1

Conceptual Modeling with Description Logics

Alex Borgida

Ronald Brachman

Abstract

The purpose of the chapter is to help someone familiar with DLs to develop a conceptual model of some universe of discourse, which is to become a knowledge base that is reasoned with.

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, which combines general ideas of object-centered modeling with a look at special modeling/ontological problems, and DL-specific solutions to them.

In general, we cover the modeling of individuals, concepts, definitions, and relationships between individuals. Among the specific issues considered are concept specialization, reification and n-ary relationships, materialization, collections, components, and epistemic aspects of individual knowledge.

1.1 Background

Information modeling is concerned with the construction of computer-based symbol structures which 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 queried and updated through special-purpose languages, analogously to the way databases are accessed and updated through query and data manipulation languages. As with all models, the advantage of information models is that they abstract away irrelevant details, and allow more efficient study 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 build on conventional programming notions such as records, etc. and as a result focused on the implementational aspects of the information being captured, in contrast to the representational aspects. *Conceptual models* offer more expressive facilities for modeling applications *directly and naturally* [hammer:mcleod], and for *structuring* information bases. These models provide semantic terms for modeling an application, such as entity, activity, agent and goal, as well as means for organizing information in terms of abstraction mechanisms.

Conceptual models play an important part in a variety of areas. Some of the earliest and most significant efforts have been reviewed in [mylopoulos:IS], and we summarize them here:

- To begin with, they were natural elements of Artificial Intelligence programs, which turned out to require the representation of a great deal of human knowledge in order to act "intelligently". Semantic networks —directed graphs labeled with natural language identifiers — were among the earliest such representations. DLs are in fact 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:ER] and later semantic data models [HullKingReview] were the result of efforts in these directions.
- 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. Ross' SADT technique [Ross:sadt] and Jackson [Jackson] were among the earliest and most explicit advocates of the modeling view of requirements.
- Independently, the object-oriented software development paradigm has proposed viewing software components (classes/objects) as models of real-world entities. This was evident in the initial motivation for the Simula language — the ur-OO language, and became a cornerstone of most object-oriented techniques, leading up to the current leader, UML [uml].

One of the interesting aspects of conceptual modeling in the database context has been the identification a number of *abstraction mechanisms* that support the development of large models by abstracting details initially, and then introducing them slowly and systematically. The result is a structured information model, which is easier to build and maintain. Among the important abstractions are aggregation (thinking of objects as wholes, not just a collection of their attributes/components), classification (abstracting away the detailed differences between individuals, so one can present a class that represents the commonalities) and generalization (abstracting the commonalities of several classes into a superclass).

1.2 Elementary DL modeling

DLs subscribe to an *object-centered* approach to modeling, in which the world is seen to be populated by individual objects, which are associated with each other through (usually binary) relationships, and are grouped into classes.

In our library domain, we might encounter a particular person, Giani, 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 Giani having borrowed Book23. Binary relationships are modeled directly in DLs using *roles and attributes*: either Giani is a filler of the lentTo role for Book23, or Book23 is the filler of the hasBorrowed role for Giani. 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**

hasBorrowed **is defined as** (**inverse** *of* lentTo)

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"; thus, in the above case lentTo(Book23,Giani) reads "Book23 lentTo Giani", while lentTo(Giani,Book23) reads "Giani 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 (poorly) imagine loan as a role of books and of persons, so that neither loan(Giani,Book23) nor loan(Book23,Giani) "read" properly. *Alex question: Is it worth having the above here?*

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. Functional relationships should be declared as attributes (assuming that the DL system being used supports the role/attribute distinction). In such cases, it is important to note whether the DL system interprets attributes as total functions (i.e., always having a cardinality of one) or partial ones. In our case, lentTo is clearly meant to be a partial function, since the book may not be on loan at some times. In the rest of this chapter we assume that attributes are partial functions, and the concept constructor **the** will be used as an abbreviation, so that (**the** p *value is in* C) is equivalent to the conjunction of (**all** p *values are in* C), (p *has* **at-most** 1 *values*) and (p *has* **at-least** 1 *values*). If the attribute r is allowed to have no filler, we will use (**all** r *values are in* C) , and redundantly say (r *has* **at-most** 1 *values*) , to emphasize the absence of the lower bound.

