6 RDF Schema language

If Semantic Web modeling in RDF is about graphs, Semantic Web modeling in the RDF Schema language (RDFS) is about sets. Some aspects of set membership can be modeled in RDF alone, as we have seen with the *rdf:type* built-in property. But RDF itself simply creates a graph structure to represent data. RDFS provides some guidelines about how to use this graph structure in a disciplined way. RDFS provides a way to talk about the vocabulary that will be used in an RDF graph. Which individuals are related to one another, and how? How are the properties we use to define our individuals related to sets of individuals, and indeed to one another? RDFS provides a way for an information modeler to express the answers to these sorts of questions, as they pertain to particular data modeling and integration needs.

As such, RDFS is like other schema language – it provides information about the ways in which we describe our data. But RDFS differs from other schema languages in important ways.

6.1 Schema Languages and their functions

RDFS is the schema language for RDF. But what is a schema language in the first place? There are a number of successful schema languages for familiar technologies, but the role that each of these languages plays in the management of information is closely tied to the particular language or system.

Let's consider document modeling systems as an example. For such a system, a schema language allows one to express the set of allowed formats for a document. For a given schema, it is possible to determine (often automatically) whether a particular

document conforms to that schema. This is the major capability provided by XML Schema definitions. XML parsers can automatically determine whether a particular XML document conforms to a given schema.

Other schema languages help us to interpret particular data. For example, a database schema provides header and key information for tables in a relational database. There is nothing in the table itself to indicate the meaning of the information in a particular column, nor to indicate which column is to be used as an index for the table. This information is appropriately included in the database schema, since it does not change from one data record to the next.

For Object-oriented programming systems, the class structure plays an organizing role for information as well. But in object-oriented programming, the class diagram does more than describe data. It determines, according to the inheritance policy of the particular language, what methods are available for a particular instance, and how they are implemented. This stands in stark contrast to relational databases and XML, in that it does not interpret information, but instead provides a systematic way for someone to describe information and available transformations for that information.

Given this variety of understandings of how schema information can be used in different modeling paradigms, one could come to wonder whether calling something a schema language actually tells us anything at all! But there is something in common among all these notions of a schema. In all cases, the schema tells us something about the information that is expressed in the system. The schema is information about the data.

How then can we understand the notion of schema in RDF? What might we want to say about RDF data? And how might we want to say it? They key idea of the schema in

RDF is that it should help provide some sense of *meaning* to the data. The mechanism by which it does this is again the concept of *inference*.

The basic idea of inference is that it is possible to know more about a set of data than what is explicitly expressed in the data. By making this extra information explicit, we explicate (in a small way) the meaning of the original data. The additional information is based in a systematic way on patterns in the original data. This idea is not completely unfamiliar for schema languages. In the case of XML Schema, the validation process provides more than just a yes/no answer, but also provides type information for the parsed data. RDFS builds upon and formalizes this idea by providing detailed axioms that express exactly what inferences can be drawn from particular data patterns.

In most modeling systems, there is a clear division between the data and its schema. The schema for a relational database is not typically expressed in a table in the database; the object model of an object-oriented system is not expressed as objects, and an XML DTD is not a valid XML document. But in many cases, modern versions of such systems do model the schema in the same form as the data; the meta-object protocol of Common Lisp and the introspection API of Java represent the object models as objects themselves. The XML Stylesheet Definition defines XML Styles in an XML language.

In the case of RDF, the schema language was defined in RDF from the very beginning. That is, all schema information in RDFS is defined with RDF triples. The relationship between "plain" resources in RDF and schema resources is made with triples, just like relationships between any other resources. This elegance of design makes it particularly easy to provide a formal description of the semantics of RDFS, simply by providing inference rules that work over patterns of triples. While this is good

engineering practice (in some sense, the RDF standards committee learned a lesson from the issues that the XML standards had with DTDs), its significance goes well beyond its value as good engineering. In RDF, everything is expressed as triples. The meaning of asserted triples is expressed in new (inferred) triples. The structures that drive these inferences, that describe the meaning of our data, are also in triples. This means that this process can continue as far as it needs to; the schema information that provides context for information on the Semantic Web can itself be distributed on the Semantic Web.

We can see this in action by showing how a set is defined in RDFS. The basic construct for specifying a set in RDFS is called an *rdfs:Class*. Since RDFS is expressed in RDF, the way in which we express that something is a class is with a triple. In particular, a triple in which the predicate is *rdf:type*, and the object is *rdfs:Class*. Here are some examples that we will use in the following discussion:

```
AllStarPlayer rdf:type rdfs:Class .
MajorLeaguePlayer rdf:type rdfs:Class .
Surgeon rdf:type rdfs:Class .
Staff rdf:type rdfs:Class .
Physician rdf:type rdfs:Class .
```

These are triples in RDF just like any other; the only way that we know that they refer to the schema, rather than the data, is because of the use of the term in the ***rdfs:*** namespace, *rdfs:Class*. But what is new here? In Chapter **Error! Reference source not found.**, we already had the notion of *rdf:type*, which we used to specify that something was a member of a set. What do we gain, by specifying explicitly that something is a set? What we gain is a description of the meaning of membership in a set. In RDF, the only 'meaning' we had for set membership was given by the results of some query*; rdf:type* actually didn't behave any differently from any other (even user-defined) property. How

can we specify what we *mean* by set membership? In RDFS, we express meaning through the mechanism of inference.

6.2 What does it mean? Semantics as inference

RDFS "extends" RDF by introducing a set of distinguished resources into the language. This is similar to the way in which a traditional programming language can be extended by defining new language-defined keywords. But there is an important difference: in RDF, we already had the capability to use any resource in any triple ("Anyone can say anything about anything"); so by identifying certain specific resources as "new keywords", we haven't actually extended the language at all! We have simply identified certain triples as having a special meaning, as defined by a standard.

How can we define the 'meaning' of a distinguished resource? In RDFS, meaning is expressed by specifying inferences that can be drawn when the resource is used in a certain way. Throughout the rest of this section, whenever we introduce a new resource in RDFS, we will answer the question "What does it mean?" with an answer of the form, "in these circumstances (defined by some pattern of triples), you can add (infer) the following new triples."

We will demonstrate this principle through a simple example of the meaning of one of the most fundamental terms in RDFS; *rdfs:subClassOf.*

6.2.1 Example. Type propagation through rdfs:subClassOf

In the previous chapter, we saw the type propagation rule as an example of inferencing in the Semantic Web. The type propagation rule applies when we use *rdf:type* and *rdfs:subClassOf* in a particular pattern. But how does this kind of inference tell us

anything about what it means to be a member of a class? In RDFS, membership in a class is only given meaning when there several (well, at least, more than one) classes involved, and some relation between those classes is known. What does it mean for something to be a member of a class? It means that it is also a member of any superclass. This meaning is expressed in RDFS in general through inferencing, and in particular, with the type propagation rule. The type propagation rule is just one of many rules in RDFS. When taken together, these rules provide a way to express a variety of relations between classes and properties. Such a collection of classes and properties can provide a rudimentary definition of a vocabulary for RDF data. That is, it defines set of elements and the relationship of those sets to the properties that describe the elements.

