KRF

Knowledge Representation Framework Project
Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Description Logic and Defeasible Inheritance

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).

The present report, PM-krf-018, can persistently be accessed as follows:

Project Memo URL:	http://www.ida.liu.se/ext/caisor/pm-archive/krf/018/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2010/016/
Date of manuscript:	2010-11-01

Related information can also be obtained through the following www sites:

KRFwebsite:	http://www.ida.liu.se/ext/krf/
AIP naming scheme:	http://aip.name/info/
The author:	http://www.ida.liu.se/~erisa/

Chapter 1

Reasoning about Classes

Almost all applications of knowledge-based systems involve both *objects* and *classes,* where each object can be a member of one or more classes. For example, a particular automobile may be a member of the class of vehicles that are owned by a particular person, the class of automobiles made by a particular manufacturer, the class of automobiles having a particular color, the class of automobiles not having passed obligatory annual testing, and so forth.

Classes like these may be defined and used *extensionally* or *intensionally.* In the extensional case the class is considered to be simply the set of all its members. If one member is added then it is another set, and therefore another class. In the intensional case, on the other hand, the class is considered to represent its definition, rather than its set of members. The class of vehicles owned by a particular person has different sets of members at different points in time, but from the intensional point of view it is considered as being the same class anyway.

In spite of this distinction, many of the properties of, and operations on classes are essentially the same regardless of whether the extensional or the intensional perspective applies. The conventional subset relation $\subseteq$ on sets is only used when the extensional view is intended. A similar symbol, $\sqsubseteq$ is used when one does not care, so that both views may apply. Therefore, statements that apply for $\sqsubseteq$ also apply for $\subseteq$, but not vice versa. For example the rule

$$A \subseteq B \wedge B \subseteq A \rightarrow A = B$$

does not generalize to $\sqsubseteq$.

The $\sqsubseteq$ relation is referred to as *subsumption* and $A \sqsubseteq B$ is pronounced *A is subsumed by B* (notice the order!) or *B subsumes A*. The union and intersection operators $\sqcup$ and $\sqcap$ are defined similarly as counterparts of $\cup$ and $\cap$. On the other hand, the membership relation ϵ is used regardless of whether the second argument is a set or an intensional class.

The need for an intensional view of classes arises for the same reasons as for the use of features. An intensional class may have different sets of members at different times, and in fact it may be considered simply as a set-valued feature. Just like features they are also used for expressing modal

statements, such as "Lars does not know which of these cars are owned by Karin."

Most aspects of ordinary, elementary set theory is used for knowledge representation as well. However, there is one use of classes that is important for knowledge representation but that is not addressed in set theory, namely, *defeasible subsumption* which means *subsumption with exceptions.* For example, one will say that the class of mammals is defeasibly subsumed by the class of land animals, since most mammals are land animals but there are exceptions, in particular, whales. This will be the topic of one chapter in this lecture note.

As an application uses both objects and classes, it is customary that there are a number of relations between the objects, and that these relations have counterparts on the class level. For example, if both persons and countries are considered as "objects" and there is relation `is-citizen-of` between persons and countries, one may consider a class `EU-countries` of the countries that are members of the European Union, and the set of persons that are citizens of some country in that set. Such a set may be expressed as, in our notation,

```
(those person that is-citizen-of some EU-countries)
```

where the operator `those` has three proper arguments, namely `person,` `is-citizen-of,` and `EU-countries` in this example, whereas the word `that` is only for readability, and the word `some` distinguishes between a few variants of the `those` operator.

Operators such as these are in principle redundant, since all that can be expressed using them can also be expressed using predicate logic, the membership relation between objects and classes, and the relations between objects. However, class-level operators often allow more concise expression of known facts, and they also make it possible to specify specific categories of statements and of computation that can be performed efficiently. Notations of this kind is also an important topic of the present note.

Chapter 2

Description Logic

Description Logic is an important branch of Knowledge Representation, and it is widely used as a basis for the OWL representation format in the semantic web development. It has a well established publication notation and a computer-oriented counterpart in the semantic web context. The present note will not use either of these, however, since the publication notation is awful and the computer-oriented counterpart is not easily readable. Instead we shall use a smooth extension of the CEL notation, and merely mention the standard publication notation and terminology towards the end of the chapter.

