relationship can be said to be playing certain "roles": LentObject and Borrower. It was exactly this second meaning of borrower—as a relationship-role—that was contrasted with the original meaning of "potential borrower" in our example of Section 10.2.

DLs allow one to define the relationship-roles associated with a relationship. In the case when the relationship is modeled by a regular DL role, such as borrowedBy, we can define lent objects as ones that are being borrowed, and borrowers, as objects that are the values of borrowedBy:

$$\begin{array}{lcl} \mathsf{LentObject} & \equiv & (\textbf{at-least}\ 1\ \mathsf{borrowedBy}) \\ \mathsf{Borrower} & \equiv & (\textbf{at-least}\ 1\ (\textbf{inverse}\ \mathsf{borrowedBy})) \end{array}$$

In the case of the reified Loan relationship, the definition of these classes would be

$$\begin{array}{lcl} \mathsf{LentObject} & \equiv & (\textbf{at-least}\ 1\ (\textbf{inverse}\ \mathsf{onLoan})) \\ \mathsf{Borrower} & \equiv & (\textbf{at-least}\ 1\ (\textbf{inverse}\ \mathsf{lentTo})) \end{array}$$

10.7.2 Materialization

There is a family of situations whose modeling is complicated by the fact that several concepts can be referred to by the same natural language term. For example, one might say "Shakespeare wrote 'Hamlet'," "The 'Hamlet' in London this season is a success," and " 'Hamlet' was cancelled tonight." But there is a difference between the abstract notion of the play 'Hamlet', various stagings of the play, and particular performances. Other familiar distinctions of this kind include the difference between an airline flight ("Air France flight 25 from Paris to London") and a particular "instance" of it—the one that will leave on May 24, 2002. Failure to model such differences can result in the same kind of problem that arises with any other form of ambiguity—inappropriate use in a context. So one can only buy tickets to play performances, but theatrical awards are given to stagings.

In each of these cases there is a relationship between a general notion (e.g., play staging) and 0-to-N more specific notions (e.g., performance of that play staging), which has been called *materialization*, and was investigated in [Pirotte *et al.*, 1994].

Let us first model some information that we would like to capture in the library domain:

```
/* Books have information about authors, etc. */
Book  ⊑  (and ...
               (all hasAuthors Person)
               (the hasTitle String))

/* Editions of books are related to the book (in a way yet to be specified)
   but have their own roles too */
BookEdition  ⊑  (and ...
                      (the publishedBy PublishingCompany)
                      (the isbnNr IsbnNumber)
                      (the format (one-of 'printed 'audio)))

/* Book copies are related to book editions, and in turn have their own roles */
BookCopy  ⊑  (and ...
                   (the callNr CallNumber)
                   (the atBranch LibraryBranch))
```

There are several alternative ways of proceeding with the modeling of such a UofD.

Since objects in each of these classes are seen to naturally have attributes like hasTitle, it is tempting to think of BookCopy as being a subclass of BookEdition so that this attribute is *inherited*. However, this would mean that each individual instance of BookCopy is a separate BookEdition, which seems wrong.

If we are *not* committed to modeling separate individual instances of each of these concepts, it is possible to combine their description into a single concept that records all the relevant information. So, for example, we could define Books to have all the attributes of the three concepts above, and thus really refer to book copies. (But see below.)

Finally, according to the results in [Pirotte *et al.*, 1994], a more appropriate approach is to view each edition of a book as determining a *subclass* of BookCopy. Each of these subclasses can then be viewed as an instance of BookEdition, for which it provides so-called “meta-roles.” Materialization is the combination of these ideas.

The materialization relationship can be modeled in DLs by a role materializationOf, connecting in our case book editions and books, and book copies and book editions. However, this sounds very unnatural when read out loud, so a better approach may be to create *subroles* of the general role materializationOf. This means that the above model would be completed by adding the following assertions

/* editionfOf is a kind of materialization relationship */
editionOf $\sqsubseteq$ materializationOf

/* Book editions are materializations of books */
BookEdition $\sqsubseteq$ (**the** editionOf Book)

/* copyOf is a kind of materialization relationship */
copyOf $\sqsubseteq$ materializationOf

/* Book copies are materializations of book editions */
BookCopy $\sqsubseteq$ (**the** copyOf BookEdition)

Often, the properties of the more abstract concept are inherited by the materialization. For example, the book edition, and then the book copy, has the same title and author as the book. In DLs, this relationships can be expressed by identifying the appropriate attribute values on the general and the materialized object:

BookEdition $\sqsubseteq$ (**same-as** hasTitle (editionOf $\circ$ hasTitle))

Several additional kinds of relationships between attributes of an object and its materialization are identified in [Pirotte *et al.*, 1994], but they are rather unclear and cannot be represented in DLs. Probably the most interesting is the case when an attribute of the more general concept has no correspondent on materialized individuals. For example, though a book edition may reasonably record the date when it was *first* and *last* printed, it seems very questionable to say that a book copy has a *last* printing date.