We begin with a simple example of the type propagation rule. We will combine this with other rules later on to show how *rdfs:subClassOf* forms the basis of a vocabulary definition.

Suppose we define a vocabulary that says that an *AllStarPlayer* is a *MajorLeaguePlayer*, and that *Kaneda* is an *AllStarPlayer*. Our ordinary understanding of how terminology works tells us that we should be able to infer that *Kaneda* is a *MajorLeaguePlayer*.

One of the challenges in vocabulary modeling in general (and RDFS in particular) is to differentiate the two uses of the words "is a" in the example. When we say that "AllStarPlayer is a MajorLeaguePlayer," we are relating two types to one another; but when we say "Kaneda is an AllStarPlayer," we are giving type information about a particular individual, that is, relating an individual to a type. We already know how to provide individual type information from RDF; in N3, this latter statement appears as

```
Kaneda rdf:type AllStarPlayer .
```

RDFS provides a new resource, *rdfs:subClassOf*, to express the "is a" relation between types. In N3, our statement about types of players appears as

```
AllStarPlayer rdfs:subClassOf MajorLeaguePlayer .
```

RDFS provides a meaning for *rdfs:subClassOf* in this situation, that states that we may infer that the type information for *Kaneda* propagates in the expected way:

```
Kaneda rdf:type MajorLeaguePlayer .
```

Inferred triples have the same status as triples that we asserted; that is, they can be used again in other rules, to produce more inferred triples.

In general, the pattern for *rdfs:subClassOf* states that if we have triples of the form

```
A rdfs:subClassOf  B .
r rdf:type A .
```

then we can infer

```
r rdf:type B .
```

This very simple interpretation of the subclass relationship makes it a workhorse for RDFS modeling (and also for OWL modeling, as described in subsequent chapters). It corresponds to a great degree to the IF/THEN construct of programming languages; IF something is a member of the subclass, THEN it is a member of the superclass. It should come as no surprise that this has a large number of modeling applications. In RDFS, there is no construct that corresponds to the ELSE clause that is present in most conventional programming languages; you can infer things from asserted membership of resources in classes, but you cannot infer things from the lack of asserted membership.

6.3 The RDFS Schema Language

There's a lot more to RDFS than just the type propagation rule of *rdfs:subClassOf*. RDFS consists of a small number of inference patterns, each one of which can provide type information for individuals in a variety of circumstances. We will go through these patterns in turn, showing examples of their use.

6.3.1 Relationship propagation through rdfs:subPropertyOf

We have just seen how RDFS provides a means by which classes can be related to one another by the subclass relationship, and the inferences that define the meaning of this construct. This allows a modeler using RDFS to describe (using the notion of sets) relationships among the elements that are described in a collection of RDF triples. But if we want to give meaning to our data, we need to do more than just talk about the elements; we need to talk about the properties that link them – the predicates of the triples. RDFS provides a simple mechanism for doing just that. The mechanism works in a way that is very similar to the type propagation rule. The mechanism is, of course, based on an inference pattern. The pattern is defined using the resource (keyword*)* *rdfs:subPropertyOf*.

The basic intuition behind the use of *rdfs:subPropertyOf* is that terminology includes verbs as well as nouns, and many of the same requirements for mapping nouns from one source to another will apply to relationships. Simple examples abound in ordinary parlance; the relationship brother is more specific than the relationship sibling; if someone is my brother, then he is also my sibling. This is formalized in RDFS using an inference rule for *rdfs:subPropertyOf* that is almost as simple as the one for *rdfs:subClassOf*.

For two properties P and R, we can assert that

```
P rdfs:subPropertyOf R .
```

What does this mean? As always, the meaning of this statement will be given in terms of the inferences that it allows to be drawn. In particular, these inferences are that whenever we have the triple

```
A P B .
```

We can infer the triple

```
A R B .
```

6.3.1.1 Example. Employment

A large firm engages a number of people in various capacities, and has a variety of ways to administer these relationships. Some people are directly employed by the firm, while others are contractors. Among these contractors, some of them are directly contracted to the company on a free-lance basis, others on a long-term retainer, and still others contract through an intermediate firm. All of these people could be said to work for the firm.

How can we model this situation in RDFS? First we need to consider the inferences we wish to be able to draw, under what circumstances. There are a number of relationships that can hold between a person and the firm; we can call them *contractsTo*, *freeLancesTo*, *indirectlyContractsTo*, *isEmployedBy* and *worksFor*.

If we assert any of these statements about some person, then we would like to infer that that person *worksFor* the firm. Furthermore, there are intermediate conclusions we

can draw; for instance, both a free-lancer and an indirect contractor contract to the firm, and indeed work for the firm.

All these relationships can be expressed in RDFS using the *rdfs:subPropertyOf* relation:

```
freeLancesTo rdfs:subPropertyOf contractsTo .
indirectlyContractsTo rdfs:subPropertyOf contractsTo .
isEmployedBy rdfs:subPropertyOf worksFor .
contractsTo rdfs:subPropertyOf worksFor .
```

The discussion will be easier to follow if we represent this as a diagram, where the arrows denote, *rdfs:subPropertyOf*.

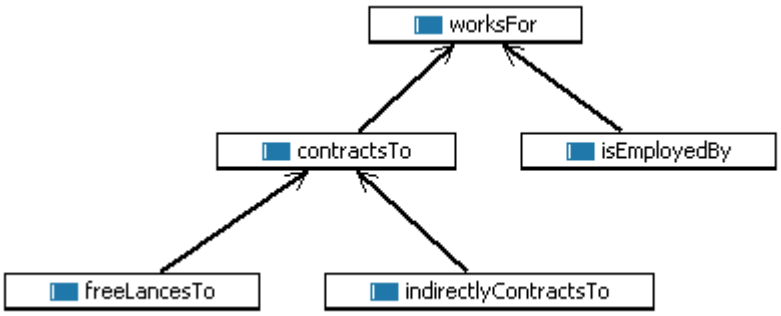

Figure 1. subPropertyRelations for workers in the firm.

In order to see what inferences can be drawn, we will need some instance data:

```
Goldman isEmployedBy TheFirm .
Spence freeLancesTo TheFirem .
Long indirectlyContractsTo TheFirm .
```

The rule that defines the meaning of *rdfs:subPropertyOf* implies a new triple, replacing any sub-property with its super-property. So, since

```
isEmployedBy rdfs:subPropertyOf worksFor .
```

we can infer that

```
Goldman worksFor TheFirm .
```

And because of the assertions about free-lancing and indirect contracts, we can infer that

```
Spence contractsTo TheFirm .
Long contractsTo TheFirm .
```

And finally, since inferred triples can be used, just as asserted triples, to make further inferences, we can further infer that

```
Spence worksFor TheFirm .
Long worksFor TheFirm .
```

In general, *rdfs:subPropertyOf* allows a modeler to describe a hierarchy of related properties. Just as in class hierarchies, specific properties are at the bottom of the tree, and more general properties are higher up in the tree. Whenever any property in the tree holds between two entities, so does every property above it.