Individuals are grouped into classes, which are modeled by concepts in DLs. For example, BOOK might be a concept in our model. Classes usually abstract out common properties of their instances, which can be expressed as necessary conditions that must hold of all instances of the concept:

```
/* Books are materials, whose callNr is an integer */
   BOOK  is subsumed by   (and
                           MATERIAL
                           (the callNr value is in INTEGER)
                           ... )
```

As mentioned in earlier chapters, one of the fundamental properties of DLs is the 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, DLs support the distinction between the notion of "borrower" as someone who can borrow material (an approved customer of the library)

```
/* Borrower, previously declared as a primitive concept, has additional properties,
such as hasBorrowed, which will hold, when necessary, the material then borrowed */
   PONTENTIAL-BORROWER  is subsumed by    (all hasBorrowed values are in BOOK)
```

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

```
/* Borrower is defined as someone who has things borrowed */
   ACTUAL-BORROWER  is defined as   (and
                                     (all hasBorrowed values are in BOOK)
                                     (hasBorrowed has at-least 1 values) )
```

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

1.3 Individuals in the world.

Some individuals are quite concrete, like a particular person, Giani, or particular copy of a book. Some are abstract, like the book "Hamlet", as opposed to any particular physical printing of it. Both these kinds of individuals have an intrinsic identity, so that if one sees on the bookshelf two brand new copies of a book, which may not be distinguishable by any property, one can still say that they are different books. In order to refer to such individuals, it would be easiest if each had a unique name but unfortunately the real world is rarely as neat as this. In this paper, as in object-oriented systems, we will be assigning arbitrary identifiers to individuals (e.g. Giani, Book23) as we go along, as a way to refer to them.

1.3.1 Values vs. objects.

It is important to distinguish *objects*, which, as we said, have intrinsic and immutable identity, and *values*, which are mathematical abstractions, such as integers, strings, lists, etc., 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 triangles with sides of length 25,12 and 20; or dates such as 1925/12/20.

Although values are "eternal", objects have an associated period of existence. Time is therefore an intrinsic part of every model, though frequently it is omitted, with the understanding that the model reflects only the state of the world at the present moment.

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 counting and identity, among others, will be off. DLs such as Classic support values from the underlying programming language (so-called "host values"). However, equally desirable would be mathematical types such as sets, bags, sequences and tuples, as supported by modern programming languages and certain semantic data models.

1.3.2 Individuals vs. references to them.

It is important to distinguish an individual from various references to it: Giani vs. "the person whose first name is the 5 letter string `"Giani"`" 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 want inter-related objects to remain related, even if names are changed. Thus "Giani borrowing Book25" is different from "card number 32245 borrowing Book25", because if Giani gets a new card (because he lost his old one, say), then the relationship between Giani 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 total (it always exists) and is unchanging. Kent [Kent79] 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 [neuman:book] reports that the same (famous) US social security number 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 (Comiti3). If Giani happens to be the current director of the NBU library, then asserting (Giani,Comiti3) **is an instance of** headOf is improper because, among others, if Giani steps down as director, according to the above model he would still be committee chair. One needs the ability to use undereferenced expressions as arguments of relationships, along the lines of the predical logic expression $headOf(directorOf(NBUlibrary), Comiti3)$.

In DLs, intentional 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 month of the birthDate of the lentTo".) Asuming that we use the notation NBUlibrary.director to refer to the filler of the director role for NBUlibrary individual, the above relationship is actually stated as "NBUlibrary.director is identical to Comiti3.head". The concept constructor **same-as** is used to express exactly such relationships, so the above situation might (naively) be modeled through the concept (*the value of* director *is* **same-as** *the value of* head) . The problem is that we need a single individual of which to assert this property, yet it is libraries that have directors and committees that have heads. In such situations one must introduce a new attribute that relates the two individuals, NBUlibrary and Comiti3. In this case, the natural relationship is the attribute hasBookSelectionCommittee. Therefore the final way of modeling this situation is

/* NBULibrary has book selection committee Comiti3 */
(NBUlibrary,Comiti3) **is an instance of** hasBookSelectionCommittee

/* The value of NBUlibrary.director equals NBUlibrary.hasBookSelectionCommittee.head */
NBUlibrary **is an instance of**
(*the value of* director *is* **same-as** *the value of* (hasBookSelectionCommittee $\circ$ head))

1.4 Classes

For the university library, some obvious classes of individuals that come to mind, in addition to library, are people, the material that the can be loaned by the library, the staff, possibly even the equipment and physical plant of the library. Also important are dates, possibly fines and library cards/ids. These classes are normally modeled using atomic/primitive concepts in DLs.