This looks like a case of meta-roles of the kind mentioned earlier. The main importance is that if one wants to have in the model attributes such as firstPrinting, then one cannot “melt” objects (book editions) into their various materializations (book copies), and is forced to model them separately.

10.7.3 Part-whole aggregation

The part-whole relationship distinguishes roles of a book such as its chapters, from others such as its publisher. There is a long history of discussions concerning this topic, with [Artale *et al.*, 1996b] being an excellent and comprehensive survey that considers, among other things, a variety of DL solutions to the problem. We present here some interesting observations.

Cognitive scientists have distinguished a variety of part-whole relationships, whose mixture has caused apparent paradoxes; according to one hypothesis these can be distinguished by differentiating three kinds of wholes—*complexes, collections* and *masses*—with parts called *components, members* and *quantities* respectively; furthermore parts can be *portions* (sharing intrinsic properties with the whole) and *segments.* Most physical objects, like book copies, are complexes of their parts (e.g., pages), but in the book domain we also find uses for collections in modeling books that are anthologies of other literary pieces.

In addition, one can qualify the nature of two aspects of the relationship between parts and wholes:

- Existence: A whole may depend on particular individual(s) for its continued existence and identity, as in the case when the part is irreplaceable (e.g., a book must have an author); or it may depend generically on a class of parts (e.g., a book copy must have a cover). Conversely, the part may depend on the whole for its existence (e.g., the chapter of a book). Finally, a part may belong exclusively to only one whole or it might be shared.
- Properties: Properties may be "inherited" from the whole to the part (e.g., ownedBy) or from the part to the whole (e.g., isDefective).

At the very least, the above provides a checklist of issues to consider whenever a part-whole relationship is encountered during modeling.

In the realm of Description Logics, Sattler [1995] offers an approach to dealing with these topics, exploiting various role-forming operators such as role hierarchies, role inverse, and transitive closure to capture the semantics of aggregation.

Specifically, special roles are introduced for the different kinds of part-whole relationships mentioned above: hasDComponent, hasDMember, hasDSegment, hasDQuantity, hasDStuff, hasDIngredient, where "D" stands for "direct." One then defines more complex relationships from these primitives:

$$\begin{aligned}\mathsf{hasComponent} &\equiv (\textbf{transitive-closure}\\ &\qquad (\textbf{or}_{role}\ \mathsf{hasDComponent}\ (\mathsf{hasDMember} \circ \mathsf{hasDComponent})))\\ \mathsf{hasPart} &\equiv (\textbf{or}_{role}\ \mathsf{hasComponent}\ \mathsf{hasMember}\ \cdots)\end{aligned}$$

indicating that members of collections of components are also components, and that hasPart is the union of the various sub-kinds of relationships.

Let us concentrate here on the component-of relationship, which is probably the one most frequently encountered in practical applications. We shall consider the table of contents of a book as an exemplar of a component attribute.

One idea is to declare attributes and roles that represent components (e.g., $\mathsf{tableOfContents}$) as specializations of $\mathsf{hasDComponent}$. This allows us to distinguish such component roles from other roles, like lentTo and $\mathsf{publisher}$.

Obviously, the inverses of such roles provide access from a part to its containing whole:

$$\begin{array}{lcl} \mathsf{isDComponentOf} & \equiv & (\textbf{\textsf{inverse}}\ \mathsf{hasDComponent}) \\ \mathsf{hasTableOfContents} & \equiv & (\textbf{\textsf{inverse}}\ \mathsf{contentsOf}) \end{array}$$

Turning to "existence" constraints, a book (but not a *copy* of a book!) depends on the existence of its specific table of contents, and conversely. Although we can specify that a book must have table of contents, as with earlier "dynamic" aspects (such as (im)mutable class membership) standard DLs are not currently equipped to express constraints stating that an attribute value cannot change.