The construct *rdfs:subPropertyOf* has no direct analog in object-oriented programming, where properties are not first-class entities (i.e., they cannot be related to one another, independent of the class in which they are defined). For this reason, unlike the case of *rdfs:subClassOf*, object-oriented programmers have no conflict with a similar known concept. The only source of confusion is that sub-property diagrams like the one above are sometimes mistaken for class diagrams.

6.3.2 Typing data by usage – rdfs:domain and rdfs:range

We have seen how inferences around *rdfs:subPropertyOf* can be used to describe how two properties relate to one another. But when we describe the usage of terms in our data, we would also like to represent how a property is used, relative to the defined classes. In particular, we might want to say that when a property is used, the subject of

the triple comes from (i.e., has *rdf:type*) a certain class, and that the object comes from some other type. These two stipulations are expressed in RDFS with the resources (keywords) *rdfs:domain* and *rdfs:range* respectively.

In mathematics, the words *domain* and *range* are used to refer to how a function (or more generally, a relation) can be used. The domain of a function is the set of values for which it is defined, and the range is the set of values it can take. In Real Analysis, for instance, the relation *squareroot* has the positive numbers as domain (since negative numbers don't have square roots in the reals), and all reals as the range (since there are both positive and negative square roots).

In RDFS, the properties *rdfs:domain* and *rdfs:range* have meanings inspired by the mathematical uses of these words. A property P can have an *rdfs:domain* and/or an *rdfs:range*. These are specified, as is everything in RDF, via triples:

```
P rdfs:domain D .
P rdfs:range R .
```

The informal interpretation of this is that the relation P relates values from the class D to values from the class R. D and R need not be disjoint, or even distinct.

The meaning of these terms is defined by the inferences that can be drawn from them. RDFS inferencing interprets domain with the inference rule:

```
IF
P rdfs:domain D .
  and
x P y .

THEN

x rdf:type D .
```

Similarly, range is defined with the rule

```
IF
P rdfs:range R .

 and

x P y .

THEN

y rdf:type R .
```

In RDFS, domain and range give some information about how the property P is to be used; *domain* refers to the subject of any triple that uses P as its predicate, and *range* refers to the object of any such triple. When we assert that property P has domain D (respectively, range R), we are saying that whenever we use the property P, we can infer that the subject (respectively object) of that triple is a member of the class D (respectively R). In short – domain and range tell us how P is to be used, but rather than signaling an error if P is used in a way that is apparently inconsistent with this declaration, instead RDFS will infer the necessary type information to bring P into compliance with its domain and range declarations. In RDFS, there is no way to assert that a particular individual is not a member of a particular class (contrast with OWL, section 4XXX). In fact, in RDFS, there is no notion of an incorrect or inconsistent inference. This means that, unlike the case of XML Schema, an RDF Schema will never proclaim an input as invalid; it will simply infer appropriate type information. In this way, RDFS behaves much more like a database schema, which declares what joins are possible, but makes no statement about the validity of the joined data.

6.3.3 Combination of domain and range with subClassOf

So far, we have seen inference patters for three resources in the ***rdfs*** namespace: *rdfs:domain*, *rdfs:range* and *rdfs:subClassOf* . We have seen how the inference patterns work on sample triples. But the inference patterns can also interact with one another in interesting ways. We can already see this happening with the three patterns we have seen so far. We will show the interaction between *rdfs:subClassOf* and *rdfs:domain* by starting with an example.

Suppose we have a very simple class tree that includes just two classes; *Woman* and *MarriedWoman* in the usual subclass relation:

```
:MarriedWoman rdfs:subClassOf :Woman .
```

Suppose we have a property called *maidenName*, whose domain is *MarriedWoman*:

```
:maidenName rdfs:domain :MarriedWoman .
```

In diagram form, this looks as follows:

Figure 2. Domain and subClassOf triples for hasMaidenName.

This unsurprising model holds some subtlety; let's examine closely what it says. If we assert the *maidenName* of anything (even if we don't know that it is a *Woman*!) the rule for *rdfs:domain* will allow us to infer that it is a *MarriedWoman*. So, for instance, if someone asserts

```
:Karen maidenName "Stephens" .
```

We can infer

```
:Karen rdf:type :MarriedWoman .
```

But we can make further inference based on the rdfs:*subClassOf* relationship between the classes, namely that

```
:Karen rdf:type :Woman .
```

There was nothing in this example that was particular to *Karen*; in fact, if we learn of any resource at all that it has a *maidenName*, then we will infer that it is a *Woman*. That is, we know that for any resource X, if we have a triple of the form

```
X :maidenName Y .
```

we can infer

```
X rdf:type :Woman .
```

But this is exactly the definition of *rdfs:domain*; that is, we have just proved that

```
:maidenName rdfs:domain :Woman .
```

This is a different way to use the definition of *rdfs:domain* than we have previously seen. So far, we have applied the inference pattern whenever a triple using *rdfs:domain* was asserted or inferred. Now, we are inferring an *rdfs:domain* triple whenever we can prove that the inference pattern holds. That is, we view the inference pattern as the definition of what it means for *rdfs:domain* to hold.

We can generalize this result to form a new inference pattern as follows:

```
IF
P rdfs:domain D .
  and
D rdfs:subClassOf C .

THEN

P rdfs:domain C .
```

That is, whenever we specify the *rdfs:domain* of a property to be some class, we can also infer that the property also has any superclass as *rdfs:domain*. The same conclusion holds for *rdfs:range*, using the same argument.

This simple definition of the meaning of *domain* and *range* is actually quite aggressive; we can draw conclusions about the type of any element based simply upon its use in a single triple, whenever we have *domain* or *range* information about the predicate. As we shall see in later examples, this can result in some surprising inferences. The definitions of domain and range in RDFS are the most common problem areas for modelers with experience in another data modeling paradigm. It is unusual to have such a strong interpretation for very common concepts.

The interaction between *rdfs:domain* and *rdfs:subClassOf* can seem particularly counterintuitive when viewed in comparison to Object-Oriented Programming. One of the basic mechanisms for organizing code in OOP is called *inheritance*. There are a number of different schemes for defining inheritance, but they typically work by propagating information *down* the class tree; that is, something (e.g., a method or a variable) that is defined at one class is also available at its subclasses.

There is a tendency for OO programmers, when they begin to work with RDFS, to expect inheritance to work in the same way. This tendency results from an "obvious" mapping from RDFS to OOP in which an *rdfs:Class* corresponds to a Class in OOP, a property in RDFS corresponds to a variable in OOP, and in which the assertion

```
P rdfs:domain C .
```

corresponds to the definition of the variable corresponding to *P* being defined at class C. From this “obvious” mapping comes an expectation that these definitions should inherit in the same way that variable definitions inherit in OOP.