2.1 Basic Description Logic - $\mathcal{ALC}$

Description Logic is applicable in situations where the following constructs are provided:

- A number of *objects*
- A number of *classes* having objects as members
- A number of *binary relations* between objects
- A membership relation ϵ between objects and classes

Each binary relation is represented by a unique symbol – the relation symbol – and a set of pairs of objects specifying those argument combinations where the relation holds. This set of pairs may be called the *contents* of the relation. The information about the contents for each of the relation symbols plus the membership relation constitute the *fact base* of the application.

If this is given, the description logic allows one to construct *class expressions* and to make *subsumption statements* involving class expressions. Subsumption statements are always of the form $A \sqsubseteq B$ where A and B are class expressions.

In basic CEL notation the subsumption statement is of course written as $[\sqsubseteq\ A\ B]$ but we allow to omit the surrounding square brackets and to use infix notation, for convenience of reading.

There is a variety of ways of writing class expressions, providing different expressivity, and in order to be able to refer to these, each variety has a name consisting of combination of capital letters in "calligraphic" font. The basic variety of description logic is characterized as $\mathcal{ALC}$. It allows the following operators for forming class expressions, in CEL notation:

- $A \sqcup B$ for the class that is defined has having the members of A and the members of B as members
- $A \sqcap B$ for the class that is defined as having as members those objects that are members of both A and B
- `(comp` A`)` for the complement class of A, that is, the class defined as having as members all objects that are not members of A
- `(those-that` r `some` B`)` , for the class having as members all objects x such that `[`r x y`]` holds for some y that is a member of B
- `(those-that` r `all` B`)` , for the class having as members all objects x such that `[`r x y`]` holds for all y that are members of B

In addition there are symbols $\bot$ and $\top$ for the empty class and the class of all objects. [1] The basic notation for the first two expressions is of course `(`$\sqcup$ A B`)` and `(`$\sqcap$ A B`)` respectively.

Rather than using the operator `those-that` it is sometimes more convenient to use the operator `those` as in the following example

```
(those person that is-citizen-of some EU-country)
```

This operator is defined in terms of the more basic operator `those-that` since

`(those` A `that` r `some` B`) =` $A \sqcap$ `(those-that` r `some` B`)`

and similarly for `all` instead of `some`. However, if more than one relation is involved then the basic operator may be more convenient, as in

```
person ⊓ (those-that is-citizen-of some EU-country) ⊓
  (those-that married-to some
     (those person that is-citizen-of some asian-country))
```

The operator name `those-that` is a bit lengthy and will therefore sometimes be abbreviated as `tho`.

2.2 Fact Base, Terminology Base, and Inference

A knowledgebase for description logic is customarily organized in two parts: a *fact base* consisting of the contents of the participating relations, including the membership relation, and a *terminology base* consisting of positive

[1] It is in fact a bit counterintuitive that Description Logic admits a single symbol for the empty class, while at the same time insisting that its "concepts" or "classes" are not merely sets, but also have an intensional aspect. There ought to be a large number of concepts, or classes, whose member sets are empty.

literals for $\sqsubseteq$ [2] but allowing composite class expressions in both their arguments, as well as positive and negative literals for the equality relation. The following are the major operations that one may wish to perform in such a knowledgebase:

- *Instance checking:* Determine whether a formula that is of the form `[`$x\ \epsilon\ C$`]` can be inferred from the knowledgebase
- *Relation checking:* Determine whether a formula of the form `[`$r\ x\ y$`]` can be inferred from the knowledgebase
- *Subsumption checking:* Determine whether a formula that is of the form `[`$\sqsubseteq\ A\ B$`]` can be inferred from the knowledgebase
- *Checking consistency of terminology base:* Determine whether the contents of the terminology base implies inconsistency.

In all these cases it is assumed that variable-free expressions are provided for x, C, etc. in the respective queries.

It is easy to implement these operations for small to medium sized knowledgebases, but some applications require performing them on extremely large knowledgebases and then the problem is nontrivial. There has been a lot of research on the problem of how to perform it efficiently, and on criteria for when it can and when it can not be done with acceptable computational complexity.

2.3 Extensions to Basic Description Logic

A number of extensions have been defined for description logic without changing its basic semantics in terms of objects, classes, a membership relation, and object-level relations. This includes the following additional ways of forming class exprssions.

- An operator `rev` on relations so that for every relation r, `(rev` r`)` is the same relation but with the arguments in the reverse order.
- An extension to the `those-that` operator whereby one can write
  `[those-that` r `at least` n B`]`
  designating the set of those x for which there exist at least n different y such that `[`$r\ x\ y$`]` .
- A similar extension using "at most" instead of "at least."
- The possibility to specify some relations as being transitive.

