10 Basic OWL

In the previous chapters, we have seen how OWL-Fast as a modeling system provides considerable support for distributed information and federation of information. Simple constructs in OWL-Fast can be combined in various ways to match properties, classes and individuals. We have seen its utility in application to social networking (FOAF) and knowledge organization (SKOS). But while OWL-Fast has provided considerable and valuable infrastructure for these projects, we have also identified capabilities that are required by these systems that OWL-Fast cannot provide. Fortunately, the W3C Semantic Web Standards go beyond OWL-Fast to provide a more comprehensive language called OWL, which provides a systematic treatment of information description. OWL provides constructs for describing information structure that will satisfy many of the outstanding requirements of FOAF and SKOS, as well as a number of more general information integration issues.

We continue our presentation of OWL with a treatment of *owl:Restriction*. This single construct opens up the representational power of OWL by allowing us to describe classes in terms of other things we have already modeled. As we shall see, this opens up whole new vistas in modeling capabilities.

10.1 Restrictions

Suppose we have defined in RDFS a class we call *BaseballTeam*, with a particular subclass called *MajorLeagueTeam*, and another class we call *BaseballPlayer*. The roster for any particular season would be represented as a property *playsFor* that relates a *BaseballPlayer* to a *BaseballTeam*. Certain players are special, in that they play for a

MajorLeagueTeam. We'd like to define that class, and call it *MajorLeaguePlayers*. If we are interested in the fiscal side of baseball, we could also be interested in the class of Agents who represent Major League Players; and then the bank accounts controlled by the Agents who represent Major League Players, etc.

One of the great powers of the semantic web is that information that has been specified by one person in one context can be re-used by others in different contexts. There is no expectation that the same source who defined the roster of players will be the one that defines the role of the agents, or of the bank accounts. If we want to use information from multiple sources together, we have to have a way to express concepts from one context in terms of concepts from the other. In OWL, this is achieved by having a facility with which we can describe new classes in terms of classes that have already been defined.

We have already seen how to define simple classes and relationships between them in RDFS and OWL, but none of the constructs we have seen so far can create descriptions of the sort we want in our Major League Baseball example. This is done in OWL using a language construct called a *Restriction*.

Let's take the example of *MajorLeagePlayer*. We informally defined a *MajorLeaguePlayer* as someone who plays on a *MajorLeagueTeam*. The intuition behind the name *Restriction* is that membership in the class *MajorLeaguePlayer* is restricted to those things that play for a *MajorLeagueTeam*. Since a *Restriction* is a special case of a *Class*, we will sometimes refer to a *Restriction* as a *Restriction Class* just to make that point clear.

More generally, a *Restriction* in OWL is a *Class* defined by describing the individuals it contains. This simple idea forms the basis for extension of models in OWL: If you can describe a set of individuals in terms of known classes, then you can use that description to define a new class. Since this new class is now also an existing class, it can be used to describe individuals for inclusion in a new class, and so on.

10.1.1 Example.Questions and Answers

Throughout this chapter, we will use a running example of questions and answers, as if we were modeling a quiz, examination or questionnaire. This is a fairly simple area that nevertheless illustrates a wide variety of uses of restriction classes in OWL.

Informally, a questionnaire consists of a number of questions, each of which has a number of possible answers. A question includes string data for the text of the question; an answer includes string data for the text of the answer. In contrast to a quiz or examination, there are typically no “right” answers in a questionnaire. In questionnaires, quizzes and examinations alike, the selection of certain answers may preclude the posing of other questions.

This basic structure for questionnaires is represented by classes and properties in OWL; any particular questionnaire is represented by a set of individual questions, answers, and concepts, and particular relationships between them.

We show the basic schema for the questionnaire in N3, and diagrammatically in Figure 10-1. Throughout the example, we will use the namespace *q:* to refer to elements that relate to questionnaires in general, and the namespace *d:* to refer to the elements of the particular example questionnaire.

```
q:Answer a owl:Class.
q:Question a owl:Class.

q:answerText a owl:DatatypeProperty;
     rdfs:domain q:Answer;
     rdfs:range xsd:string.

q:optionOf a owl:ObjectProperty;
     rdfs:domain q:Answer;
     rdfs:range q:Question;
     owl:inverseOf q:hasOption.

q:hasOption a owl:ObjectProperty .

q:questionText a owl:FunctionalProperty,
                 owl:DatatypeProperty;
     rdfs:domain q:Question;
     rdfs:range xsd:string.
```

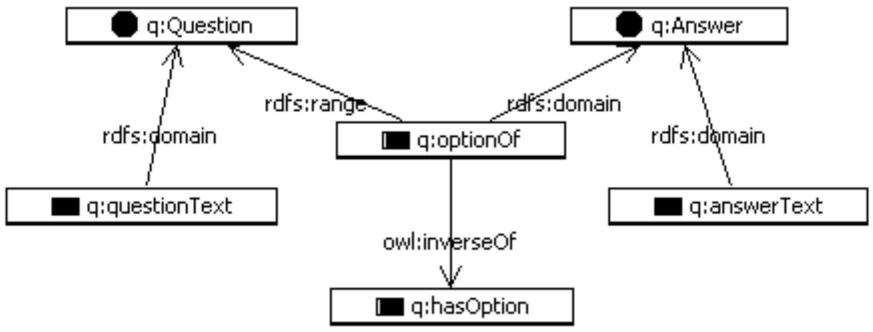


Figure 10-1. Question, Answer and properties that describe them.

A particular questionnaire will have questions and answers. For now, we will start with a simple questionnaire that might be part of the screening for the helpdesk of a cable television and internet provider:

What system are you having trouble with?

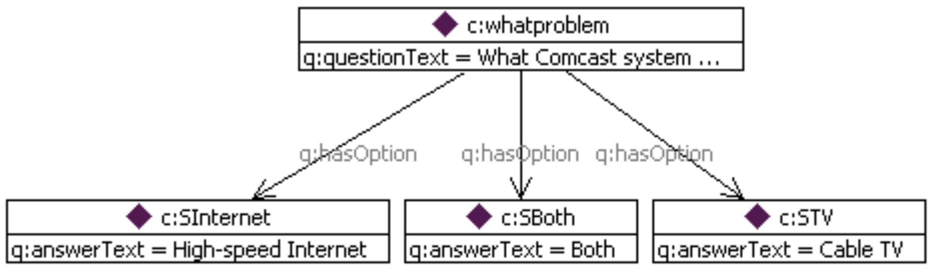
Possible answers (3): Cable TV, High-speed Internet, Both

What television symptom are you seeing?

Possible answers (4): No Picture, No Sound, Tiling, Bad reception

This is shown below in N3 and graphically in Figure 10-2.

```
d:WhatProblem a q:Question;
     q:hasOption  d:STV, d:SInternet, d:SBoth;
     q:questionText "What system are you having
trouble with?" .

d:STV a q:Answer;
     q:answerText "Cable TV".

d:SInternet a q:Answer;
     q:answerText "High-speed Internet".

d:SBoth a q:Answer;
     q:answerText "Both".

d:TVsymptom a q:Question;
     q:questionText "What television symptoms are you
having?";
     q:hasOption d:TVSnothing, d:TVSnosound,
d:TVStiling, d:TVSreception .

d:TVSnothing a q:Answer;
     q:answerText "No Picture".

d:TVSnosound a q:Answer;
     q:answerText "No Sound".

d:TVStiling a q:Answer;
     q:answerText "Tiling".

d:TVSreception a q:Answer;
     q:answerText "Bad reception".
```



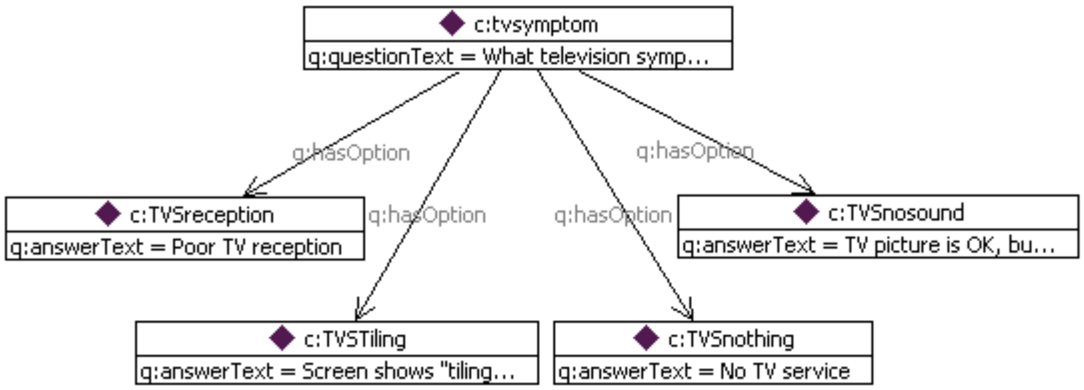

c:tvsymptom
q:questionText = What television symp...
q:hasOption
q:hasOption
q:hasOption
q:hasOption
c:TVSreception
q:answerText = Poor TV reception
c:TVSnosound
q:answerText = TV picture is OK, bu...
c:TVSTiling
q:answerText = Screen shows "tiling...
c:TVSnothing
q:answerText = No TV service

Figure 10-2. Some particular questions and their answers.