But in RDFS, there is no notion of inheritance per se; the only mechanism at work in RDFS is inference. The inference rule in RDFS that most closely corresponds to the OO notion of inheritance is the subclass propagation rule from section 6.2.1; that the members of a subclass are also members of a class. The ramifications of this rule for instances correspond to what one would expect from inheritance; since an instance of a subclass is also an instance of the parent class, then anything we say about members of the parent class will necessarily hold for all instances of the subclass; this is consistent with usual notions of inheritance.

The interaction between *rdfs:domain* and *rdfs:subClassOf* , on the other hand, is more problematic. Using to the “obvious” interpretation, we asserted that the variable *hasMaidenName* was defined at *MarriedWoman*, and then inferred that it was defined at a class higher in the tree, namely *Woman*. Seen from an OO point of view, this interaction seems like inheritance *up* the tree, i.e., just the opposite of what is normally expected of inheritance in OOP.

The fallacy in this conclusion comes from the “obvious” mapping of rdfs:domain as defining a variable relative to a class. In the Semantic Web, because of the AAA slogan, a property can be used anywhere, and must be independent of any class. The property *maidenName* was, by design, always

available for any resource in the universe (including members of the class *Woman*); the assertion or inference of *rdfs:domain* made no change in that respect. That is, it is never accurate in the Semantic Web to say that a property is "defined for a class". A property is defined independently of any class.

6.4 RDFS Modeling Combinations and Patterns

The inference rules for RDFS are few in number and quite simple, nevertheless their effect can be quite subtle in the context of shared information in a semantic web. In this section, we outline a number of patterns of use of the basic RDFS features, illustrating each one with a simple example.

6.4.1 Set Intersection

It is not uncommon for someone modeling in RDFS to ask whether some familiar notions from logic are available. "Can I model set intersection in RDFS?" is a common question. The technically correct answer to this question is simply "no." There is no explicit modeling construct in RDFS for set intersection (nor for set union). However, when someone wants to model intersections (or unions), they don't always need to model them explicitly. They often only need certain particular inferences that are supported by these logical relations. Sometimes these inferences are indeed available in RDFS through particular design patterns that combine the familiar RDFS primitives in specific ways.

In the case of intersection n particular, one of the inferences someone might like to draw is that if a resource x is in C, then it is also in both A and B. Expressed formally,

the relationship they are expressing is that C ⊆ A ∩ B. This inference can be supported by making C a common subclass of both A and B, as follows:

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

How does this support an intersection-like conclusion? From the inference rule governing *rdfs:subClassOf*, it is evident that from the triple

```
x rdf:type C .
```

We can infer

```
x rdf:type B .
x rdf:type A .
```

as desired. Notice that we can only draw the inferences in one direction; from membership in C, we can infer membership in A and B. But from membership in A and B, we cannot infer membership in C. That is, we cannot express A ∩ B ⊆ C. This is the sense in which RDFS cannot actually express set intersection; it can only approximate it by supporting the inferencing in one direction.

6.4.1.1 Example: Hospital skills

Suppose we are describing the staff at a hospital. There are a number of different jobs and people who fill them, including nurses, doctors, surgeons, administrators, orderlies, volunteers, etc. A very specialized role in the hospital is the `Surgeon`. Among the things we know about surgeons, is that they are members of the hospital staff. They are also qualified physicians. Logically, we would say that Surgeon ⊆ Staff ∩ Physician, that is, *Surgeon* is a subset of those people who are both staff members and physicians.

Notice that we don't want to say that every staff physician is a surgeon, so the set inclusion goes only one way. From this statement, we want to be able to infer that if Kildare is a surgeon, then he is also a member of staff and he is a physician.

If we say

```
Surgeon rdfs:subClassOf Staff .
Surgeon rdfs:subClassOf Physician .
Kildare rdf:type Surgeon .
```

Then we can infer that

```
Kildare rdf:type Staff .
Kildare rdf:type Physician .
```

We cannot make the inference the other way; that is, if we were to assert that Kildare is a physician and member of staff, no RDFS rules are applicable, and no inferences are drawn. This is appropriate; consider the case in which Kildare is a psychiatrist. As such, he is both a member of staff and physician, but it is inappropriate to conclude that he must be a surgeon.

6.4.2 Property Intersection

In RDFS, properties are treated in a way analogous to the treatment of classes, and all the same operations and limitations apply. Even though it might seem unfamiliar to think of a property as a set, we can still use the set combination terms (intersection, union) to describe the functionality supported for properties. As was the case for Class intersections and unions, RDFS cannot express these things exactly, but it is possible to approximate these notions with judicious use of *subPropertyOf*.

One of the inferences we can express using *subPropertyOf* is that one property is an intersection of two others, $P \subseteq R \cap S$. That is, if we know that two resources x and y are related by property P,

```
x P y .
```

we want to be able to infer both

```
x R y .
x S y .
```

6.4.2.1 Example. Patients in hospital rooms

Suppose we are describing patients in a hospital. When a patient is assigned to a particular room, we can infer a number of things about the patient; we know that they are on the duty roster for that room, and that their insurance will be billed for that room. How do we express that both of these inferences come from the single assignment of a patient to a room?

```
lodgedIn rdfs:subPropertyOf billedFor .
logdedIn rdfs:subPropertyOf assignedTo .
```

Now if patient *Marcus* is *lodgedIn Room101*,

```
Marcus lodgedIn Room101 .
```

We can infer the billing and duty roster properties as well:

```
Marcus billedFor Room101 .
Marcus assignedTo Room101 .
```

Notice that we cannot make the inference in the other direction; that is, if we were to assert that *Marcus* is *billedFor Room101* and *assignedTo Room101*, no RDFS rules are applicable, and no inferences can be drawn.

6.4.3 Set Union

Using a pattern similar to the one we used for set intersection, we can also express certain things about set unions in RDFS. In particular, we can express that $A \cup B \subseteq C$. We do this by making C a common superclass of A and B, thus:

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

Any instance x that is a member of either A or B is inferred to be also a member of C; i.e.,

```
x rdf:type A .
   or
x rdf:type B .

implies

x rdf:type C .
```

6.4.3.1 Example. All-Stars.

In determining the candidates for a season's All Stars, the league rules state that they will select among all the players who have been named Most Valuable Player (*MVP*), as well as among those who have been top scorers (*TopScorer*) in their league. We can model this in RDFS by making *AllStarCandidate* a common superclass of *MVP* and *TopScorer* as follows:

```
MVP rdfs:subClassOf AllStarCandidate .
TopScorer rdfs:subClassOf AllStarCandidate .
```

Now, if we know that *Reilly* was named *MVP* and *Kaneda* was a *TopScorer*:

```
Reilly rdf:type MVP .
Kaneda rdf:type TopScorer .
```

We can infer that both of them are All Star candidates:

```
Reilly rdf:type AllStarCandidate .
Kaneda rdf:type AllStarCandidate .
```

as desired. Notice that as in the case of intersection, we can only draw the inference in one direction, that is, we can infer that AllStarCandidate ⊇ MVP ∪ TopScorer , but not the other way around.

In summary, we can use *rdfs:subClassOf* to represent statements about intersection and union as follows:

C ⊆ A ∩ B (by making C *rdfs:subClassOf* both A and B)
C ⊇ A ∪ B (by making both A and B *rdfs:subClassOf* C).