An important observation is that the same individual may be an instance of multiple classes, without one being necessarily a subclass of another: some book might be both hard-cover and a science book. Unfortunately, in many modeling languages based on object-oriented implementations, one is forced to create a special subclass for this notion, in order to guarantee a unique "minimal" class for every individual. This is not a modeling principle — it is an implementation obstacle.

An interesting distinction, which becomes relevant when changes in the model are admitted, is between classes that represent unchangeable, inherent properties of objects, such as PERSON and BOOK, and classes that represent more transient properties, such as BORROWER or BOOK-ON-LOAN. The name of the last concept in fact gives away the basis of this distinction: the second category of classes arise as a result of some relationship (or activity that establishes a relationship) between objects: in this case, the library lending a book to a borrower. In the data modeling literature (e.g., [AlbanoBGO93]) such classes are called "roles", but to avoid confusion with DL roles, we will call them RelationshipRoles. The modeling of these will be considered in further detail when discussing relationships.

1.5 Necessary vs. Sufficient Conditions on Class Membership

As we have explained earlier, dealing with 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 class membership.

For example, BOOK-ON-LOAN might naturally be defined as

```
/* A book is on loan if it borrowed by someone */
   BOOK-ON-LOAN is defined as  BOOK  and  (lentTo has at-least 1 values) )
```