Consider an application for managing a questionnaire in a web portal. This application performs a query against this combined data to determine what question(s) to ask next. Then for each question, it presents the text of the question itself, and the text of each answer, with a select widget (e.g., radio button) next to it. We haven't yet defined enough information for such an application to work; we have made no provisions to determine which questions to ask before any others, or how to record answers to questions. We start with the latter.

We first define a new property *hasSelectedOption*, a subproperty of *hasOption*:

```
q:hasSelectedOption a owl:ObjectProperty;
     rdfs:subPropertyOf q:hasOption .
```

When the user who is taking a questionnaire answers a question, a new triple will be entered to indicate that a particular option for that question has been selected. That is, if the user selects "Cable Television" from the options of the first question *d:WhatProblem*, then the application will add the triple

```
d:WhatProblem q:hasSelectedOption d:STV .
```

to the triple store. Notice that there is no need to remove any triples from the triple store; the original *d:hasOption* relationship between *d:WhatProblem* and *d:STV* still holds.

As we develop the example, the model will provide ever increasing guidance for how the selection of questions will be done. □

10.1.2 Why "Restriction"?

The language construct in OWL for creating new class descriptions based upon descriptions of the prospective members of a class is called the Restriction (*owl:Restriction*). An *owl:Restriction* is a special kind of class (i.e., *owl:Restriction* is a *rdfs:subClassOf owl:Class*). A Restriction is a class that is defined by a description of its members, in terms of existing properties and classes.

In OWL, as in RDF, the AAA slogan holds; any statement about an individual is allowed. Hence, the class of all things in owl (*owl:Thing*) is unrestricted. A *Restriction* is defined by providing some description that limits (or restricts) the kinds of things that can be said about a member of the class. So if we have a property *orbitsAround*, it is perfectly legitimate to say that anything *orbitsAround* anything else. But if we restrict the value of *orbitsAround* by saying that its object must be *TheSun*, then we have defined the class of all things that orbit around the Sun.

10.1.3 Example. Answered Questions

In the questionnaire example, we have addressed the issue of recording answers to questions, by defining a property *hasOption* that relates a question to answer options, and a subproperty *hasSelectedOption* to indicate those answers that have been selected by the user taking the questionnaire. Now we want to address the problem of selecting which question to ask. There are a number of considerations that go into such a selection, but one of them is that (under most circumstances) we do not want to ask a question for which we already have an answer. This suggests a class of questions that have already been answered. We will define the set of *AnsweredQuestions* in terms of the properties we have already defined. Informally, an answered question is any question that has a selected option. □

10.1.4 Kinds of Restrictions

OWL provides three kinds of restriction, indicated by the three owl keywords *owl:allValuesFrom*, *owl:someValuesFrom* and *owl:hasValue*. Each of these describes how the new class is constrained by the possible asserted values of properties.

Additionally, a restriction class in OWL is defined by the keyword *owl:onProperty*. This specifies what property is to be used in the definition of the restriction class. For example, the restriction defining the objects that orbit around the Sun will use *owl:onProperty orbitsAround*; the restriction defining major league players will use *owl:onProperty playsFor*.

A restriction is a special kind of a class, so it has individual members just like any class. But membership in a restriction class must satisfy the conditions specified by the kind of restriction (*owl:allValuesFrom*, *owl:someValuesFrom* or *owl:hasValue)* as well as the *onProperty* specification.

owl:someValuesFrom

owl:someValuesFrom is used to produce a restriction of the form, "All individuals for which some value of the property P comes from class C", e.g., one could define the class *AllStarPlayer* as "All individuals for which some value of the property *playsFor* comes from the class *AllStarTeam*." In N3, this restriction looks like

```
[ a owl:Restriction;
     owl:onProperty :playsFor;
     owl:someValuesFrom :AllStarTeam]
```

Notice the use of the […] notation in N3; as a reminder from chapter 3, this refers to an anonymous node (a *bnode*) described by the properties listed here; that is, this refers to a single bnode, which is the subject of three triples, one per line (separated by semi-colons).

The restriction class defined in this way refers to exactly the class of individuals that satisfy these conditions on *playsFor* and *AllStarTeam*. In particular, if an individual actually has some value from the class *AllStarTeam* for the property *playsFor*, then it is a member of this restriction class.

10.1.4.1.1 Example. Answered Questions

An answered question is one that has some value from the class *Answer* for the property *hasSelectedOption*. This can be defined in N3 as

```
q:AnsweredQuestion owl:equivalentClass
[ a owl:Restriction;
          owl:onProperty q:hasSelectedOption;
          owl:someValuesFrom q:Answer ] .
```

Since

```
d:WhatProblem q:hasSelectedOption d:STV.
```

and

```
d:STV a Answer.
```

are asserted triples, the individual *d:WhatProblem* satisfies the conditions defined by the restriction class. That is, there is some value (*someValue*) for the property *hasSelectedOption* that is in the class *Answer*. Individuals that satisfy the conditions specified by a restriction class are inferred to be members of it. In N3, this inference can be represented as follows:

```
d:WhatProblem a [ a owl:Restriction;
                  owl:onProperty q:hasSelectedOption;
                  owl:someValuesFrom q:Answer ]
```

and hence, according to the semantics of *equivalentClass*,

```
d:WhatProblem a AnsweredQuestion.
```

These definitions and inferences can be seen in Figure 10-3.

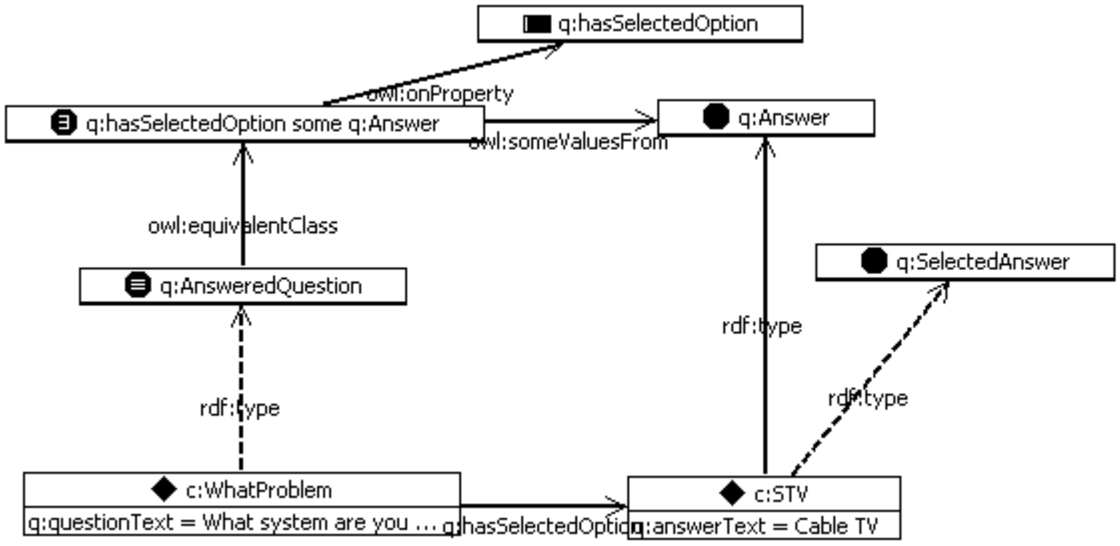


Figure 10-3. Definition of q:AnsweredQuestion, and the resulting inferences for d:WhatProblem. Since d:WhatProblem has something (d:STV) of type q:Answer on property q:hasSelectedOption, it is inferred (dotted line) to be a member of AnsweredQuestion.

□

owl:allValuesFrom

owl:allValuesFrom is used to produce a restriction class of the form, “the individuals for which all values of the property P come from class C. In N3, this restriction looks like

```
[ a owl:Restriction;
     owl:onProperty P;
     owl:allValuesFrom C]
```

The restriction class defined in this way refers to exactly the class of individuals that satisfy these conditions on P and C. If an individual *x* is a member of this *allValuesFrom* restriction, a number of conclusions can follow, one for each triple describing *x* with property *P*. In particular, every value of property *P* for individual *x* is inferred to be in class *C*. So if individual *AllStarTeam* (a member of the class *BaseballTeam*) is a member of the restriction class defined by *owl:onProperty hasPlayer;* and *owl:allValuesFrom StarPlayer*, then every player on the *AllStarTeam* is a *StarPlayer*. If *AllStarTeam hasPlayer Kaneda* and *AllStarTeam hasPlayer Gonzales*, then both *Kaneda* and *Gonzales* have type *StarPlayer*.