6.4.4 Property Union

One can use *rdfs:subPropertyOf* to combine properties from different sources, in a way that is analogous to the way in which *rdfs:subClassOf* can be used to combine classes as a union. If two different sources use properties P and Q in similar ways, then a single amalgamated property R can be defined with *rdfs:subPropertyOf* as follows:

```
P rdfs:subPropertyOf R .
Q rdfs:subPropertyOf R .
```

For any pair of resources x and y related by P or by Q

```
x P y .
```

or

```
x Q y .
```

we can infer that

```
x R y .
```

6.4.4.1 Example. Merging library records

Suppose one library has a table in which it keeps lists of patrons and the books they have borrowed. It uses a property called *borrows* to indicate that a patron has borrowed a book. Another library uses *checkedOut* to indicate the same relationship.

Just as in the case of classes, there are a number of ways to handle this situation. If we are sure that the two properties have exactly the same meaning, we can make one property equivalent to another with a creative use of *rdfs:subPropertyO*f as follows:

```
borrows rdfs:subPropertyOf checkedOut .
checkedOut rdfs:subPropertyOf borrows .
```

Then any relationship that is expressed by one library will be inferred to hold for the other. In such a case, both properties are essentially equivalent.

If we aren't sure that the two properties are used in exactly the same way, but we have an application that we do know wants to treat them as the same, then we use the Union pattern to create a common super-property of both, as follows:

```
borrows rdfs:subPropertyOf hasPossession .
checkedOut rdfs:subPropertyOf hasPossession .
```

Using these triples, all patrons and books from both libraries will be related by the property *hasPossession*, thus merging information from the two sources.

6.4.5 Property transfer

When modeling the relationship between information that comes from multiple sources, a common requirement is to state that if two entities are related by some relationship in one source, that the same entities should be related by a corresponding relationship in the other source.

This can be accomplished quite easily in RDFS with a single triple. That is, if we have a property P in one source and property Q in another source, and we wish to state that all uses of P should be considered as uses of Q, we can simply assert that

```
P rdfs:subPropertyOf Q .
```

Now, if we have any triple of the form

```
X P Y .
```

We can infer that

```
X Q Y .
```

It may seem strange to have a design pattern that consists of a single triple; but this use of *rdfs:subPropertyOf* is so pervasive that it really merits being called out as a pattern in its own right.

6.4.5.1 Example: Terminology reconciliation

There are a growing number of standard information representation schemes being published in RDFS form. Information that has been developed in advance of these standards (or in a silo away from them) needs to be re-targeted to be compliant with the standard. This process can involve a costly and error-prone search-and-replace process through all the data sources. When the data is represented in RDF, there is often an easier option available, using the Property Transfer pattern.

As a particular example, the *Dublin Core* is a set of standard attributes used to describe bibliographic information for library systems. One of the most frequently used Dublin Core terms is *dc:creator*, which indicates an individual (person or organization) that is responsible for having created a published artifact.

Suppose that a particular legacy bibliography system uses the term *author* to denote the person who created a book. This has worked fine for this system, because it was not intended to classify books that were created without an author, e.g., compilations (which instead have an *editor*).

How can we make this data conformant to the Dublin Core, without performing a costly and error-prone process to copy-and-replace *author* with *dc:creator*?

This can be achieve in RDFS with the single triple

```
author rdfs:subPropertyOf dc:creator .
```

Now any individual for which the author property has been defined will now have the same value defined for the (standard) *dc:creator* property. The work is done by the RDFS inference engine, instead of by an off-line editing process. In particular, this means that legacy applications that are using the *author* property can continue to operate without modification, while newer, Dublin Core compliant applications can use the inferred data to operate in a standard fashion.

6.5 Challenges

Each of the patterns given above demonstrates the utility of combining one or more RDFS constructs to achieve a particular modeling goal. In this section, we outline a number of modeling scenarios that can be addressed with these patterns, and show how they can be applied to address these challenges.

6.5.1 Term reconciliation

One of the most common challenges in terminology management is the resolution of terms used by different agents who want to use their descriptions together in a single federated application. For example, suppose that one agent uses the word "analyst", while another uses the word "researcher". There are a number of relationships that can hold between these two usages; we will examine a number of common relations as a series of challenges.

CHALLENGE

How do we then enforce the assertion that any member of the one class will automatically be treated as a member of the other?

There are a number of approaches to this situation, depending on the details of the situation. All of them can be implemented using the patterns we have identified so far.

SOLUTION

Let's first take the case in which we determine that a particular term in one vocabulary is fully subsumed by a term in another, for example, we determine that a *researcher* is actually a special case of an *analyst*. How can we represent this fact in RDFS?

First, we examine the inferences that we want RDFS to draw, given this information. If a researcher is a special case of an analyst, then all researchers are also analysts. We can express this sort of "IF/THEN" relationship with a single *rdfs:subClassOf* relationship, thus:

```
Researcher rdfs:subClassOf Analyst .
```

Now any resource that is a Researcher, e.g.,

```
Wenger rdf:type Researcher .
```

will be inferred to be an analyst as well:

```
Wenger rdf:type Analyst .
```

If the relationship happens to go the other way around (that is, all analysts are researchers), the *rdfs:subClassOf* triple can be reversed accordingly.

CHALLENGE

What if the relationship is more subtle; suppose that there is considerable semantic overlap between the two concepts "analyst" and "researcher," but neither concept is defined in a sharp, formal way. It seems that there could be some analysts who are not researchers, and also vice-versa. Nevertheless, for the purposes of the federated application, we want to treat these two entities as the same. What can we do?

SOLUTION

In such a case, we can use the Union pattern outlined above. We can define a new term (for the federated domain) that is not defined in either of the sources, e.g., "investigator." Then we effectively define *investigator* as the union of *researcher* and *analyst*, using the common superproperty idiom:

```
Analyst rdfs:subClassOf Investigator .
Researcher rdfs:subClassOf Investigator .
```

Described this way, we have made no commitment to a direct relationship between *analyst* and *researcher*, but we have provided a federated handle for speaking of the general class of these entities.

CHALLENGE

At the other extreme, suppose that we determine that the two classes really are identical in every way – that these two terms really are just two words for the same thing.

In terms of inference, we would like any member of one class to be a member of the other, and vice-versa.

SOLUTION

RDFS does not provide a primitive statement of class equivalence, but the same result can be achieved with creative use of *rdfs:subClassOf*:

```
Analyst rdfs:subClassOf Researcher .
Researcher rdfs:subClassOf Analyst .
```

This may seem a bit paradoxical, especially to someone who is accustomed to object-oriented programming. But the conclusions based on RDFS inferencing are clear. For example, if we know that

```
Reilly rdf:type Researcher .
Kaneda rdf:type Analyst .
```

We can infer the other statements

```
Reilly rdf:type Analyst .
Kaneda rdf:type Researcher .
```

In effect, the two *rdfs:subClassOf* triples (or indeed, any cycle of *rdfs:subClassOf* triples) together assert the equivalence of the two classes.