To model the fact that each table of contents belongs exclusively to one book, we can use qualified number restrictions

$$\mathsf{TableOfContents} \sqsubseteq (\textbf{\textsf{the}}\ \mathsf{contentsOf}\ \mathsf{Book})$$

Finally, the inheritance of properties (e.g., $\mathsf{isDefective}$) across component-like attributes is modeled using constructs such as **same-as**, which relate attribute/role chains set-theoretically, in the same manner as shown with materialization:

$$\mathsf{Book} \sqsubseteq (\textbf{\textsf{same-as}}\ \mathsf{isDefective}\ (\mathsf{hasTableOfContents} \circ \mathsf{isDefective}))$$

Note however that several of these representations require quite expressive language constructs, whose combination may result in a language for which subsumption is undecidable.

10.7.4 General constraints

In many modeling exercises one will encounter general constraints that characterize valid states of the world. For example, the $\mathsf{dueDate}$ of a book must be later than the lentOn date.

Except for a few cases involving identity of attribute paths, these constraints will not be expressible in standard DLs, due to their limited expressive power. Several widely distributed systems, such as CLASSIC and LOOM, offer "escape hatches"—concept constructors that allow one to describe sets of individuals using some very powerful language, such as a programming language (CLASSIC's test-concepts) or some variant of first-order logic (LOOM's assertions). These concept definitions are usually opaque as far as concept-level reasoning is concerned, because the system cannot guarantee correctness for such an expressive formalism. However, these concepts can have an impact as far as the ABox reasoning is concerned, since the latter resembles a logical model, and therefore we can do relatively simple "evaluation" as a way of recognizing individuals. Thus, in CLASSIC, the test-concept (**test** date-after (dueDate lentOn)) would invoke the date-after function on the dueDate and lentOn attributes of an individual object, and check that the first is temporally after the second, thus classifying individuals, or detecting errors in the ABox.

More general than these procedural extensions are DL systems that are *extensible* in the sense that a "knowledge language engineer" can add new concept constructors, and extend the implementation in a principled way. For example, if we wanted to deal with dates and durations (clearly a desirable feature for libraries), we would want to be able to compare dates, add durations to dates, etc. General approaches to extending DLs have been described, among others, in [Baader and Hanschke, 1991b; Borgida, 1999; Horrocks and Sattler, 2001].

10.7.5 Views and contexts

Although the initial goal is usually to provide a single model of the UofD, it turns out to be very important to preserve the various "views" of the information seen by different stake-holders and participants. For example, a book that is in the library (and by definition, this would mean that it has no value for the lentTo role) is of interest to the staff, for example to help find it; for this, it may have a role location, which might specify some shelf or sorting area; this attribute may be attached to the MaterialInLibrary concept.

On the other hand, a view of Material called MaterialOnLoan (which *requires* a lentTo role value), would be a natural place to keep information about dueDate and nrOfRenewals—attributes that would normally appear on the relationship itself. This view is of particular interest to the borrower, but also the staff in charge of sending overdue notices.

Incidentally, the above pattern of replacing a binary relationship having attributes by two views can be applied any time one of the participants in the relationship is restricted to appear in at most one tuple (e.g., every book can be loaned to at most one borrower).

10.8 A conceptual modeling methodology

The world of object-oriented software development has produced a vast literature on methodologies (e.g., [Shlaer and Mellor, 1988]) for identifying objects, classes, methods, etc., for a particular application. Instead of considering this voluminous material here, we will recapitulate some of the issues raised above by extending the outline of a simple DL knowledge engineering methodology first presented in [Brachman *et al.*, 1991]. The reader is referred to that article for more details, including a long worked-out example.

We present the main steps of modeling, with suggestions for refinements to be accomplished in later passes; this is in order to avoid the modeler becoming overwhelmed by details:

- Identify the individuals one can encounter in the UofD. Revisit this later considering issues such as materialization and values.
- Enumerate concepts that group these values.
- Distinguish independent concepts from relationship-roles.
- Develop a taxonomy of concepts. Revisit this later considering issues such as disjointness and covering for subconcepts.
- Identify any individuals (usually enumerated values) that are of interest in all states of the world in this UofD.
- Systematically search for part-whole relationships between objects, creating roles for them. Later, make them sub-roles of the categories of roles mentioned in Section 10.7.3.
- Identify other 'properties' of objects, and then general relationships in which objects participate.
- Determine local constraints involving roles such as cardinality limits and value restrictions. Elaborate any concepts introduced as value restrictions.
- Determine more general constraints on relationships, such as those that can be modeled by subroles or **same-as**. (The latter often correspond to "inheritance" across some relationship other than IS-A, and have been mentioned in several places earlier.)
- Distinguish essential from incidental properties of concepts, as well as primitive from defined concepts.
- Consider properties of concepts such as rigidity, identifiers, etc., and use the techniques of [Guarino and Welty, 2000] to simplify and realign the taxonomy of primitive concepts.