The first extension is characterized by the code letter $\mathcal{I}$; the second and third one by the code letter $\mathcal{Q}$ in addition to the base combination $\mathcal{ALC}$. The extension of $\mathcal{ALC}$ to allow transitive relations is characterized as $\mathcal{S}$.

In addition, there are also proposed extensions that assume a generalized semantics, for example by introducing actions and changes in time, or by introducing defeasible inheritance besides the strict inheritance represented by $\sqsubseteq$.

[2] That is, using the $\sqsubseteq$ predicate without negation on it.

2.4 Expressivity

Description logic has limited expressivity on purpose; this is the key to its good computational complexity properties, i.e. its performance properties even in worst-case situations. The following are a few examples of what can be expressed and what can not be expressed. We assume an application containing the following classes:

- `persons`
- `books`
- `languages`
- `countries`

We also assume the following relations:

- `owns` between persons and books
- `written-in` between books and languages
- `knows` between persons and languages
- `lives-in` between persons and countries
- `has-language` between countries and languages

The set of persons that own at least one book that is written in Icelandic language can then be written as follows.

```
(those persons that owns some
       (those books that written-in some {icelandic}) )
```

In the subexpression on the second line it does not matter if one puts `all` or `some` since what follows is a class of one single member. In the following examples we shall omit `some` or `all` in such cases.

The set of persons that own a book in a language that they do not know: this set can not be expressed in description logic as defined here. (There are however extensions where it can be expressed, but then complexity properties are lost).

However, consider instead the statement "John owns a book that is written in a language that he does not know." This statement can in fact be expressed, by successively defining the following classes:

1. Books that are owned by John:

   ```
   (those books that (rev owns) {john})
   ```

2. Those languages that books owned by John are written in:

   ```
   (those languages that (rev written-in) some
          (those books that (rev owns) {john}) )
   ```

3. Those languages that John knows:

   ```
   (those languages that (rev knows) {john})
   ```

4. Those languages that John does not know:

```
(languages ⊓
  (comp (those languages that (rev knows) {john})) )
```

Actually the following shorter expression can be used instead:

```
(languages ⊓ (comp (those-that (rev knows) {john})))
```

The problem is then solved if we can state that the classes in item 2 and item 4 have a common member. This can be written as

```
A ⊓ B /= ⊥
```

for expressing that the intersection of the two classes `A` and `B` is not the empty class.

2.5 Published Literature on Description Logic

The literature on Description Logic has developed its own notation and terminology. The following is a dictionary of terms in description logic, with translations:

- role = relation
- concept = class
- individual = object
- concept assertion = positive literal using ϵ
- role assertion = positive literal using one of the relations
- general concept inclusion = positive literal using $\sqsubseteq$, i.e. subsumption statement
- A-box = fact base (A for 'assertion')
- T-box = terminology base

The OWL terminology uses "class" and "object," but uses the term "property" instead of relation.

The publication notation for description logic uses the following conventions. The $\sqsubseteq$ predicate is used like above. Membership is written as $x : C$ for $x \epsilon C$, and relation literals for objects are written as $(x, y) : r$ for `[r x y]` .

For term expressions, the $\sqcup$ and $\sqcap$ functions are used like here. The complement class is written as $\neg B$ for `(compl` B`)` . The reverse operator on relations is written as an exponent dash, so `(rev` r`)` is r^-. The `those-that` operator is written as follows:

- $\exists\ r.B$ for `(those-that` r `some` B`)`
- $\forall\ r.B$ for `(those-that` r `all` B`)`
- $\leq n\ r.B$ for `(those-that` r `at most` n B`)`

This is a quite compact notation that is maybe convenient when one is studying the theoretical properties of the notation, but it has the disadvantages of not going so well with the conventional use of the quantifiers, and of not being very natural when concrete examples are to be written out. It is for these reasons that we have used the CLE notation here.

The Description Logic Website [3] provides a wealth of further information about description logics, including a list of description logic reasoners.

[3] http://dl.kr.org/

Chapter 3

Defeasible Inheritance

Defeasible Inheritance addresses the problem when there is a certain number of exceptions to the subsumption relation between classes. This problem has been extensively studied for a long time in Artificial Intelligence research. The present chapter will describe the standard view of this topic, some examples from the classical reportoire of "difficult cases," but also a recent development in this area that brings much additional clarity to the topic. This recent development will be used as frame of reference for the presentation.