10.1.4.1.2 Example. Question Dependencies

In our questionnaire example, we might want to ask certain questions only after particular answers have been given. To accomplish this, we begin by defining the class of all selected answers, based on the property *hasSelectedOption* we have already defined. We can borrow a technique from Chapter 6XXX to do this. First, we define a class for the selected answers:

```
q:SelectedAnswer a owl:Class ;
     rdfs:subClassOf q:Answer .
```

We want to ensure that any option that has been selected will appear in this class. This can be done easily by asserting that

```
q:hasSelectedOption rdfs:range q:SelectedAnswer .
```

This ensures that any value *V* that appears as the object of a triple of the form

```
? q:hasSelectedOption V .
```

Is a member of the class *SelectedAnswer*. In particular, since we have asserted that

```
d:WhatProblem q:hasSelectedOption d:STV .
```

we can infer that

```
d:STV a q:SelectedAnswer .
```

Now that we have defined the class of selected answers, we describe the questions that can be asked, only after those answers have been given. We introduce a new class called *EnabledQuestion*; only questions that also have type *EnabledQuestion* are actually available to be asked.

```
q:EnabledQuestion a owl:Class.
```

When an answer is selected, we want to infer that certain dependent questions become members of *EnabledQuestion*. This can be done with an *owl:allValuesFrom* restriction.

To begin, each answer potentially makes certain questions available for asking. We define a property called *enablesCandidate* for this relationship. In particular, we say that an answer enables a question, if selecting that answer causes the system to consider that question as a candidate for the next question to ask.

```
q:enablesCandidate a owl:ObjectPropery;
           rdfs:domain q:Answer ;
           rdfs:range q:Question .
```

In our example, we only want to ask a question about television problems, if the answer to the first question indicates that there is a television problem.

```
d:STV q:enablesCandidate d:TVsymptom.
d:SBoth q:enablesCandidate d:TVsymptom.
```

That is, if the answer to the first question, “What system are you having trouble with?” is either “Cable TV” or “Both”, then we want to be able to ask the question, “What television symptoms are you having?”

The following *owl:allValuesFrom* restriction does just that; it defines the class of things all of whose values for *d:enablesCandidate* comes from the class *d:EnabledQuestion*:

```
[ a owl:Restriction;
     owl:onProperty q:enablesCandidate;
     owl:allValuesFrom q:EnabledQuestion]
```

Which answers should enforce this property? Not just any answer; we only want this for the answers that have been selected. But how do we determine which answers have been selected? So far, we only have the property *hasSelectedOption* to indicate them.

That is, for any member of *SelectedAnswer*, we want it to also be a member of this restriction class. This is exactly what the relation *rdfs:subClassOf* does for us.

```
q:SelectedAnswer rdfs:subClassOf
   [ a owl:Restriction;
        owl:onProperty q:enablesCandidate;
        owl:allValuesFrom q:EnabledQuestion].
```

Let’s watch how this works, step by step. When the user selects the answer “Cable TV” for the first question, the type of d:STV is asserted to be *SelectedAnswer*, as above.

```
d:STV a q:SelectedAnswer.
```

However, because of the *rdfs:subClassOf* relation, *d:STV* now has the restriction as a type:

```
d:STV a
   [ a owl:Restriction;
        owl:onProperty q:enablesCandidate;
        owl:allValuesFrom q:EnabledQuestion].
```

Any individual that is a member of this restriction necessarily satisfies the *allValuesFrom* condition; that is, any individual that it is related to by *d:enablesCandidate* must be a member of *d:EnabledQuestion*. Since

```
d:STV q:enablesCandidate d:TVsymptom.
```

we can infer that

d:TVsymptom a q:EnabledQuestion.

as desired.

Finally, since we have also asserted the same information for the answer *d:SBoth*,

```
d:SBoth q:enablesCandidate d:TVsymptom.
```

We can see this inference and the triples that led to it in Figure 10-4.

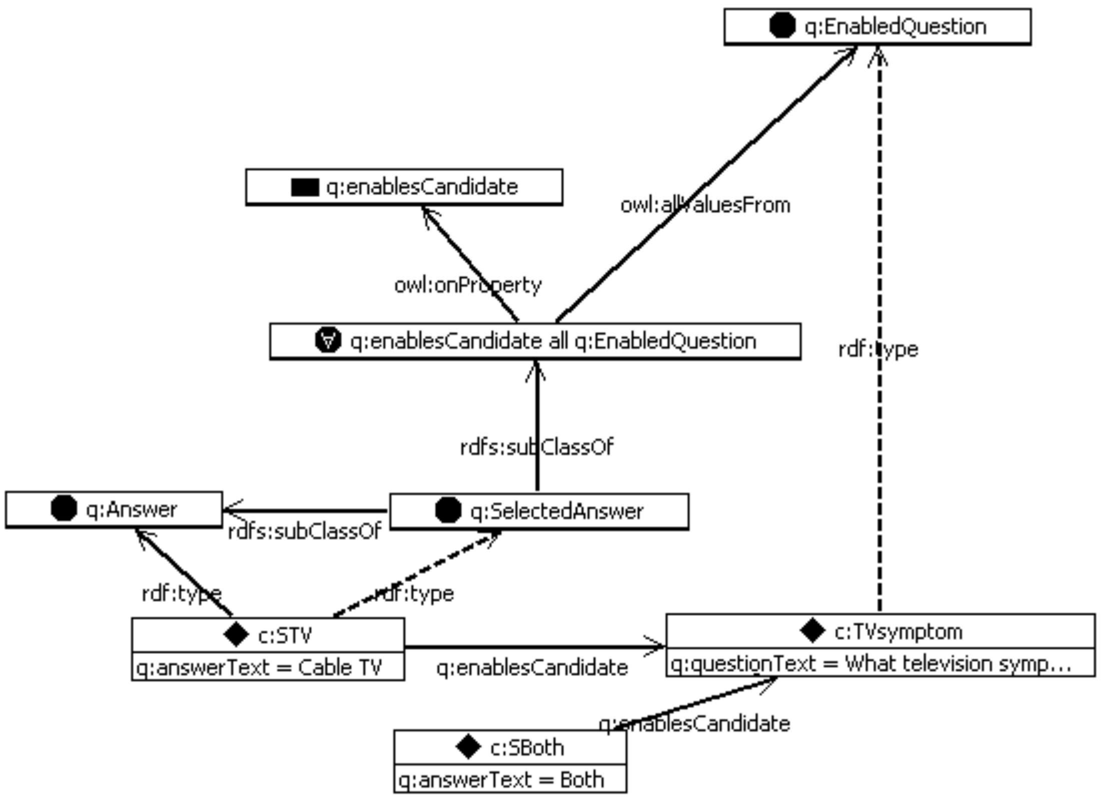


Figure 10-4. d:STV enablesCandidate TVSymptom. But it is also a member of a restriction on the property enablesCandidate, stipulating that all values must come from the class q:EnabledQuestion. We can therefore infer that d:TVSymptom has type q:EnabledQuestion.

Since *SBoth* also enables the candidate *TVSymptom*, the same conclusion will be drawn if the user answers "Both" to the first question. If we were to extend the example with another question about Internet symptoms *d:InternetSymptom*, then we could express all the dependencies in this short questionnaire in this way as follows:

```
d:STV q:enablesCandidate d:TVsymptom.
d:SBoth q:enablesCandidate d:TVsymptom.
d:SBoth q:enablesCandidate d:InternetSymptom.
d:SInternet q:enablesCandidate d:InternetSymptom.
```

The dependency tree is shown graphically in Figure 10-5.

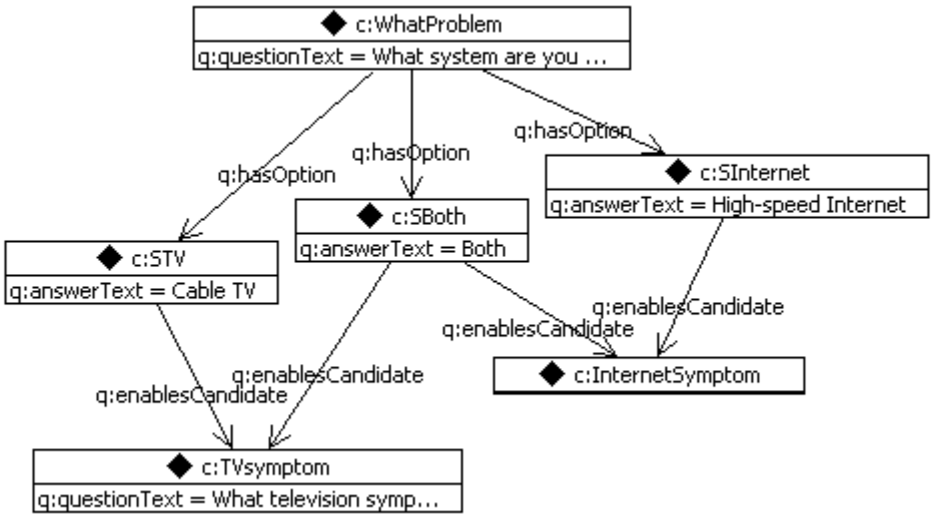


Figure 10-5. Questions and the answers that enable them.