Suppose that we also want to require that the book be hard-cover, if it is to be loaned out. There might appear to be two options for modeling this:

```
/* Option 1 – being hardcove is  part of the definition of BOOK-ON-LOAN */
   BOOK-ON-LOAN is defined as   (and BOOK
                                 (lentTo has at-least 1 values)
                                 (binding includes value 'hardcover)
                                )

/* Option 2 –  being hardcover is an additional necessary condition */
   BOOK-ON-LOAN is defined as  BOOK  and  (lentTo has at-least 1 values) )
   BOOK-ON-LOAN is subsumed by    (binding includes value '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 others, it 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 need to also know it is hardcover. Hence the second modeling option is the right one, since, in fact, one can deduce that a book on loan is hardcover, if this was not known ahead of time.

An even subtler 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

BOOK **and** (dueDate *has* **at-least** 1 *values*)) **is subsumed by** BOOK-ON-LOAN

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

BOOK-ON-LOAN **is subsumed by** (dueDate *has* **at-least** 1 *values*)

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

1.6 Subclasses

For many of the above classes, there are specialized sub-classes, representing subsets of individuals that are also of interest. For example, the MATERIAL could be BOOK, JOURNAL, VIDEOTAPE, CD-ROM, MICROFICHE, etc. In turn, BOOK may have subclasses MONOGRAPH, EDITED-COLLECTION, PROCEEDING, etc. [1] And BORROWERs may be INSTITUTIONs or INDIVIDUAL-BORROWERs, with the later 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.

1.6.1 Disjointness of subclasses.

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 class to the necessary properties of the other class:

BOOK **is subsumed by** **not** JOURNAL
JOURNAL **is subsumed by** **not** JOURNAL

Often, entire collections of subclasses are disjoint[2]. For this purpose, it is useful for the DL to 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

[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, DATE, etc.

PRINT **is subsumed by** (**disjoint primitive** *subconcept of* MATERIAL *with discriminant* medium)
VIDEO **is subsumed by** (**disjoint primitive** *subconcept of* MATERIAL *with discriminant* medium)
AUDIO **is subsumed by** (**disjoint primitive** *subconcept of* MATERIAL *with discriminant* medium)

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

BOOK **is subsumed by** (**disjoint primitive** *subconcept of* MATERIAL *with discriminant* format)
JOURNAL **is subsumed by** (**disjoint primitive** *subconcept of* MATERIAL *with discriminant* format)
PROCEEDINGS **is subsumed by** (**disjoint primitive** *subconcept of* MATERIAL *with discriminant* forr

As in the above example, it is important to allow for multiple groups of subclasses for the same class. MATERIALs could also be subdivided into CIRCULATING and NON-CIRCULATING, with former being SHORT-TERM and LONG-TERM.

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.

1.6.2 Covering by subclasses.

In addition to disjointness, it is natural to consider whether some set of subclasses fully covers the superclass. In the above example, we may want to state that CIRCULATING material must be either short term or long term. For DLs that support concept disjunction, this is easy:

CIRCULATING **is subsumed by** (**or** SHORT-TERM LONG-TERM)

Since SHORT-TERM, in turn, has CIRCULATING as a superclass, the possibility arises of modeling CIRCULATING as a definition

CIRCULATING **is defined as** (**or** SHORT-TERM LONG-TERM)

Semantically, the result will be the same (in terms of deductions that will be performed). If one was doing top-down hierarchy design, it might be less natural to revise the initially primitive CIRCULATING, into a defined concept. But this may be a matter of taste.

In Classic, and other languages that do not support disjunction and general negation, there is no real way to specify that some class is covered by the union of a certain subset of subclasses.

1.6.3 Defined vs. primitive subclasses.

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, NON-CIRCULATING should be *defined*:

CIRCULATING **is subsumed by** MATERIAL

NON-CIRCULATING **is defined as** MATERIAL **and** (**not** CIRCULATING)

In this case, the logic of the system can then deduce both the disjointness of CIRCULATING and NON-CIRCULATING, and the fact that MATERIAL is the union of CIRCULATING and NON-CIRCULATING, without having stated anything explicitly about either. This shows clearly the power of a reasoning system that is capable of supporting definitions.

In general, any primitive subclass C of a class B can be turned into a defined one by simply adding some attribute isC? that takes on a boolean value, and then defining

C **is defined as** B **and** (isC? **includes** *value* true)

This trick does not appear to have much to recommend it. However, in DLs that allow values to appear in concept descriptions, this idea can be generalized to simulate partitions. First, consider what happens if we want to model persons, and their division into males and females. As an alternative to defining two disjoint and covering subclasses, MAN and WOMAN, of class PERSON, we could introduce the attribute gender for PERSON, with possible value 'm (for masculine) and 'f (for feminine):

PERSON **is subsumed by** ... (**all** gender *values are in* (**enumerated** *set:* 'm , 'f))

This would allow MAN and WOMAN to be actually defined, in terms of the corresponding values of gender:

MAN **is defined as** PERSON **and** (gender **includes** *value* 'm))

WOMAN **is defined as** PERSON **and** (gender **includes** *value* 'f))

Once again. the partitioning of PERSON into MAN and WOMAN is automatically deducible. In general, one can add some attribute corresponding to a discriminant, such as format, with an enumerated set of possible values corresponding to the subclasses (e.g., 'monograph, 'journal, 'editedCollection, etc.) and then define the corresponding subclasses:

BOOK **is subsumed by** (**all** format *values are in* (**one-of** 'monograph,'journal,'editedCollection))

MONOGRAPH **is defined as** BOOK **and** (format **includes** *value* 'monograph))

JOURNAL **is defined as** BOOK **and** (format **includes** *value* 'journal))

EDITED-COLLECTION **is defined as** BOOK **and** (format **includes** *value* 'editedCollection))

If the attribute for the discriminant (e.g., format) does not seem to be independently motivated and useful in the application, this encoding can be avoided. On the other hand, in cases such as gender, which seems to be a natural descriptor of persons, one might start with the attribute originally, and introduce the subclasses *only if useful in more than one case*; otherwise, one can use nested complex descriptions in concepts. This is one of the distinguishing characteristics of DLs, allowing us to avoid introducing unnecessary names. For example, if MAN and WOMAN would only be used in the description of PERSON as follows:

```
PERSON  is subsumed by  ...
            (all mother values are in WOMAN)
            (all father values are in MAN)
```

then we could say instead