6.5.2 Instance-level data integration

Suppose you have contributions to a single question coming from multiple sources. In the case where the question determines which instances are of interest, there is a simple way to integrate them, using *rdfs:subClassOf*. We will give an example from a military domain.

A command and control mission planner wants to determine where ordnance can be targeted, or more specifically, where it cannot be targeted. There are a number of

different sources of information that contributes to this decision. One source provides a list of targets and their types, some of which must never be targeted (civilian facilities like churches, schools, and hospitals). Another source provides descriptions of airspaces, some of which are off-limits (e.g., politically defined no-fly zones). A target is determined to be off-limits if it is excluded on the grounds of either of these data sources.

CHALLENGE

Bring together these data sources so that all of these sources (and any new ones that are subsequently discovered) are brought together into a single place.

SOLUTION

The solution is to use the Union construction to join together the two information sources into a single, federated class.

```
fc:CivilianFacility rdfs:subClassOf cc:OffLimits .
space:NoFlyZone rdfs:subClassOf cc:OffLimits .
```

Now any instance from either the facility descriptions or the airspace descriptions that have been identified as restricted will be inferred to have *cc:OffLimitsTarget*.

6.5.3 Readable labels with rdfs:label

Resources on the Semantic Web are specified by URIs, which provide a globally scoped unique identifier for the resource. But URIs are not particularly attractive or meaningful for human readers. RDFS provides a built-in property called *rdfs:label* whose intended use is to provide a printable name for any resource. This provides a standard way for presentation engines (e.g., web pages or desktop applications) to display the print name of a resource.

Depending on the source of the RDF data that is being displayed, there might be another source for human-readable names for any resource. One solution would be to change the display agent to use a particular display property for each resource. A simpler solution can be done entirely using the semantics of RDFS, through a combination of the property union and property transfer patterns.

Suppose we have imported RDF information from an external form, e.g., a database or spreadsheet. There are two classes of individuals defined by the import, *Person* and *Movie*. For *Person*, a property called *personName* is defined that gives the name by which that person is professionally known. For *Movie*, the property called *movieTitle* gives the title under which the movie was released. Some sample data from this import might be as follows:

```
Person1 personName "James Dean" .
Person2 personName "Elizabeth Taylor" .
Person3 personName "Rock Hudson" .
Movie1 movieTitle "Rebel Without a Cause" .
Movie2 movieTitle "Giant" .
Movie3 movieTitle "East of Eden" .
```

CHALLENGE

We would like to use a generic display mechanism, which uses the standard property *rdfs:label* to display information about these people and movies. How can we use RDFS to achieve this?

SOLUTION

The answer is to define each of these properties as sub-properties of rdfs:label as follows:

```
personName rdfs:subPropertyOf rdfs:label .
movieTitle rdfs:subPropertyOf rdfs:label .
```

When the presentation engine queries for *rdfs:label* of any resource, by the rules of RDFS inferencing, it will find the value of *personName* or *movieTitle*, depending on which one is defined for a particular individual. There is no need for the presentation engine to include any code that understands the (domain-specific) distinction between *Person* and *Movie*.

6.5.4 Data typing based on use

Suppose a shipping company has a fleet of vessels that it manages. The fleet includes new vessels that are under construction, vessels that are being repaired, vessels that are currently in service, and vessels that have been retired from service. The information that the company keeps about its ships might include the information in the following table:

Table *Vessel*

Name	Maiden Voyage	Next Departure	Decommission Date	Destruction date	Commander
Berengaria	June 16, 1913		1938		Johnson
QEII	May 2, 1969	Mar 4, 2010			Warwick
Titanic	April 10, 1912			April 14, 1912	Smith
Constitution	July 22, 1798	Jan 12, 2009			Preble

This information can be expressed in RDF triples in the manner outlined in Chapter 3. Each row corresponds to a resource of type *ship:Vessel*; triples express the information that appears in the body of the table, e.g.,

```
ship:Berengaria ship:maidenVoyage "Dec. 16, 1946" .
ship:QEII ship:nextDeparture "Mar 4, 2010" .
```

In addition to the class *ship:Vessel*, we can have subclasses that correspond to the status of the ships, e.g.

```
ship:DeployedVessel rdfs:subClassOf ship:Vessel .
ship:InServiceVessel rdfs:subClassOf ship:Vessel .
ship:OutOfServiceVessel rdfs:subClassOf ship:Vessel .
```

A *DeployedVessel* is one that has been deployed sometime in its lifetime. An *InServiceVessel* is one that is currently in service, while and *OutOfServiceVessel* is one that is currently out of service (for any reason, including retired ships and ships that have not been deployed).

CHALLENGE

How can we automatically classify each vessel into these more specific subclasses, depending on the information we have about it in the table? For instance, if a vessel has had a maiden voyage, then it is a *ship:DeployedVessel*. If its next departure is set, then it is an *ship:InServiceVessel*. If it has a decommission date or a destruction date, then it is an *ship:OutOfServiceVessel*.

SOLUTION

We can enforce these inferences using *rdfs:domain* as follows:

```
ship:maidenVoyage rdfs:domain ship:DeployedVessel .
ship:nextDeparture rdfs:domain ship:InServiceVessel .
ship:decommissionedDate rdfs:domain ship:OutOfServiceVessel .
ship:destructionDate rdfs:domain ship:OutOfServiceVessel .
```

The whole structure is shown in Figure 3. *Vessel* has three subclasses, *DeployedVessel*, *InServiceVessel* and *OutOfServiceVessel*. Each of these is in the domain of one or more of the properties *maidenVoyage*, *nextDeparture*, *decommissionedDate* and *destructionDate* as shown in the triples above and in the figure. Four instances are shown; *maidenVoyage* is specified for all four of them, so all of them have been

classified as *DeployedVessel*. *QEII* and *Constitution* have *nextDeparture* dates specified, so these two are classified as *InServiceVessel*. The remaining two vessels, *Titanic* and *Berengaria*, have specified *destructionDate* and *decommissionedDate* respectively and hence are classified as *OutOfServiceVessel*.