10.1.4.1.3 Example: Prerequisites

In the previous example, we supposed that when we answered one question, that it made all of its dependent questions eligible for asking. Another way in which questions are related to one another in a questionnaire is as prerequisites. If a question has a number of prerequisites, all of them must be answered appropriately in order for the question to be eligible.

Consider the following triples defining a bit of a questionnaire:

```
d:NeighborsToo a q:Question;
     q:hasOption  d:NTY, d:NTN, d:NTDK;
     q:questionText "Are other customers in your
building also experiencing problems?" .

d:NTY a q:Answer;
     q:answerText "Yes, my neighbors are experiencing
the same problem.".

d:NTN a q:Answer;
     q:answerText " No, my neighbors are not
experiencing the same problem.".

d:NTDK a q:Answer;
     q:answerText "I don’t know.".
```

This question makes sense only if the current customer lives in a building with other customers, and is experiencing a technical problem. That is, this question depends on the answers to two more questions, shown below. The answer to the first question (*d:othersinbuilding*) should be *d:OYes*, and the answer to the second question *(d:whatissue)* should be *d:PTech*:

```
d:othersinbuilding
      a q:Question ;
      q:hasOption d:ONo , d:OYes ;
      q:questionText
              "Do you live in a multi-unit dwelling
with other customers?” .

d:OYes a q:Answer;
     q:answerText "Yes." .

d:ONo a q:Answer;
     q:answerText " No.".


d:whatIssue
      a q:Question ;
      q:hasOption d:PBilling , d:PNew, d:PCancel,
d:PTech ;
      q:questionText
              "What can customer service help you with
today?” .

d:PBilling a q:Answer;
     q:answerText "Billing question." .

d:PNew a q:Answer;
     q:answerText "New account".

d:PCancel a q:Answer;
     q:answerText "Cancel account".

d:PTech a q:Answer;
     q:answerText "Technical difficulty".
```

A graphic version of these questions can be seen in Figure 10-6 .

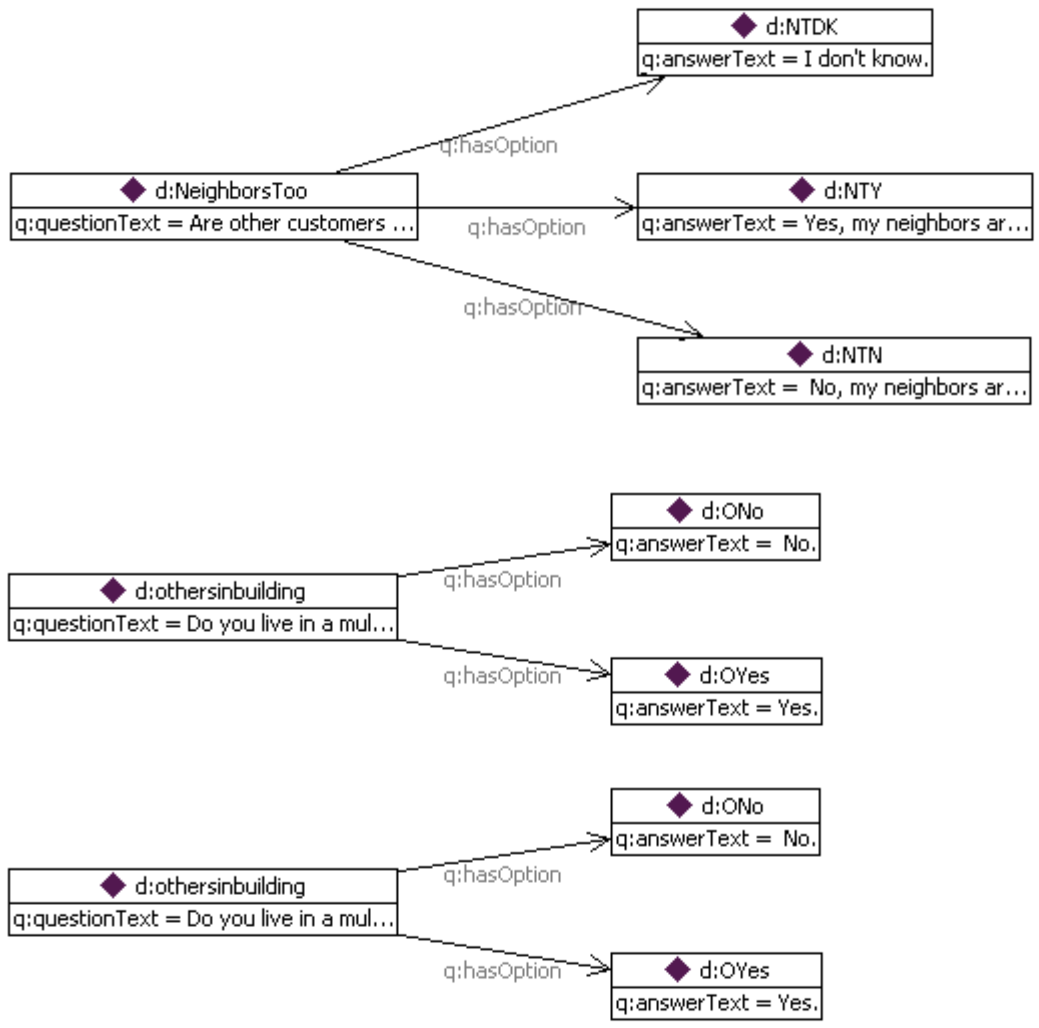


Figure 10-6. Questions about neighbors have two prerequisites questions.

CHALLENGE

How can we model the relationship between *d:NeighborsToo*, *d:whatIssue* and *d:othersinbuilding* so that we will only ask *d:NeighborsToo* when we have appropriate answers for both *d:whatIssue* and *d:othersinbuilding*?

SOLUTION

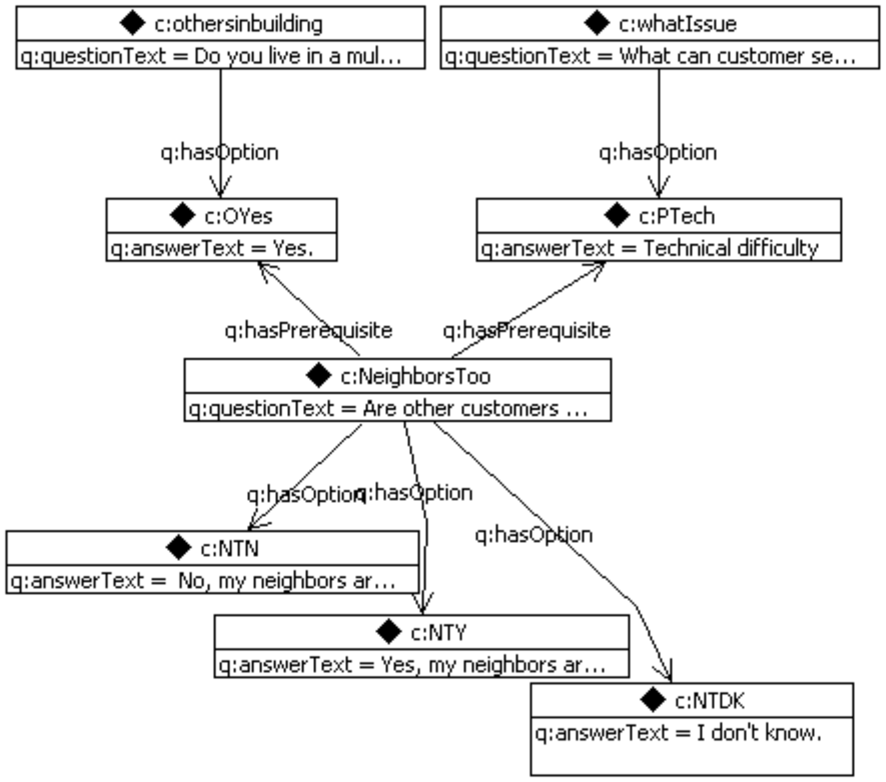
We introduce a new property *q:hasPrerequisite* that will relate a question to its prerequisites:

```
q:hasPrerequisite
      rdfs:domain q:Question ;
      rdfs:range q:Answer .
```

We can indicate the relationship between the questions using this property.

```
d:NeighborsToo q:hasPrerequisite d:PTech, d:OYes .
```

We display this prerequisite structure in graphical form in



c:othersinbuilding
q:questionText = Do you live in a mul...
c:whatIssue
q:questionText = What can customer se...
q:hasOption
q:hasOption
c:OYes
q:answerText = Yes.
c:PTech
q:answerText = Technical difficulty
q:hasPrerequisite
q:hasPrerequisite
c:NeighborsToo
q:questionText = Are other customers ...
q:hasOption
q:hasOption
q:hasOption
c:NTN
q:answerText = No, my neighbors ar...
c:NTY
q:answerText = Yes, my neighbors ar...
c:NTDK
q:answerText = I don't know.