```
PERSON  is subsumed by  ...
            (all mother values are in (gender includes value 'f) )
            (all father values are in (gender includes value 'm) )
```

In a DL such as Classic, which does not allow recursive concept specifications, the second alternative is in fact the only one available (even though it leaves implicit the fact that mother and father need to be persons).

1.6.4 Dynamics of (sub)class membership.

Especially in the database context, another conceptually important distinction concerning classes is whether an object's membership in it can change with time. Thus, once a book is acquired, it remains in the library as a book till it is disposed of in some way or other. (In our view, if it is photographed onto a microfiche, the library no longer has the book, it has a microfiche.) On the other hand, the classification of any material as loanable on short-term (i.e., member of concept SHORT-TERM) can obviously change with time. Standard DLs have not developed special modeling tools for this case, because the issue is inherently dynamic. DLs extended with the notion of time, such as [artale:franconi:time], are of course well suited to express such issues.

One could try some ad-hoc technique such as defining two top-level disjoint classes, MUTABLE and IMMUTABLE, and require that every concept be made a subclass of one or the other. However, in this case we would run into trouble because we would want MATERIAL to be immutable, but the SHORT-TERM subclass be mutable — a contradiction in face of inheritance. By keeping only the special MUTABLE class, and only representing mutability explicitly, one has a reasonable approximation, since if membership in some class is mutable, then it is almost surely mutable in its subclasses. The purists will note of course that there is no "semantics" to this solution — it is simply a syntactic marking and a convnetion on its use.

1.7 Modeling relationships

As mentioned earlier, binary relationships are modeled in DLs using roles/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 that role; domain constraints state the kinds of objects that can be related via the 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.

```
BOOK  is subsumed by   ...
              (the title value is in STRING)
              (the callNr value is in MATERIAL-IDENTIFIER)
              (all author values are in PERSON)
```

As mentioned in Section 1.2, we can use the attribute lentTo to model when someone borrows a book:

```
BOOK  is subsumed by   ...
              (all lentTo values are in BORROWER)
```

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

```
LIBRARY  is subsumed by   ...
              (all hasOnLoan values are in MATERIAL)
              (all hasAvailable values are in MATERIAL)
              (all hasMissing values are in MATERIAL)
```

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, it would be better to model the situation using appropriate subclasses of MATERIAL, 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** (which, in general, lead to undecidability).

1.7.1 Reified relationships.

It is sometimes useful to attach attributes to qualify relationships. 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 definition:

```
LOAN  is subsumed by
          (the lentTo value is in BORROWER)
          (the onLoan value is in MATERIAL)
          (the lentOn value is in DATE)
          (the dueOn value is in DATE)
          (the NrOfRenewals value is in (max 3))
```

Note the use integer ranges to restrict the number of times a loan can be renewed.

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 constraint that there can't be several objects recording $R(a, b, ...)$ (just as with dates, in Section 1.3.1). Without that, in the above model of LOAN, it is not possible to distinguish between the binary relationship $LOAN(borrower, material)$, recording who currently has books out on loan, and the ternary relationship $LOAN(borrower, material, lentOn)$, which would record a history of the loans. For this reason, it would be important to distinguish a reified relationship and its "defining" tuple-attributes. Currenly, only the $\mathcal{DLR}$ description logic [degiacomo:dlr] can express n-tuples.

1.7.2 Role hierarchies.

When thinking about relationships between roles, users of DLs are encouraged to consider not just the inverse relationship, but also specialization. This is all the more reasonable, since once we reify a relationship, we are allowed to create subclasses of it.

In the library domain, examples of such role specialization tend to be contrived, but such natural examples as daughterOf being a sub-role of childOf, which in turn is a sub-role of relativeOf, indicate that this is a naturally useful construct.

In most implemented DL systems, one is limited to primitive role hierarchies, although in general, the theory of DLs supports role definitions and subsumption reasoning on roles on an eqaul footing with reasoning about concepts (reference chapter???).

1.7.3 Relationship Roles.

A subtle, but important distinction can be drawn between objects that *may* participate in a relationship (the domain restrictions on the participant attributes) 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": LENT-OBJECT 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 1.2.

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

```
LENT-OBJECT  is defined as   (borrowedBy has at-least 1 values)
BORROWER  is defined as   ( (inverse of borrowedBy) has at-least 1 values)
```

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