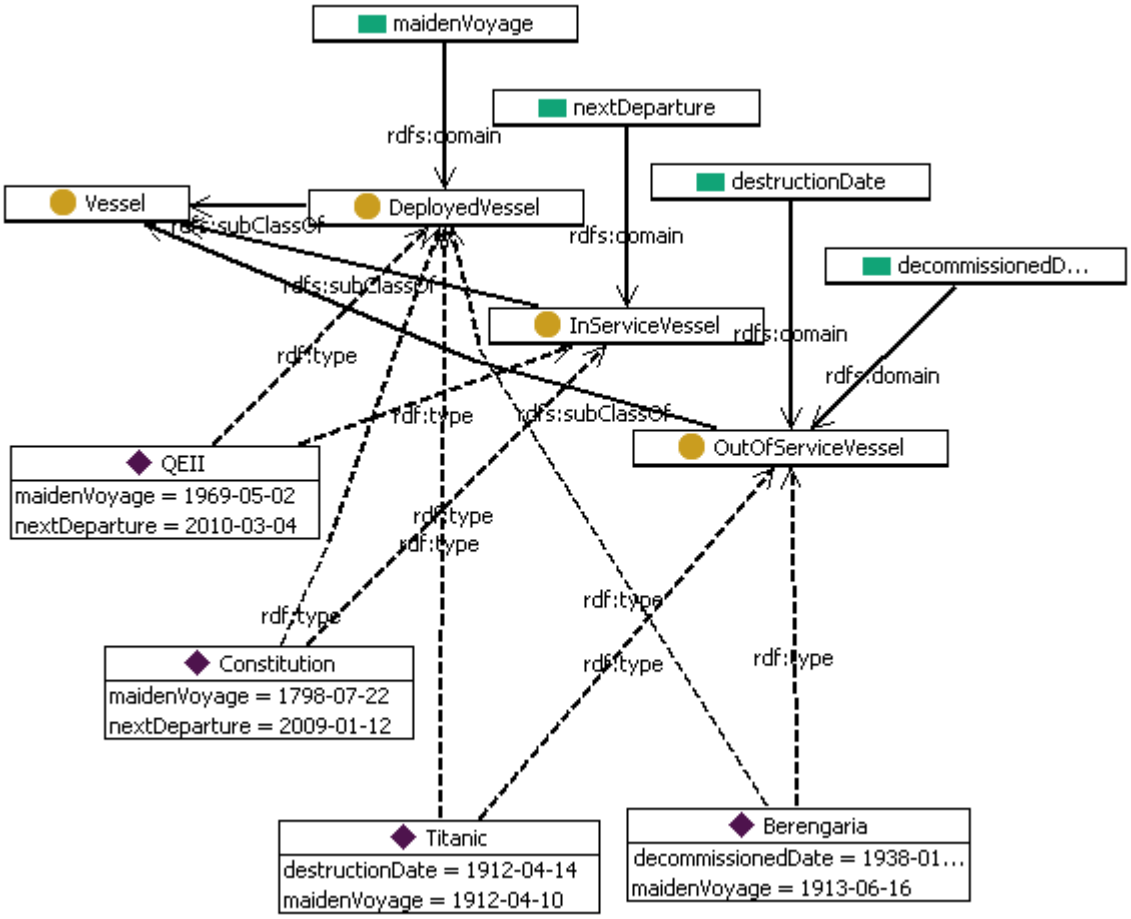

Figure 3. Inferring classes of vessels from the information known about them.

CHALLENGE

All of these inferences concern the subject of the rows, i.e., the vessels themselves. It is also possible to draw inferences about the entities in the other table cells. How can we express the fact that the commander of a ship has the rank of "Captain"?

SOLUTION

We express ranks as classes, as follows:

```
ship:Captain rdfs:subClassOf ship:Officer .
ship:Commander rdfs:subClassOf ship:Officer .
ship:LieutenantCommander  rdfs:subClassOf  ship:Officer
.
ship:Lieutenant rdfs:subClassOf ship:Officer .
ship:Ensign rdfs:subClassOf ship:Officer .
```

Now we can express the fact that a ship's commander has rank *Captain* with *rdfs:range* as follows:

```
ship:hasCommander rdfs:range ship:Captain .
```

From the information in the table, we can infer that all of *Johnson*, *Warwick*, *Black* and *Montgomery* are members of the class *ship:Captain*. These inferences, as well as the triples that led to them, can be seen in Figure 4.

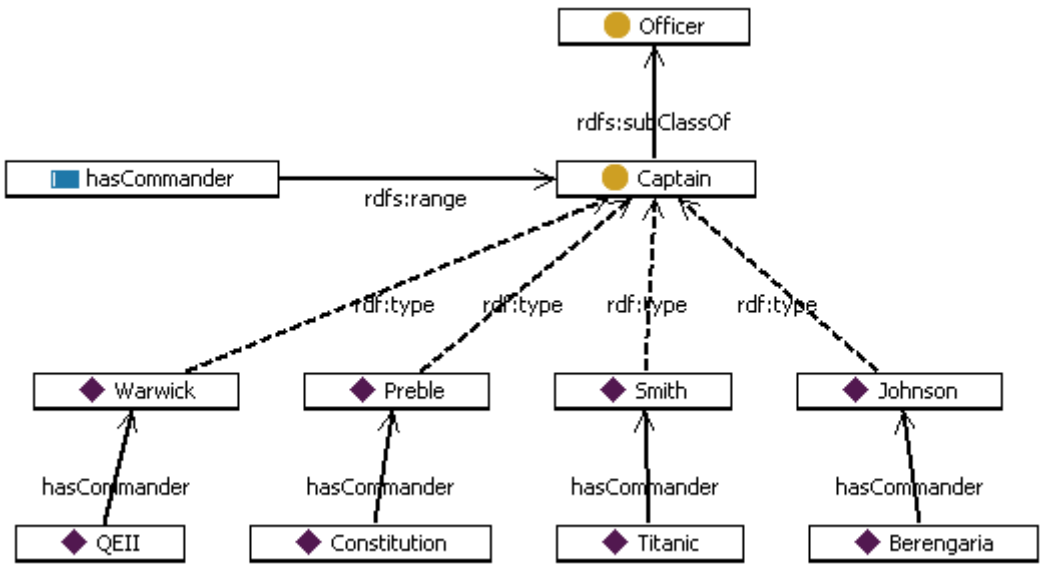

Figure 4. Inferring that the commanders of the ships have rank "Captain."

6.5.5 Filtering undefined data

A related challenge is to sort out individuals based on what information is defined for them. The set of individuals for which a particular value is defined should be made available for future processing; those for which it is undefined should not be processed.

CHALLENGE

In the example above, the set of vessels for which *nextDeparture* is defined could be used as input to a scheduling system, that plans group tours. Ships for which no *nextDeparture* is known should not be considered.

SOLUTION

It is easy to define the set of vessels that have *nextDeparture* specified by using *rdfs:domain*. First define a class of *DepartingVessels* that will have these vessels as its members. Then define this to be the domain of *nextDeparture*

```
ship:DepartingVessel rdf:type rdfs:Class .
ship:nextDeparture rdfs:domain ship:DepartingVessel .
```

From the table above (and as shown in), only the *Consitution* and the *QEII* are members of the class *ship:DepartingVessels*, and can be used by a scheduling program.

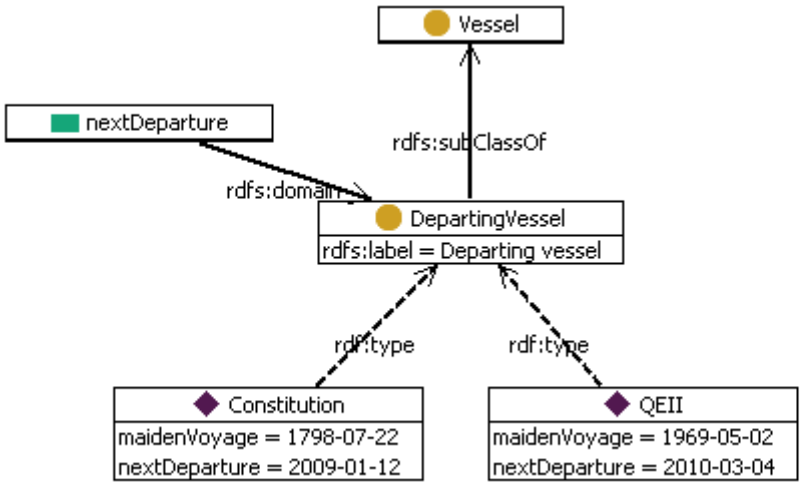

Figure 5. Ships with a nextDeparture specified are Departing Vessels.