Figure 10-7. Some questions and their prerequisites.

Now we want to say that we will infer that something is a *d:EnabledQuestion* if all of its prerequisite answers are selected. We begin by asserting that

```
[ a owl:Restriction ;
  owl:onProperty q:hasPrerequisite;
  owl:allValuesFrom q:SelectedAnswer ]
   rdfs:subClassOf q:EnabledQuestion .
```

Any question that satisfies the restriction will be inferred to be a member of *d:EnabledQuestion*. But how can we infer that something satisfies this restriction?

In order for an individual x to satisfy this restriction, we have to know that every time there is a triple of the form

```
x hasPrerequisite y .
```

that *y* must be a member of the class *d:SelectedAnswer*. But by the Open World Assumption, we don't know if there might be another triple of this form, for which *y* is not a member of *d:SelectedAnswer*. Given the Open World Assumption, how can we ever know that all prerequisites have been met?

The rest of this challenge will have to wait for section 11.6, where we discuss various methods by which we can (partially) close the world in OWL. The basic idea is that if we can say how many prerequisites a question has, then we can know when all of them have been selected. If we know that a question has only one prerequisite, and we find one that is satisfied, then it must be the one. If we know that a question has no prerequisites at all, then we can determine that it is an *EnabledQuestion* without having to check for any *SelectedAnswers* at all.

owl:hasValue

The third kind of restriction in OWL is called *owl:hasValue*. As in the other two kinds of restriction, it acts on a particular property as specified by *owl:onProperty*. It is used to produce a restriction whose description is of the form, "All individuals that have the value A for the property P". In N3, this restriction looks like

```
[ a owl:Restriction;
     owl:onProperty P;
     owl:hasValue A]
```

Formally, the *hasValue* restriction is just a special case of the *someValuesFrom* restriction, in which the class C happens to be a singleton set {A}.

Although it is 'just' a special case, *owl:hasValue* has been identified in the OWL standard in its own right because it is a very common and useful modeling form. It effectively turns specific instance descriptions into class descriptions. For example, "the set of all planets orbiting the Sun" and "the set of all baseball teams in Japan" are defined using *hasValue* restrictions.

10.1.4.1.4 Example. Priority Questions.

Suppose that in our questionnaire, we assign priority levels to our questions. First we define a class of priority levels, and particular individuals that define the priorities in the questionnaire:

```
q:PriorityLevel a owl:Class .
q:High a q:PriorityLevel .
q:Medium a q:PriorityLevel .
q:Low a q:PriorityLevel .
```

Then we define a property that we will use to specify the priority level of a question

```
q:hasPriority
      rdfs:range q:PriorityLevel .
```

We have defined the range of *q:hasPriority*, but not its domain. After all, we might want to set priorities for any number of different sorts of things, not just questions.

We can use *owl:hasValue* to define the class of high priority items:

```
q:HighPriorityItem owl:equivalentClass
[ a owl:Restriction;
          owl:onProperty q:hasPriority;
          owl:hasValue q:High ] .
```

These triples are shown graphically in Figure 10-8.

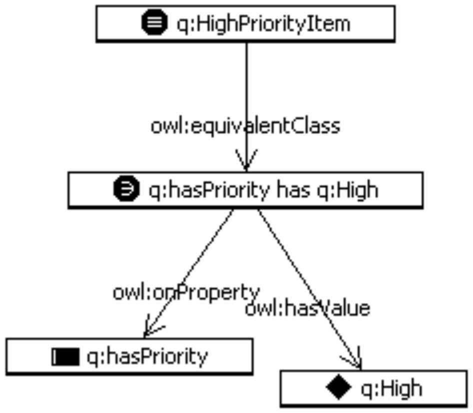


Figure 10-8. Definition of a HighPriorityItem as anything that has value High for the hasPriority property.

We can describe Medium and Low priority questions in the same manner:

```
q:MediumPriorityItem owl:equivalentClass
[ a owl:Restriction;
          owl:onProperty q:hasPriority;
          owl:hasValue q:Medium ] .
q:LowPriorityItem owl:equivalentClass
[ a owl:Restriction;
          owl:onProperty q:hasPriority;
          owl:hasValue q:Low ] .
```

If we assert the priority level of a question, e.g.,

```
d:WhatProblem q:hasPriority q:High .
d:InternetSymptom q:hasPriority q:Low .
```

then we can infer the membership of these questions in their respective classes:

```
d:WhatProblem a q:HighPriorityItem .
d:InternetSymptom a q:LowPriorityItem .
```

We can also use *owl:hasValue* to work "the other way around"; suppose that we assert that *d:TVsymptom* is in the class *HighPriorityItem*:

```
d:TVsymptom a q:HighPriorityItem .
```

Then by the semantics of *owl:equivalentClass*, we can infer that *d:TVsymptom* is a member of the restriction class, and must be bound by its stipulations. Hence we can infer that

```
d:TVsymptom q:hasPriority q:High .
```

Notice that there is no stipulation in this definition to say that a *HighPriorityItem* must be a question; after all, we might set priorities for things other than questions. The only way we know that *d:TVsymptom* is a *q:Question* is that we already asserted that fact. In the next chapter, we will see how to use set operations to make definitions that combine restrictions with other classes.□

10.2 CHALLENGE PROBLEMS

As we have seen in the examples in the previous sections, the class constructors in OWL can be combined in a wide variety of powerful ways. In this section, we will present a series of challenges that can be addressed using these constructs in OWL. Often the application of the construct is quite simple; we have chosen these challenge problems because of their relevance to modeling problems that we have seen in real modeling projects.

10.2.1 Challenge: Local Restriction of Ranges

We have already seen how *rdfs:domain* and *rdfs:range* can be used to classify data according to how it is used. But in more elaborate modeling situations, a finer granularity of domain and range inferences is needed. Consider the following example of describing a vegetarian diet:

```
:Person a owl:Class .
:Food a owl:Class .
:eats rdfs:domain :Person .
:eats rdfs:range :Food .
```

From these triples, and the following instance data

```
:Maverick :eats :Steak .
```

we can conclude two things:

```
:Maverick a :Person .
:Steak a :Food .
```

The former is implied by the domain information, the latter by the range information.

Suppose we want to define a variety of diets in more detail. What would this mean? First, let's suppose that we have a particular kind of person, called a *Vegetarian*, and the kind of food that a *Vegetarian* eats, which we will call simply *VegetarianFood* , as subclasses of *Person* and *Food* respectively:

```
:Vegetarian a owl:Class ;
    rdfs:subClassOf :Person .
:VegetarianFood a owl:Class ;
    rdfs:subClassOf :Food .
```

Suppose further that we say

```
:Jen a :Vegetarian ;
    :eats :Marzipan .
```

We would like to be able to infer that

```
:Marzipan a :VegetarianFood .
```

but not make the corresponding inference for Maverick's steak, until someone asserts that he, too, is a vegetarian.

CHALLENGE

It is tempting to represent this with more domain and range statements, thus:

```
:eats rdfs:domain :Vegetarian .
:eats rdfs:range :VegetarianFood .
```

But given the meaning *of rdfs:domain* and *rdfs:range*, we can draw inferences from these triples that we do not intend. In particular, we can infer

```
:Maverick a :Vegetarian .
:Steak a :VegetarianFood .
```

which would come as a surprise both to Maverick and the vegetarians of the world.

How can the relationship between vegetarians and vegetarian food be correctly modeled with the use of the *owl:Restriction*?

SOLUTION

We can define the set of things that only eat *VegetarianFood* using a *owl:allValuesFrom* restriction; we can then assert that any *Vegetarian* satisfies this condition using *rdfs:subClassOf*. Together, it looks like this:

```
:Vegetarian rdfs:subClassOf
   [ a owl:Restriction;
      owl:onProperty :eats ;
      owl:allValuesFrom :VegetarianFood] .
```

Let's see how it works. Since

```
:Jen a :Vegetarian .
```

we can conclude that

```
:Jen a [ a owl:Restriction;
      owl:onProperty :eats ;
      owl:allValuesFrom :VegetarianFood] .
```

Combined with the fact that

```
:Jen :eats :Marzipan .
```

We can conclude that

```
:Marzipan a :VegetarianFood .
```

as desired. How does *Maverick* fare now? We won't say that he is a *Vegetarian*, only, as we have stated already, that he is a *Person*. That's where the inference ends; there is no stated relationship between *Maverick* and *Vegetarian*, so there is nothing on which to base an inference. Maverick's steak remains simply a *Food*, not a *VegetarianFood*.