```
LENT-OBJECT  is defined as   ( (inverse of onLoan) has at-least 1 values)
BORROWER  is defined as   ( (inverse of lentTo) has at-least 1 values)
```

1.7.4 Reifying classes.

Classes can themselves be treated as (meta)objects, with their own relationships. These are frequently useful for capturing aggregate information, which would not normally be associated with every individual in the class. For example, if we were dealing with only one library, and did not want to have a LIBRARY class, the roles onLoan, missing, and available could be associated as meta-roles of the class MATERIAL.

In general, the great difficulty is to have assertions about meta-class objects that have an effect on the instances of the class. In part because of the complex semantics of such "reflection", DLs do not currently support such connections.

1.8 Modeling ontological aspects of relationships.

The material in this section deals with some special 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 epistomological aspects (constructs relating to the structure of objects), which are captured by notions such as InstanceOf and IsA. In particular, observe that the special meaning of the later two are incorporated in a fundamental way into DL semantics and reasoning. The relationship kinds to be discussed below occur relatively frequently, and pose difficulties to the uninitiate, but do not have "built-in" solutions in DLs.

1.8.1 Groupings.

In some situations objects are grouped together into collections, which take on an identity of their own[1]. For example, although "The Adventures of Huckleberry Finn", "The Adventures of Tom Sawyer", etc., are books by Mark Twain, they also appear in "The Unabridged Mark Twain", edit by Lawrence Teacher. And some of these pieces might also appear in an "Anthology of American Humor". There are even cases where pieces, such as G.B. Shaw's play "Caesar and Cleopatra", were first published by the author in a trilogy bound in a single book, with a totally different title ("Three plays for puritans"). Note that this trilogy is a distinct entity (it has a separate 30 page forward by Shaw), so in general it is inappropriate to model collections simply as sets. Moreover, it would also be inappropriate to make the collection be a concept, with its members as instances, since collections are created "dynamically" in the world, and we would be constantly be creating new concepts in the domain model[2].

Therefore, the appropriate model for this situation is a concept such as COLLECTED-WORKS for the grouping, connected to its members by a special relationship — say hasMember.

```
COLLECTED-WORKS is subsumed by ...
        (all hasMember values are in MATERIAL)
        (the title value is in STRING)
        (the publishedBy value is in PUBLISHER)
```

As the above examples suggest, there is no general relationship between the roles of the collection and those of its members. Also, note that a collection could have collections as members (e.g., there could be a series of books, such as Penguin Classics, in which the "Three Plays for Puritans" might appear).

In order to capture properly the distinctive semantics of the relationship between a collection and its members, we suggest two modeling conventions: (i) use the hasMember role to capture the relationship; (ii) all conceptual models should have an immediate subconcept of **thing** called COLLECTION-OF-THINGS. The above example would then be extended so that COLLECTED-WORKS is a subconcept of COLLECTION-OF-THINGS.

1.8.2 Materialization.

Materialization [Pirotte94] describes the relationship illustrated by the distinction between the abstract notion of the play "Hamlet" and various editions of Hamlet published by different publishers; and between these and the actual individual books on shelves; (or similarly, productions of the play, various individual performances,

[1] This was first introduced in SDM [hammer:mcleod], using the paradigmatic example of a convoy of ships.
[2] Convoys of ships are example collections that can not only be created but also dissolved dynamically. This would necessitate removing concepts from the domain model too – a clearly undesirable situation.

and videotapes). The difficulty in modeling these notions is due to the confusion introduced by natural language nouns, since all of the above are normally called "the play Hamlet".

To model such situations in DLs, one creates a concept for the abstract notion (e.g., the play Hamlet by Shakespeare), and then relates the more concrete materialization of it (e.g., the 1958 Penguin paperback edition) using an attribute (e.g., editionOf). To be systematic, one could try using the same identifier, materializationOf, everywhere; however, the resulting model is not readable as natural English. Instead, we suggest having a built-in role materializationOf, with inverse materializedBy, and then declaring attributes such as editionOf to be specializations of it. Thus, the relationship between books and their editions would be modeled as follows:

```
/* Books have information about authors, etc. */
   BOOK  is subsumed by   ...
                    (all hasAuthor values are in PERSON)
                    (the hasTitle value is in STRING)

 /* editionfOf is a kind of materialization relationship */
   editionOf  is subsumed by   materializationOf