3.1 Basic Approach to Defeasible Inheritance

3.1.1 Semantic Assumptions

As basic framework we uses classes of objects and the subsumption relation $\sqsubseteq$ from the treatment of Description Logic. Objects and membership are not used, and it is assumed that classes have a reasonable number of members. (This will be modified later on). Three additional predicates are introduced:

- `[sub .c .d .m]` expresses that the class `.c` is defeasibly subsumed by the class `.d` i.e. there may be some exceptions
- `[dj .c .d]` expresses that the classes `.c` and `.d` are disjoint i.e. they have no common member; no exceptions allowed
- `[nsub .c .d .e]` is used for declaring an exception to the transitivity of `sub`. This will be explained below.

The third argument `.m` of the `sub` predicate will be used for a *doubt index* that will be introduced later on, and for the beginning discussion we shall omit this third argument.

An *inheritance network* consists of a set of positive literals using these predicates. The *completion* of such an inheritance network is obtained by adding more literals that are obtained as conclusions from the given ones. For example, if `[sub horses mammals]` and `[mammals land-animals]` are in the given network, then `[sub horses land-animals]` should be in its extension.

There are some positive and some negative requirements on network extensions. Positive requirements specify conclusions that must be drawn, for example (tentatively):

```
(imp (and [sub .c .d][sub .d .e]) [sub .c .e])
```

Negative requirements specify situations that must not be present there, for example through the following requirement on the extension:

```
(not (and [sub .c .d][sub .c .e][dj .d .e]))
```

It is assumed that all given classes have a non-empty set of members. This assumption is necessary for the above restriction to be applicable.

Finally, `(imp [⊑ .c .d] [sub .c .d] )`

If there are not any exceptions to the subsumption then all of this is very simple. It is the possibility of exceptions that complicate the matter.

3.1.2 On-Path Preclusion

Two basic kinds of structures are at the heart of the matter. One is the issue of *preclusion.* Consider the following inheritance network.

```
GA dj WA
E sub GA
RE sub E
RE sub WA
C sub RE
```

In this structure, the use of transitivity for `sub` obtains both that `[C sub GA]` and `[C sub WA]` but this is a contradiction since we also have `[GA dj WA]`

The acronyms in this example are derived from the classical toy example for illustrating this structure, where `E` stands for the class of elephants, `GA` and `WA` stand for gray animals and white animals, respectively, `RE` stands for royal elephants which are supposed to be white as an exception to ordinary elephants that are gray, and `C` stands for a particular royal elephant ("Clyde") or a group of specific royal elephants.

Intuitively, it seems that we should prefer the conclusion `[C sub WA]` over the conclusion `[C sub GA]` since it is based on more specific information. The information about `RA` *precludes* the chain of links from `C` to `GA` to take effect.

Exceptions like in this toy example are very common in class structures that arise in practical applications, and it is necessary to have some way of handling them and living with them. Here we shall first describe a recently developed approach, and later on proceed to other proposed approaches.

In order to deal with exceptions we modify the transitivity axioms for `sub` so that it is as follows:

```
(imp (and [sub .c .d] [sub .d .e] [-nsub .c .d .e])
     [sub .c .e] )
```

The contradiction can then be avoided by adding the following literals to the inheritance network:

```
[nsub RE E GA]
[nsub C E GA]
```

Both of these are needed in order to suppress the conclusions that otherwise lead to a contradiction. The first one is needed since other wise one obtains that royal elephants are both gray and white, and the second one is needed since otherwise one obtains `[C sub E]` and from there `[C sub GA]` . However, it turns out that the following restriction can be imposed:

```
(imp (and [nsub .d .e .g] [sub .c .d])  [nsub .c .e .g])
```

If this rule is added to the set of general restrictions on extensions of inheritance networks then it is sufficient to only have one `nsub` literal in our example network, namely

```
[nsub RE E GA]
```

and the rest will follow by inference.