The whole model and inferences are shown in Figure 10-9.

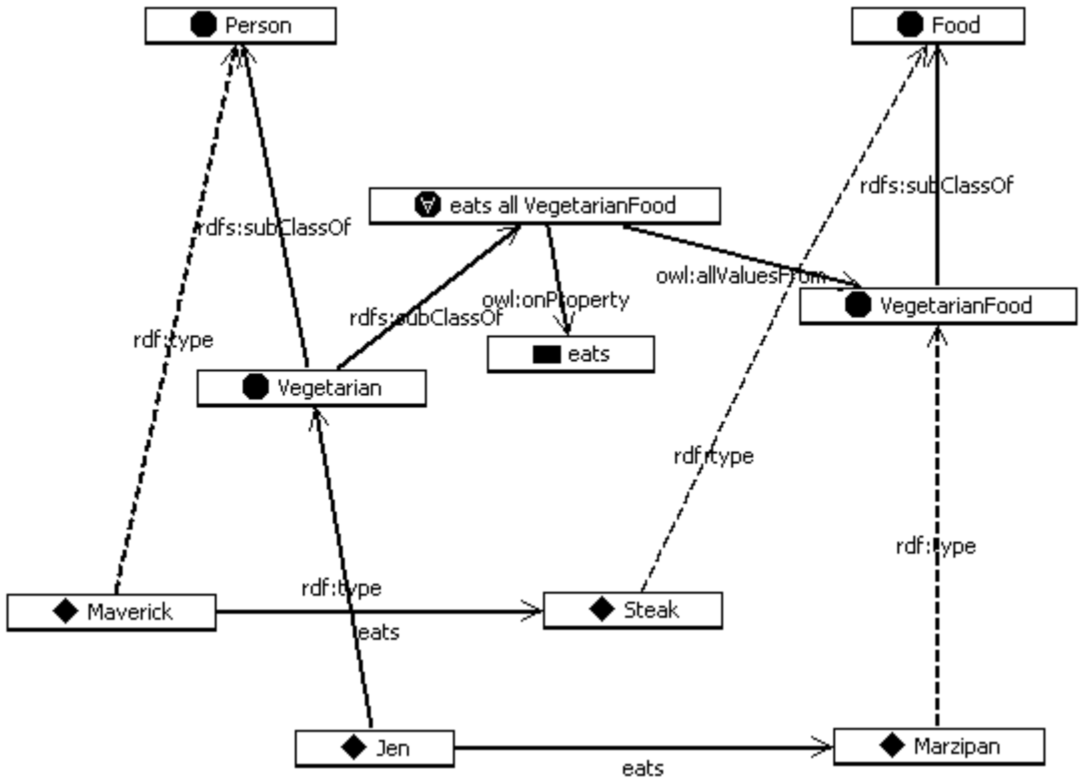


Figure 10-9. Definition of a Vegetarian as a restriction on what they eat.

10.2.2 Challenge: Filtering data based on explicit type

In section 3.2, we saw how tabular data can be used in RDF by considering each row to be an individual, the column names as properties, and the values in the table as values. We saw sample data in table 3.10, which we repeat here as Table 10-1.

Product						
ID	**Model No.**	**Division**	**Product Line**	**Manufacture location**	**SKU**	**Available**
1	ZX-3	Manufacturing support	Paper machine	Sacramento	FB3524	23
2	ZX-3P	Manufacturing support	Paper machine	Sacramento	KD5243	4
3	ZX-3S	Manufacturing support	Paper machine	Sacramento	IL4028	34
4	B-1430	Control Engineering	Feedback Line	Elizabeth	KS4520	23
5	B-1430X	Control Engineering	Feedback Line	Elizabeth	CL5934	14
6	B-1431	Control Engineering	Active Sensor	Seoul	KK3945	0
7	DBB-12	Accessories	Monitor	Hong Kong	ND5520	100
8	SP-1234	Safety	Safety Valve	Cleveland	HI4554	4
9	SPX-1234	Safety	Safety Valve	Cleveland	OP5333	14

Table 10-1 Typical tabular data for RDF import (cf. table 3-10)

Some sample triples from this data (also from section 3.2) include:

Subject	**Predicate**	**Object**
man:Product1	rdf:type	man:Product
man:Product1	man:Product_ID	1
man:Product1	man:Product_ModelNo	ZX-3
man:Product1	man:Product_Division	Manufacturing support
man:Product1	man:Product_Product_Line	Paper Machine
man:Product1	man:Product_Manufacture_Location	Sacramento
man:Product1	man:Product_SKU	FB3524
man:Product1	man:Proudct_Available	23
man:Product2	rdf:type	man:Product
man:Product2	man:Product_ID	2
man:Product2	man:Product_ModelNo	ZX-3P
man:Product2	man:Product_Division	Manufacturing support
man:Product2	man:Product_Product_Line	Paper Machine
man:Product2	man:Product_Manufacture_Location	Sacramento
man:Product2	man:Product_SKU	KD5243
man:Product2	man:Proudct_Available	4

man:Product3	rdf:type	man:Product
man:Product4	rdf:type	man:Product
man:Product5	rdf:type	man:Product
		…

Table 10-2. Sample triples from table 10-1 (cf. table 3.10)

Each row from the original table appears as an individual in the RDF version. Each of these individuals has the same type, namely *man:Product*, from the name of the table. This data includes only a limited number of possible values for the "Product_Line" field, and they are known in advance (e.g., "Paper machine", "Feedback line", "Safety Valve", etc.).

A more elaborate way to import this information would be to still have one individual per row in the original table, but to have rows with different types depending on the value of the Product Line column. For example, the following triples (among others) would be imported:

```
man:Product1 rdf:type ns:Paper_machine .
man:Product4 rdf:type ns:Feedback_line .
man:Product7 rdf:type ns:Monitor .
man:Product9 rdf:type ns:SafetyValve .
```

This is a common situation when actually importing information from a table. It is quite common for type information to appear as a particular column in the table. If we use a single method for importing tables, all the rows become individuals of the same type. A software intensive solution would be to write a more elaborate import mechanism that allows a user to specify which column should specify the type. A model-based solution would use a model in OWL, and an inference engine, to solve the same problem.

CHALLENGE

Build a model in OWL so that we can infer the type information for each individual, based on the value in the "Product Line" field, but using just the simple imported triples described in chapter 3.

SOLUTION.

Since the classes that the rows will be members of (i.e., the product lines) are already known, we need to describe those classes:

```
ns:Paper_Machine rdf:type owl:Class .
ns:Feedback_Line rdf:type owl:Class .
ns:Active_Sensor rdf:type owl:Class .
ns:Monitor rdf:type owl:Class .
ns:Safety_Valve rdf:type owl:Class .
```

Each of these classes must include just those individuals with the appropriate value for the property *man:Product_Product_Line*. The class constructor that achieves this uses an *owl:hasValue* restriction, as follows:

```
ns:Paper_Machine owl:equivalentClass
    [ a owl:Restriction;
        owl:onProperty man:Product_Product_Line
        owl:hasValue "Paper machine"] .

ns:Feedback_Line owl:equivalentClass
    [ a owl:Restriction;
        owl:onProperty man:Product_Product_Line
        owl:hasValue "Feedback line"] .

ns:Active_Sensor owl:equivalentClass
    [ a owl:Restriction;
        owl:onProperty man:Product_Product_Line
        owl:hasValue "Active sensor"] .

ns:Monitor owl:equivalentClass
    [ a owl:Restriction;
        owl:onProperty man:Product_Product_Line
        owl:hasValue "Monitor"] .

ns:Safety_Valve owl:equivalentClass
    [ a owl:Restriction;
        owl:onProperty man:Product_Product_Line
        owl:hasValue "Safety Valve"] .
```

Each of these definitions draws inferences as desired. Consider *man:Product1* ("ZX-3"), for which the triple

```
man:Product1 man:Product_Product_Line "Paper machine" .
```

has been imported from the table. The first triple ensures that *man:Product1* satisfies the conditions of the restriction for *Paper_Machine*. Hence,

```
man:Product1 rdf:type [ a owl:Restriction;
       owl:onProperty man:Product_Product_Line
       owl:hasValue "Paper machine" ] .
```

can be inferred. But since this restriction is equivalent to the definition for *man:Paper_Machine*, we have

```
man:Product1 rdf:type man:Paper_Machine .
```

as desired.