10.9 The ABox: modeling specific states of the world

So far, we have concentrated on describing the conceptual model at the level of concepts. In some applications we may want to use our system to keep models of specific states of the world—somewhat like a database. As discussed in Chapter 2, this involves stating for each specific individual zero or more fillers for its attributes and roles, and asserting membership in zero or more concepts (primitive, but also possibly defined).

One of the challenging aspects of modeling the state of the world with DLs is remembering that unlike databases, DL systems *do not make the closed-world assumption*. Thus, in contrast with standard databases, if some relationship is not known to hold, it is not assumed to be false.

One consequence of this is that any question about the membership of an individual in a concept, or its relationship to another individual, has *three* possible answers: definitely yes, definitely no, or unknown. The positive side of this is that it allows the modeling of states with partial information: one can model that BOOK22 is an instance of Book, and hence has exactly one filler for isbnNr, yet not know what that value is. Chapter 12 shows how this feature has been exploited in developing a family of DL applications for configuring various devices.

Another consequence of the above stance is that in some cases individuals are not recognized as satisfying definitions when one might expect them to. For example, suppose we only know that hasAuthor relates BOOK22 to SHAKESPEARE, who in turn is known to be an instance of Englishman. This, by itself, is not enough to classify BOOK22 as an instance of concept (**all** hasAuthor Englishman); we must also know that there are no other possible fillers for BOOK22's hasAuthor role—i.e., that BOOK22 is an instance of (**at-most** 1 hasAuthor)—before we can *try* to answer definitively whether BOOK22 is an instance of (**all** hasAuthor Englishman). Even in this case, if the answer is not 'yes', we may get 'no' or 'maybe.'

A final consequence of not making the closed-world assumption is that there is a clear distinction between the state of the world (out there) and our (system's) *knowledge* of it. This is reflected by the terminology used above (e.g., "we must also *know* there are no other possible fillers"). As a result, in modeling a domain one may find it necessary to specify concepts that involve the state of our knowledge base, rather than the state of the world. For example, we might want to find out exactly which books in the KB are not known to have a ISBN number. The description (**and** Book (**at-most** 0 isbnNr)) will not do the job, because the second constraint would conflict with one of the the necessary conditions of Book, which is that it must have have exactly one isbnNr. What is happening here is that the **at-most** 1 constraint concerns the state of the world, while the **at-most** 0 condition involves the KB's knowledge of the world. To deal with this, we need some form of *epistemic* operator, so we can define the concept

UnknownIsbnBook ≡ (**and** Book (**at-most** 0 (**known** isbnNr)))

The general problem of adding an epistemic operator to DLs is considered in [Donini *et al.*, 1998a], but this is not available in currently implemented DLs. A "hack" would be to introduce for such roles a subrole, whose identifier indicates its epistemic nature:

$$\mathsf{knownToHaveAuthor} \sqsubseteq \mathsf{hasAuthor}$$

and then be sure to assert fillers only about the "known" variant. Unfortunately, there is no way to tell a DL that such roles automatically have the "closed-world assumption."

10.10 Conclusions

There are a wide variety of sources that discuss the application of object-oriented approaches to modeling a domain. The same principles apply to conceptual modeling in general. For this reason, we have concentrated here on some of the more subtle issues and ontological issues that arise during modeling, and the different ways in which these can be encoded in DLs. In some cases the issues examined were suggested by features of DLs themselves.

In the process, we covered most of the kinds of questions that would have to be addressed while modeling something like the library domain, and uncovered some of the strengths and also some of the weaknesses of DLs in representing this conceptual model. The latter include difficulty in representing (structured) values, constraints related to the dynamic aspects of the domain, certain forms of "inheritance" (e.g., for materialization), and meta-information. These were balanced by the multitude of features dealing with primitive and defined concepts, necessary and sufficient conditions for concept specification, and the treatment of roles as first-class citizens in subclasses and composition.

Probably the biggest problem in developing an appropriate conceptual model for a domain is that of testing it for correctness and completeness. The former is supported by the reasoning and explanation facilities provided by DLs. The latter, as usual, is much more difficult to achieve.