3.1.3 Directly Contradictory Subsumptions – the Nixon Diamond

The following small example exemplifies the other major type of complication.

```
P dj AP
Q sub P
R sub AP
N sub Q
N sub R
```

This example also obtains two contradictory conclusions, like the previous one, namely `[N sub P]` and `[N sub AP]` but here the two paths are entirely symmetric. The example is usually called the "Nixon Diamond" since it referred to U.S. president Richard Nixon who was a quaker, but also a republican, and quakers were known to be pacifists whereas republicans were assumed to be "anti-pacifists" – remote from any pacifist position. [1]

In the previous example it was also fairly obvious how to insert the `nsub` literal, merely given the structure of the network and without knowing the meaning of its nodes, but here this is not possible due to the symmetry. One can see two possibilities, therefore: either use additional information from the application so as to obtain the right choice of `nsub` literal, or introduce *two* `nsub` literals, namely

```
[nsub N R AP]
[nsub N Q P]
```

so that neither conclusion is possible. If one does not know which is the case then this is of course the reasonable thing to do.

[1] The diagram was also drawn a bit differently, which made it look like a rhombus, that is, a "diamond" in the sense of American English.

3.1.4 Adding Exception Literals by Default

Suppose now that a very large inheritance network has been provided and the persons using it have merely provided the subsumption literals, but not the `nsub` literals that are necessary in order to characterize the exceptions from the general subsumption statements. One can easily write a general-purpose program that identifies whether the given network is inconsistent, for example in the ways that was shown in our two examples. The question is then: will it be necessary to go back to the users and ask them to modify their networks, probably with additional `nsub` literals, so that they become correct, or will it be possible to have an automatic way of repairing the given inheritance network?

It has not yet been demonstrated conclusively if there exists such a repair mechanism that works to satisfaction in all cases, but some things are anyway known about this problem. Two types of methods are being proposed, *single-extension methods* and *multiple-extension methods.* Single-extension methods are those that generate one single extension, for example by adding a number of `nsub` literals. Multiple-extension methods generate *several* extensions and take the intersection between them, that is, they accept those inferred literals that are present in all the extensions obtained.

Single-extension methods are also called *sceptical* ones, and multiple-extensions methods are also called *credolous* ones. The difference between these methods appears the most clearly for directly contradictory subsumptions, like in the Nixon diamond. In a single-extension approach, because of the symmetry of the given network, there is no choice but to introduce `nsub` literals for both or all of the paths in the network that cause a contradiction, for example, the two literals shown in the previous subsection.

In multiple-extension approaches one may consider network repair rules that obtain larger extensions. In the Nixon diamond example, one may have one extension where `[nsub N R AP]` is added, and another extension where `[nsub N Q P]` is added. The first extension will then contain `[N sub P]` and the other extension will contain `[N sub AP]` but the intersection of these extensions will contain neither of those.

Such a variant of the multiple-extension approach will therefore obtain the natural conclusion in the particular case of the basic Nixon diamond. Unfortunately it may lead to problems, although this depends on exactly how one defines the repair mechanism. Consider an extension of the basic Nixon diamond example where one has added the following two literals:

```
P sub IM
AP sub IM
```

If one wishes to have a concrete interpretation of the classes involved, one may read `IM` as "ideologically motivated." In this case both extensions will contain the inferred literal `[N sub IM]` and therefore it will be taken to be a conclusion of the given inheritance network (or at least this is what strikes the mind at first when you see the example). Such a conclusion, which is obtained because it is present in each one of several extensions, although different chains of inference were used in those extensions, is called a *floating conclusion.*

The problem with floating conclusions in general is that they are not always well motivated, and that is also the case in this example. The problem is

that there is no a priori reason why the objects in the class N should belong to one or the other of the two classes P and AP. From a commonsense point of view one can observe that there may be many intermediate positions concerning the use of military force, between "pacifist" and "antipacifist." From a formal point of view, which is more solid, one must observe that it has merely been stated that the two classes P and AP are disjoint, but they have not been stated to be the complement of each other. Therefore there is no support for obtaining [N sub IM] as a conclusion.

3.1.5 Minimal Extension with Respect to nsub

Let us return to the question that was posed above: is it possible to define some method whereby missing nsub literals can be added automatically, based on an analysis of a given inheritance network without such literals? The best available candidate for such a method is arguably a multiple-extension method based on minimization of the nsub predicate, that is, based on obtaining one or more sets of argument combinations for nsub which are as small as possible, obtaining the corresponding extensions of the given inheritance network, and then making a combination of those extensions. In this case, "smallest possible" means that if any of the members of that set is removed, then the resulting extension is contradictory. Also, the "extensions" are defined as sets of literals that contain all the literals in the given inheritance network, plus those literals that are necessary in order to satisfy a set of given axioms.