Furthermore, this definition maintains coherence of the data, even if it came from a different source, other than the imported table. Suppose that a new product is defined according to the following RDF:

```
os:ProductA rdf:type man:Paper_Machine .
```

The semantics of *owl:equivalentClass* means that all member of man:Paper_Machine are also members of the restriction. In particular,

```
os:ProductA rdf:type [ a owl:Restriction;
       owl:onProperty man:Product_Product_Line
       owl:hasValue "Paper Machine" ] .
```

Finally, because of the semantics of the restriction, we can infer

```
os:ProductA man:Product_Product_Line "Paper Machine" .
```

The end result of this construct is that regardless of how product information is brought into the system, it is represented both in terms of *rdf:type* and *man:Product_Product_Line* consistently.

10.2.3 Challenge: Relationship transfer in SKOS

When mapping from one model to another, or even when specifying how one part of a model relates to another, it is not uncommon to make a statement of the form, "everything related to A by property p should also be related to B, but by property q." For example, "everyone who plays for the All Star team is governed by the league's contract," or "every work in the Collected Works of Shakespeare was written by Shakespeare." We refer to this kind of mapping as *relationship transfer*, since it involves transferring individuals in a relationship with one entity to another relationship with another entity.

In Chapter 8XXX, we saw how SKOS provides a framework for describing knowledge organization systems like thesauri, taxonomies, and controlled vocabularies. Not surprisingly, the issue of relationship transfer appears in this system, as well. In Section 8.2.2XXX, we saw a special-purpose rule for managing collections, namely

If we have triples of the form

```
X skos:narrower C .
C skos:member Y .
```

Then we can infer the triple

```
X skos:narrower Y .
```

In the case in which a collection C is narrower than a concept X, we can say that "Every member of C is also narrower than X." That is, the rule that governs the treatment of *skos:narrower* in the context of a *skos:Collection* is a relationship transfer.

CHALLENGE

Represent the SKOS rule for propagating *skos:narrower* in the context of a *skos:Collection* using constructs in OWL. For example, represent the constraint

```
IF agro:MilkBySourceAnimal skos:member Y .
THEN agro:Milk skos:narrower Y .
```

in OWL.

SOLUTION

First, let's define an inverse for *skos:member*

```
skos:isMemberOf owl:inverseOf skos:member .
```

We already have an inverse for *skos:narrower* which is *skos:broader*.

With these inverses, we can re-state the problem as

```
IF Y skos:isMemberOf agro:MilkBySourceAnimal .
THEN Y skos:broader agro:Milk .
```

How do we specify, in OWL, the set of all things Y that are members of *agro:MilkBySourceAnimal*? We can use an *owl:hasValue* restriction for that.

```
agro:MembersOfMilkSource owl:equivalentClass
    [ a owl:Restriction ;
      owl:onProperty skos:isMemberOf ;
      owl:hasValue agro:MilkBySourceAnimal ] .
```

We can also describe the set of all things that have *agro:Milk* as a broader category. We will call it *agro:NarrowerThanMilk*, since these things are narrower than *Milk* (i.e., *Milk* is broader than they are)

```
agro:NarrowerThanMilk owl:equivalentClass
   [ a owl:Restriction ;
     owl:onProperty skos:broader ;
     owl:hasValue agro:Milk ] .
```

Now, to say that all members of one of these classes is in the other, we simply use *rdfs:subClassOf*, thus:

```
ex:MembersOfMilkSource rdfs:subClassOf agro:NarrowerThanMilk .
```

You can think of *rdfs:subClassOf* as something like an IF/THEN relationship; IF something is a member of the subclass, THEN it is a member of the superclass. In this case, both the subclass and the superclass are restrictions; when this happens, the IF/THEN takes on more meaning. In this case, it takes on the meaning IF an individual *skos:isMemberOf agro:MilkBySourceAnimal*, then that individual(has) *skos:broader* (concept) *agro:Milk*. With the inverses as defined above, this is equivalent to saying

IF

```
agro:MilkBySourceAnimal skos:member X
```

THEN

```
agro:Milk skos:narrower X
```

as desired.

10.2.4 Relationship transfer in FOAF

A similar situation arises in FOAF with groups of people (section 8.3.5XXX). Recall that FOAF provides two ways to describe members of a group. First, via the *foaf:member* relation, that relates an individual member G of *foaf:Group* to the individuals who are in that group. Second, that same group G is related to an *owl:Class* by *the foaf:membershipClass* property. We take an example from the life of Shakespeare to illustrate this.

Suppose we define a *foaf:Group* for Shakespeare's Children, as follows:

```
b:Shakespeares_Children
      a foaf:Group ;
      foaf:name "Shakespeare's Children” ;
      foaf:member b:Susanna , b:Judith , b:Hamnet ;
      foaf:membershipClass b:ChildOfShakespeare .

b:ChildOfShakespeare a owl:Class .
```

The FOAF standard specifies that the following rule should hold:

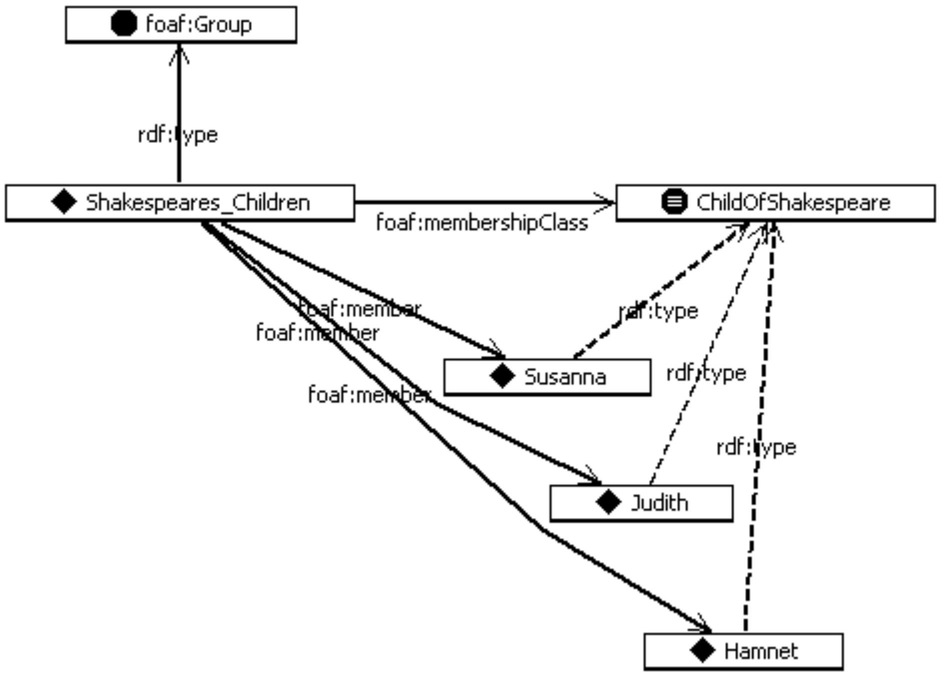
IF

```
 b:Shakespeares_Children foaf:member ?x
```

THEN

```
?x rdfs:type b:ChildOfShakespeare .
```

Figure 10-10 shows graphically the result of this rule in the case of Shakespeare's family; fine lines represent asserted triples; the three bold lines represent the triples that are to be inferred.



10-10 Inferences based on membershipClass in FOAF

CHALLENGE

How can we get the inferences shown in Figure 10-10 by using only the constructs from OWL (i.e., without special-purpose rules)?

SOLUTION

The solution parallels the solution for relationship transfer in SKOS, but in this case, the relationship we are transferring to is *rdf:type*. We begin as we did in that example, by defining an inverse of *foaf:member*:

```
b:memberOf owl:inverseOf foaf:member .
```

Now we can define *ChildOfShakespeare* to be (equivalent to) the class of all individuals that are *b:memberOf b:Shakespeares_Children* using an *owl:hasValue* restriction:

```
b:ChildOfShakespeare
      a owl:Class ;
      rdfs:label "Child of Shakespeare”;
      owl:equivalentClass
              [ a owl:Restriction ;
                owl:hasValue b:Shakespeares_Children ;
                owl:onProperty b:memberOf
              ] .
```

Let’s follow the progression of Shakespeare’s children through this inference. From Figure 10-10, we begin with three triples:

```
b:Shakespeares_Children foaf:member b:Hamnet .
b:Shakespeares_Children foaf:member b:Judith .
b:Shakespeares_Children foaf:member b:Susanna .
```

By the semantics of *owl:inverseOf*, we can infer

```
b:Hamnet foaf:memberOf b:Shakespeares_Children .
b:Judith foaf:memberOf b:Shakespeares_Children .
b:Susanna foaf:memberOf b:Shakespeares_Children .
```

Therefore all three are also members of the restriction defined above, and hence we can conclude that

```
b:Hamnet rdf:type b:ChildOfShakespeare .
b:Judith rdf:type b:ChildOfShakespeare .
b:Susanna rdf:type b:ChildOfShakespeare .
```

Following similar reasoning, we can also turn this inference around backwards; if we instead assert that

```
b:Hamnet rdf:type b:ChildOfShakespeare .
b:Judith rdf:type b:ChildOfShakespeare .
b:Susanna rdf:type b:ChildOfShakespeare .
```

We can infer that

```
b:Shakespeares_Children foaf:member b:Hamnet .
b:Shakespeares_Children foaf:member b:Judith .
b:Shakespeares_Children foaf:member b:Susanna .
```

10.2.5 Discussion

Just because we can represent something in OWL, doesn't necessarily mean that we should do so. How do the solutions in sections 10.2.3 and 10.2.4 compare to the ones that were actually taken by the SKOS and FOAF standards?