/* editions of books are related to the book, but have their own roles too */
   BOOK-EDITION  is subsumed by   ...
                    (the editionOf value is in BOOK)
                    (the publishedBy value is in PUBLISHING-COMPANY)
                    (the isbnNr value is in ISBN-NUMBERS)
                    (the format value is in (enumerated set: 'printed,'audio) )
```

Furthermore, the relationship between book copies and editions is modeled as

```
/* Book copies will be related to book editions via copyOf */
   copyOf  is subsumed by   materializationOf

/* Book copies in turn have their own roles */
   BOOK-COPY  is subsumed by   ...
                    (the copyOf value is in BOOK-EDITION)
                    (the callNr value is in CALL-NUMBERS)
                    (the atBranch value is in LIBRARY-BRANCH)
```

Normally, the properties of the materialization are related to those of the more abstract concept: e.g., the book edition is likely to have the same author and title as the book. Pirotte et al [pirotte:vldb:94] distinguish 3 kinds of "inheritance" from the object being materialized, including cases where (i) values of some enumerated role value might become fixed for a materialized version; .e.g, a book copy could have a format with exactly one value, such as 'audio; (ii) values of some attributes become role identifiers for the materialized object; e.g., book copies might have attribute print-format being either 'paperback or 'hard-cover; or audio-format one of 'tape, 'cd, etc.

In DLs, some of these relationships can be expressed by adding appropriate attributes on the materialized concept and relating them to the attributes of the general concept:

```
BOOK  is subsumed by  ...
                   (the value of  hasTitle  is  same-as  the value of  (editionOf ∘ hasTitle))
```

1.8.3 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 [artale:partwhole] being probably the most comprehensive recent survey; it considers, among others, a variety of DL solutions to the problem. We summarize here some of the main observations.

Cognitive scientists have distinguished a variety of part/whole relationships, whose mixture has caused apparent paradoxes; these include component/integral-object, portion/mass, stuff/object, feature/activity, place/area and member/collection[1].

To capture the special ontology of the part-whole relationship, one should be able to

- explicitly create complex wholes from parts;
- distinguish between parts and other roles of a whole;
- treat partOf as a transitive relationship;
- refer to parts by generic names
- express relationships between wholes and parts concerning their existence and properties. *Existence*: the whole may rigidly depend on particular individuals (e.g., if the part is irreplaceable); or depend generically on a class of parts. 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 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).

One thorough attempt at dealing with these topics description logics is that of Sattler [sattler:partof], who exploits various role-forming operators including role hierarchies, inverse, and transitive closure to capture the semantics of aggregation.

Special roles are introduced for the different kinds of part-whole relationships mentioned above: has-d-component, has-d-member, has-d-segment, has-d-quantity, has-d-stuff, has-d-ingredient, where "d" stands for "direct". One then defines more complex relationships from these primitives:

[1] This has been discussed in a preceding subsection

has-component **is defined as** (**transitive_closure**
(has-d-component $\mathbf{or}_{role}$ (has-d-member $\circ$ has-d-component)))

hasPart **is defined as** has-component $\mathbf{or}_{role}$ has-member $\mathbf{or}_{role}$ ...
indicating that members of collections of components are also components, and that hasPart is the union of the various sub-kinds of relationships .

For this chapter, let us consider only 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.

As illustrated in the case of materialization, one idea is to declare attributes and roles that represent components (e.g., indexOf, chapterOf) as specializations of has-d-component. This allows us to distinguish such component roles from other roles, like lentTo and publisher.

Obviously, the inverses of such roles provide access from a part to its containing whole:
is-d-component-of **is defined as** (**inverse** *of* has-d-Component)
hasIndex **is defined as** (**inverse** *of* indexOf)

Turning to "existence" constaints, a book (but not a *copy* of a book!) depends rigidly on the existence of its specific index, and conversely. As with earlier "dynamic" aspects, such as (im)mutable class membership, standard DLs are not currently equiped to express such constraints.

To model the fact that an index belongs exclusively to one book, we can use qualified number restrictions

INDEX **is subsumed by** ...
(indexOf *has* **exactly** 1 *value(s) of* **kind** BOOK)

Finally, the inheritance of properties (e.g., isDefective) accross 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:

BOOK **is subsumed by** ...
(*the value of* isDefective *is* **same-as** *the value of* (hasIndex $\circ$ 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.

1.9 Views/contexts.

Although the initial goal is 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 MATERIAL-IN-LIBRARY concept.

On the other hand, a view of MATERIAL called MATERIAL-ON-LOAN, (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).

1.10 The Abox: modeling specific states of the world

So far, we have concentrated on describing the conceptual model at the level of generic concepts. In some applications we may want to use our system to keep models of a specific state of the world — somewhat like a database. As discussed in Chapter ???, 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 most 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.* In other words, unless explicitly stated or infered, there can be additional fillers for a role.

The purpose of this is to allow incomplete information: one can know that book22 is an instance of BOOK, and hence has exactly one filler for isbnNr, yet not know what that value is. In other words, there is an important distinction to be made between the state of the world and our (system's) *knowledge* of it.

One consequene of the above stance, is that in some cases individuals are not recognized as satisfying descriptions/definitions when one might expect. For example, suppose we know that Shakespear is an hasAuthor filler for book22, and he is known to be an instance of ENGLISHMAN. This, by itself, is not enough to classify book22 as an instance of concept **(all** hasAuthor *values are in* ENGLISHMAN**)** ; if we also know that there are no other authors possible — i.e., that book22 is an instance of **(**hasAuthor *has* **at-most** 1 *values***)** , then the classification takes place.

We mention here two potential puzzling phenomena in DLs related to role fillers.

First, the formal meaning of universal quantification (e.g., **(all** hasAuthor *values are in* ENGLISHMAN**)**) is that it holds of any individual that is known to have no fillers for a role. This means that a book without an author — one that has the desription (hasAuthor *has* **at-most** 0 *values*) asserted of it — will be recognized as an instance of (**all** hasAuthor *values are in* ENGLISHMAN) .

Second, unlike in typed databases and programming languages, DL roles and attributes are by default applicable to any individual unless expressly prohibited (using a restriction such as (p *has* **at-most** 0 *values*) or by providing a restricted "domain" for the role). This is in fact quite handy for modeling so-called semi-structured data [semiStrcutured?], but can be disconcerting to a user, if errors (e.g., when an author is given a value for the attribute hasTitle) are not dected by the system because the conceptual model was not made totally explicit.

A final consequence of not making the closed-world assumption, is that in practical applications one finds a variety of situations where one would want to define concepts which 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 do not have an ISBN number. The description BOOK **and** (isbnNR *has* **at-most** 0 *values*) will of course not do the job: every book must have an ISBN number according to the necessary conditions asserted about books, so the above description is incoherent. What is needed in this case is some form of *epistemic* operator, so we can define the concept BOOK-WITHOUT-KNOWN-ISBN **is defined as** ((**known** isbnNr) *has* **at-most** 0 *values*)

Such constructors have been considered in [epistemicDL], but are rarely available in current DLs. A "hack" would be to introduce for such roles a variant subrole, whose identifier indicates its epistemic nature:

knownToHaveAuthor **is subsumed by** hasAuthor

and then be sure to assert fillers only about the "known" variant.

1.11 What if you cannot express it in a DL?

Clearly there are many kinds of constraints that cannot be expressed in even the most expressive general DLs studied so far.

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, by necessity, opaque as far as concept-level reasoning is concerned. However, they can have an impact as far as the Abox is concerned, since the latter resembles a logical model, and therefore we can do relatively simple "evaluation" as a way of recognizing individuals.

There are also a variety of extensions to DLs for specific purposes, dealing with subjects such as time, "key/uniqueness" constraints, parts-components, plans, etc.

The pratical difficulty is that these extensions are usually not integrated into actual running systems.

More hopeful are systems that are *extensible* in the sense that one 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. Two general approaches for extending DLs in a general way have been described in Section ????.

Alex question: Muchof what I say here dependes on what has been written in earlier chapters on extensibility. Probably the best will be to have an example with dude dates of books.

1.12 Conclusions

By now, there are a wide variety of books and articles which illustrate the application of object-oriented approaches to modeling some UofD [?refs?]. For this reason, we have concentrated more on the issues that arise when encoding the relevant distinctions in DLs.

Alex question: A problem throughout has been what language contructs to assume available.