In the specificity example, in particular, we showed that it is necessary to add some nsub literal in order to avoid a contradiction, and it is sufficient to add the literal [nsub RE E GA] and in the "Nixon" example it is necessary and sufficient to add one of the two literals [nsub N R AP] and [nsub N Q P] .

The remaining problem – and the crucial problem for these methods – is what kind of "combination" shall be used for those extensions. The sub *intersection method* consists of just taking their intersection of these extensions and using its contents of sub literals. This will run into the problem with floating conclusions, but that is the only known problem with it. [2]

Another possibility is the nsub *union method* which consists of first taking the union of all the nsub literals in all those extensions, add them to the originally given inheritance network, and again obtain all the conclusions using the given set of axioms. This is guaranteed to be a contradiction-free network, and with respect to its contents of sub literals it will be a subset of, or equal to the result from the first method. At this point there is not sufficient information to say whether it loses some conclusions that it should have been able to draw, and that are obtained using the first method.

Both of these methods obtain the intended results for the preemption example with the elephants and for the simple "Nixon" problem. With respect to the extended "Nixon" problem with the floating conclusion, the sub intersection method obtains no conclusions, as intended, for the relations between N on one hand and P and AP on the other, but it does obtain an unintended floating conclusion.

[2] There are some additional details that have to be introduced for this method, but which will be omitted here.

The `nsub` union method does not do any better on this example, given the rules shown above. There will still be two extensions, the union of their `nsub` contents consists of `[nsub N R AP]` and `[nsub N Q P]` , and therefore there is no conclusion *in the final extension* with respect to the relations between `N` on one hand and `P` and `AP` on the other. It would seem then that the floating conclusion is not obtained. Unfortunately, however, it is obtained through another path since one obtains both `[R sub IM]` and `[Q sub IM]` , and from there one obtains `[N sub IM]`

3.1.6 Introducing the `dsub` Predicate

We now arrive at the final correction that must be made to the approach in order to make it work right, at least as far as is known at present. The problem with the floating conclusion for the `nsub` union method leads to the solution. We introduce an auxiliary predicate `dsub` which stands for "derived subsumption" and make the convention that the original predicate `sub` will only be used for literals in the *given inheritance network,* and not for inferred literals. The transitivity axiom is modified as follows:

```
(imp (and [dsub .c .d] [sub .d .e] [-nsub .c .d .e])
     [dsub .c .e])
```

In addition there is the following axiom:

```
(imp [sub .d .e] [dsub .d .e] )
```

The effect of the first axiom is that it is only possible to use the transitivity of subsumption in the "upward" direction, by gradually proceeding one step up from the uppermost node in the chain, and it is not possible to extend it in the "downward" direction.

With this modification the `nsub` union method avoids obtaining the (unintended) floating conclusion. The `sub` intersection method will unfortunately still suffer from obtaining that floating conclusion.

3.1.7 Summary of the Basic Approach

We have now gradually introduced the required constructs, in such a way that our specific examples have provided the explanation for why various constructs were necessary. Here is a summary of the conventions that have been introduced in this section. The following predicates are used

- `[sub .c .d .m]` expresses a statement in the *given* inheritance network that the class `.c` is defeasible subsumed by the class `.d` i.e. there may be some exceptions. The last argument has not yet been put in use
- `[dsub .c .d .m]` is similar to the previous predicate, but it is used for *inferred* literals
- `[dj .c .d]` expresses that the classes `.c` and `.d` are disjoint i.e. they have no common member; no exceptions allowed
- `[nsub .c .d .e]` is used for declaring an exception to the transitivity of `sub.` This will be explained below.

The following axioms are used, with the exclusion of the third argument for `sub` and `dsub`

```
(imp (and [dsub .c .d] [sub .d .e] [-nsub .c .d .e])
     [dsub .c .e])

(not (and [dsub .c .d][dsub .c .e][dj .d .e]))

(imp (and [nsub .d .e .g] [sub .c .d])  [nsub .c .e .g])

(imp [⊑ .d .e] [sub .d .e] )

(imp [sub .d .e] [dsub .d .e] )
```

The full specification actually contains a few more axioms, but they have a technical nature and are not necessary here.