As we have seen, the SKOS standard uses a special-purpose rule to define the meaning of *skos:narrower* in the context of a *skos:Concept* and a *skos:Collection*. This means that a SKOS user can express the relationship between *Milk* and *MilkBySourceAnimal* simply by asserting the triple

```
agro:Milk skos:narrower agro:MilkBySourceAnimal .
```

Then the rule takes care of the rest. This is certainly much simpler for the SKOS user, than to construct the pair of restrictions shown in section 10.2.3.

This advantage for the rule-based approach goes even further; SKOS in fact defines the rule with a bit more generality, as follows:

```
X P C .
P rdf:type skos:CollectableProperty .
C skos:member Y .
```

Then we can infer the triple

X P Y.

That is, this rule works for any *skos:CollectableProperty*, which includes *skos:narrower*, *skos:broader*, and *skos:related*. A single rule can express the constraints for three different properties. To do the same using the OWL relationship transfer pattern, you would have to repeat the pattern once for each property, and for each concept/collection pair that you want to specify the relationship for. Seen from this point of view, the rule seems like a far superior solution.

On the other hand, writing a special-purpose rule into the standard has its own drawbacks. What language should the rules be written in? What processor will process the rules? The pragmatic answer to this is that the rules are written in the natural language that the standard specification is written in, and the processing will be done by each application, rather than by a general purpose inference engine. This has the drawback that every application has to implement the rule again. In contrast, the OWL solution (and OWL solution) can make use of generic software, and takes advantage of standard semantics.

For better or worse, the SKOS standard has chosen to express this rule as part of the standard, leaving its implementation to each application.

The situation for FOAF is a bit different. Unlike the situation for SKOS, there is only one property (*foaf:membershipClass*) that is affected by the transfer rule. Furthermore, a FOAF user has to assert the triple

```
b:Shakespeares_Children foaf:membershipClass b:ChildOfShakespeare .
```

in order for the transfer rule to come in to play (in contrast to SKOS, this isn't built-in to some other construct like *skos:Collection*). That is, the FOAF user is already explicitly indicating at what point the rule is to be invoked. Furthermore, the ground-up evolutionary strategy of FOAF argues against putting special-purpose meanings into the standard, since there is a good chance that these things could be changed or superseded by future versions. As it stands, any FOAF user can already express (in OWL) the relationship between a *foaf:Group* and its *foaf:members*, or indeed any other class and property as needed or desired. This is quite in accord with the AAA slogan, and in particular with the ground-up empowerment of the FOAF user community that is manifest in the rest of the FOAF project.

Since SKOS is on track to become a W3C recommendation, it is possible to put a few rules into its specification, and keep some control on how the rules interact. Furthermore, SKOS is not domain-specific; it is intended to be usable across several domains. As such, SKOS must anticipate that any number of concept/collection pairs might require this rule.

When modeling in a more vertical domain, some of these conditions may not hold. Certainly it is not common for most modelers to be seeking W3C recommendation status, which means that any rules that are put into the model can have possible adverse interactions with other rules. Furthermore, it is not uncommon when modeling a particular vertical domain to find that there are a few very distinguished instances in which some part of the model needs to be replicated at another place; the examples given above about the "Complete Works of Shakespeare" and "All Star Team" are such examples. In these cases, the relationship transfer is part of the description of these concepts, and is not something that needs to be repeated an indefinite number of times. In such cases, it may be just as convenient to describe the relationships using constructs in OWL. This seems to be the case for group membership in OWL; the modeler is making a very special statement about a group when they relate it to its *membershipClass*; it is not out of the question to have a somewhat involved way to express this relationship, especially if it can be done without cluttering up the FOAF standard itself.

A final comment about the comparative practice of expressing rules as part of a standards document vs. an explicit representation in a semantic model has to do with the very nature of modeling as an intellectual pursuit. One reason to model knowledge about a domain in the first place is to understand the ramifications of that model; to understand when there are conflicts between one viewpoint of the world and another. When rules are represented as part of a practice (e.g., encoded into a standard), the rules are not themselves available for automated analysis. That is, suppose that a rule in FOAF were to have some adverse interaction with a rule from SKOS. How would we know not to use these two standards together? In the next chapter, we introduce notions of inconsistency and contradiction, and see how representations that remain within the OWL standard can detect such interactions in advance of their application to any actual web data.

10.3 Alternative characterizations of OWL

In this book, we characterize OWL and its semantics with respect to the standard interpretation of OWL as RDF triples. Other characterizations have been used during the history of OWL, and even appear in user interfaces of some tools. Each characterization uses its own vocabulary to describe exactly the same things. In this section we will review some of the most common ways that you will encounter for talking about OWL restrictions and classes, and provide a recommendation for best practice terminology.

The semantics of *rdfs:subClassOf* and *owl:equivalentClass* are quite easy to characterize in terms of the inferences that hold:

```
X rdfs:subClassOf Y .
```

can be understood as a simple IF/THEN relation; if something is a member of X, then it is a member of Y.

```
X owl:equivalentClass Y .
```

can be understood as two IF/THEN relations, one going each way; if something is a member of X, then it is also a member of Y, and vice versa.

These relations remain unchanged in the case where X and/or Y are restrictions. We can see these relationships with examples taken from the solar system:

Consider two classes; one is a named class *SolarBody*; we'll call this class 'A' for this discussion. The other is a restriction *onProperty orbits hasValue TheSun*. We'll call this class 'B'.

We can say that all solar bodies orbit the sun by asserting

```
A rdfs:subClassOf B .
```

This is, if something is a solar body, then it orbits the sun. This situation is sometimes characterized by saying that "orbiting the sun is a *necessary condition* for *SolarBody*". The intuition behind this description is that if you know that something is a *SolarBody*, then it is necessarily the case that it orbits the sun. Since such a description of the class *SolarBody* describes the class, but does not provide a complete characterization of it (that is, you cannot determine from this description that something is a member of

SolarBody), this situation is also sometimes denoted by saying that "orbiting the sun is a *partial* definition for the class *SolarBody*".

If, on the other hand, we say that solar bodies are the same as the set of things that orbit the sun, we can express this in OWL compactly as

```
A owl:equivalentClass B .
```

Now we can make inferences in both directions; if something orbits the sun, then it is a *SolarBody*; but also, if it is a *SolarBody* , then it orbits the sun. This situation is sometimes characterized by saying that "orbiting the sun is a *necessary and sufficient condition* for *SolarBody*". The intuition behind this description is that if you that something is a *SolarBody*, then it is necessarily the case that it orbits the sun. But furthermore, if you want to determine that something is a *SolarBody*, it is sufficient to establish that it orbits the sun. Furthermore, since such a description does fully characterize the class *SolarBody*, this situation is also sometimes denoted by saying that "orbiting the sun is a *complete* definition for the class *SolarBody*".

Finally, if we say that all things that orbit the sun are solar bodies, we can express this compactly in OWL as

```
B rdfs:subClassOf A .
```

That is, if something orbits the sun, then it is a *SolarBody*. Given the usage of the words *necessary* and *sufficient* above, one could be excused for believing that in this situation one would say that "orbiting the sun is a *sufficient condition* for *SolarBody.*" However, it is not common practice to use the word *sufficient* in this way. Despite the obvious utility of such a statement from a modeling perspective and its simplicity in terms of OWL (it is no more complex than *partial* or *complete* definitions), there is no term corresponding to *partial* or *complete* for this situation.

Because of the incomplete and inconsistent ways in which words like *partial*, *complete*, *sufficient* and *necessary* have been traditionally used to describe OWL, we strongly discourage their use, in favor of the simpler and consistent use of the OWL terms *rdfs:subClassOf* and *owl:equivalentClass*.

10.4 Fundamental Concepts

The following fundamental concepts were introduced in this chapter:

owl:Restriction: The building block in OWL that describes classes by restricting the values allowed for certain properties.

owl:hasValue: a type of restriction that refers to a single value for a property.

owl:someValuesFrom: a type of restriction that refers to a set from which some value for a property must come.

owl:allValuesFrom: a type of restriction that refers to a set from which all values for a property must come.

owl:onProperty: link from a restriction to the property it restricts.