3.2 Additional Interesting Cases

The literature on defeasible inheritance contains a number of *test scenarios* that are useful for the process of verifying that proposed methods do obtain intended results. However, it must be understood that their value is only as test cases: they can provide counterexamples to proposed methods, and if there are no counterexamples then they can provide a certain, moderate credibility for the method, but they can never be proofs of correctness. That requires a systematic definition of, and use of a semantics for the predicates that are involved.

3.2.1 Off-Path Preclusion

Consider the following extension of the "elephant" scenario:

```
GA dj WA
E sub GA
RE sub E
RE sub WA
C sub RE
C sub CE
CE sub E
```

The last two literals have been added. The reader is encouraged to draw a diagram showing the connections between the classes. The structure in this example is more complex since now there are two distinct paths from `C` to `GA` , namely via `CE` and via `RE,` but both proceed through `E.` Should one or should one not allow an extension containing `[C sub GA]` in this example, which would make it similar to the "Nixon" example in the sense that no conclusion is finally obtained concerning whether `C` is in `WA` or is in `GA` ? This is called a situation with *off-path preclusion.* The principled analysis of this situation is as follows. There are two possibilities with respect to `CE` and `RE` . Either they have strongly overlapping contents, so that they are almost equal with respect to contents, or not. In the former case, we have no reasonable reason supporting `[C sub GA]` but in the latter case

the path via CE provides independent information so that the link from C to GA is as likely as the one to WA.

Moreover, with respect to the formal inference, in the former case there should be sub relations both ways between CE and RE ; in the opposite case not. This is because we must assume that the given inheritance network *provides complete information* about the subsumption links between classes, except for those cases where these links can reasonably be inferred; it is only the nsub literals that maybe should be inferred automatically.

Now, the structure of the given network is such that we will necessarily have [nsub RE E GA] which means that in the former case we will also have [nsub CE E GA] using one of the axioms. We will obtain [C sub E] but not [C sub GA] because of the nsub literal, which is what is intended.

In the latter case, however, we will also obtain [nsub C E GA] and this will preclude the subsumption chain via CE from being used, and in this case this is an unintended effect.

The solution to this problem is to generalize the transitivity axiom for dsub so that it looks as follows.

```
(imp (and [dsub .c .d] [sub .d .e] [sub .e .g]
          [-nsub .c .d .e] [-nsub .d .e .g] )
     [dsub .c .g])
```

The previously stated axiom transitivity axiom can be obtained as a special case of this one. This takes care of one-step off-path preclusion as in this example.

One can also construct structures with *two-step off-path preclusion* where there is one more off-path class, similar to CE in this example, but one step up along the subsumption chain from C to GA. This requires are further more general axiom where there are four subsumption links instead of three. However, it is very doubtful whether two-step or other multip-step off-path preclusion structures exist in practice, so the axiom shown above is likely to be sufficient for practical purposes, as far as we know today.

3.2.2 Choice of Breakpoint

Next, consider the following example

```
A sub B
B sub C
C sub D
D dj G
A sub G
```

Should we obtain [A dsub C] as a conclusion in this case?

There may be some intuitive arguments in favor of that conclusion, along the line of "it has not been negated." There may also be intuitive arguments in the opposite direction, such as "if this were the case then [sub A D] would follow, but we know that that is not the case, therefore no."

The inference methods that were described in the previous section will not obtain the conclusion [A dsub C] and this applies in fact for both the sub intersection method and the nsub union method.

Here is one concrete example that agrees with the latter position.

```
CitizenOfGuyana sub LivesInLatinAmerica
LivesInLatinAmerica sub SpouseHispSpeaking
SpouseHispSpeaking sub HispanicSpeaking
HispanicSpeaking dj EnglishSpeaking
CitizenOfGuyana sub EnglishSpeaking
```

where `HispanicSpeaking` is the class of those having Spanish or Portuguese as their first language, and `SpouseHispSpeaking` is the class of those persons whose husband or wife has either of these as their first language. We would not like to infer that most citizens of Guyana (which is an English-speaking country) are married to Hispanic-speaking persons.

So what is the right answer? In order to have a reliable answer to that question we need a formal definition of the semantics for the `sub` relation as well as for the other ones used here. This is the topic of the next section.

3.2.3 Conclusion

Chapter 2 has introduced Description Logic, Chapter 3 has introduced Defeasible Inheritance. It is a natural next step to combine these two, so that defeasible inheritance can be applied to classes as described in description logic. The basic idea is straightforward, but the technical details of how this can be done correctly and efficiently is a topic of current research.