6.5.6 RDFS and Knowledge Discovery

The use of *rdfs:domain* and *rdfs:domain* differs dramatically from similar notions in other modeling paradigms. Because of the inference-based semantics of RDFS (and OWL), domains and ranges are not used to validate information (as is the case, for example, in OO modeling and XML), but instead are used to determine new information based on old information. We have just seen how this unique aspect of *rdfs:domain* and *rdfs:range* support particular uses of filtering and classifying information.

These definitions are among the most difficult for beginning semantic web modelers to come to terms with. It is common for beginning modelers to find these tools clumsy and difficult to use. This difficulty can be ameliorated to some extent by noticing that RDFS in general, and *domain* and *range* in particular, are best understood as tools for knowledge discovery rather than knowledge description. On the semantic web, we don't know in advance how information from somewhere on the web should be interpreted in a new context. The RDFS definitions of *domain* and *range* allow us to discover new things about our data based on its use.

What does this mean for the skillful use of *domain* and *range* in RDFS? They are not to be used lightly, e.g. merely as a way to bundle together several properties around a class. Filtering results such as those shown in these challenge problems are the result of the use of *domain* and *range*. Proper use of *domain* and *range* must take these results into account. Recommended use of *domain* and *range* goes one step further; the recommended use is in one of these patterns, where some particular knowledge filtering or discovery pattern is intended. When used in this way, (e.g., using *domain* to describe

which ships are departing), it is guaranteed that the meaning of *domain* and *range* will be appropriate even in a web setting.

6.6 RDFS Shortcomings

While RDFS has considerable applicability in data amalgamation and the simplicity of its small number of axioms makes it compact and easy to implement, there are some clear shortcomings of RDFS that arise even in very simple modeling situations.

6.6.1 Multiple domains/ranges

In our shipping example, we had two definitions for the domain of *nextDeparture*:

```
nextDeparture rdfs:domain DepartingVessel .
nextDeparture rdfs:domain InServiceVessel .
```

What is the interpretation of these two statements? Is the domain of *nextDeparture* *DepartingVessels*, *InServiceVessel*, or both? What does that mean?

The right way to understand what a statement or set of statements means is to understand what inferences can be drawn from them. Let's consider the case of the QEII, for which we have the following asserted triples:

```
ship:QEII ship:maidenVoyage "May 2, 1969" .
ship:QEII ship:nextDeparture "Mar 4, 2010" .
ship:QEII ship:hasCommander Warwick .
```

The rules of inference for *rdfs:domain* allow us to draw the following conclusions:

```
ship:QEII rdf:type ship:DepartingVessel .
ship:QEII rdf:type ship:InServiceVessel .
```

Each of these conclusions is drawn from the definition of *rdfs:domain*, as applied respectively to each of the domain declarations given above. This behavior is not a result

of a discussion of "what will happen when there are multiple domain statements?" but rather a simple logical conclusion based on the definition of *rdfs:domain*.

How can we interpret these results? Any vessel for which a *nextDeparture* is specified, will be inferred to be a member (i.e., *rdf:type*) of both classes, *DepartingVessel* and *InServiceVessel.* Effectively, any such vessel will be inferred to be in the *intersection* of the two classes specified in the domain statements. This in and of itself is not a shortcoming of RDFS, even though most people find it counter-intuitive.

In Object-Oriented modeling, when one asserts that a property (or field, or variable, or slot) is associated with a class (as is done by *rdfs:domain*), the intuition is that "it is now permissible to use this property to describe members of this class." If there are two such statements, then the intuitive interpretation is that "it is now permissible to use this property with members of either of these classes." Effectively, multiple domain declarations are interpreted in the sense of set union; you may now use this property to describe any item in the *union* of the two specified domains. For someone coming in with this sort of expectation, the behavior of RDFS can be something of a surprise.

This interaction makes it necessary to exercise some care when modeling information with the expectation that it will be merged with other information. Let's suppose we have another modeling context, in which a company is managing a team of traveling salespersons. Each salesperson has a schedule of business trips. Some of the triples that define this model are as follows:

```
sales:SalesPerson rdfs:subClassOf foaf:Person .
sales:sells rdfs:domain sales:SalesPerson .
sales:sells rdfs:range sales:ProductLine .
sales:nextDeparture rdfs:domain sales:SalesPerson .
```

That is, we have a sales force that covers certain *ProductLines*; each member travels on a regular basis, and it is useful for us to track the date of the next departure of any particular *SalesPerson*.

In the context of the Semantic Web, we are interested in the prospect of merging information from multiple sources. Suppose we were to merge the information for our sales force management with the schedules of the ocean liners. This merge only becomes interesting if we map some of the items in one model to items in another. An obvious candidate for such a mapping is between *sales:nextDeparture* and *ship:nextDeparture*. Both refer to dates, and the intuition is that they specify the next departure date of something or someone. So a simple connection to make between the two models would be to link these two properties, e.g.,

```
sales:nextDeparture rdfs:subPropertyOf ship:nextDeparture .
ship:nextDeparture rdfs:subPropertyOf sales:nextDeparture .
```

using the mutual *subPropertyOf* pattern as described in section 6.4.4.1. The intuition here is that the two uses of *nextDeparture*, one for ships and the other for sales, are in fact the same.

Let's see what inferences are drawn from this merger. Suppose we have a triple that describes a member of the sales force

```
sales:Johannes sales:nextDeparture "May 31, 2008" .
```

and we already have the triple about the QEII:

```
ship:QEII ship:nextDeparture "Mar 4, 2010" .
```

What inferences can we draw from these two triples? Using first the *rdfs:subPropertyOf* inferences, then the *rdfs:domain* inferences, and finally using the *rdfs:subClassOf* triple with *foaf:Person*, we get the following inferred triples:

```
sales:Johannes ship:nextDeparture "May 31, 2008" .
ship:QEII sales:nextDeparture "Mar 4, 2010" .
sales:Johannes rdf:type ship:DepartingVessel .
ship:QEII rdf:type sales:SalesPerson .
ship:QEII rdf:type foaf:Person .
```

These inferences start off innocently enough, but become more and more counter-intuitive as they go on, and eventually (when the QEII is classified as a *foaf:Person*) becoming completely outrageous (or perhaps dangerously misleading, especially given that the Monarch herself might actually be a *foaf:Person*, causing the inferences to confuse the Monarch with the ship named after her). The asserted triples, and the inferences that can be drawn from them, are shown in Figure 6.

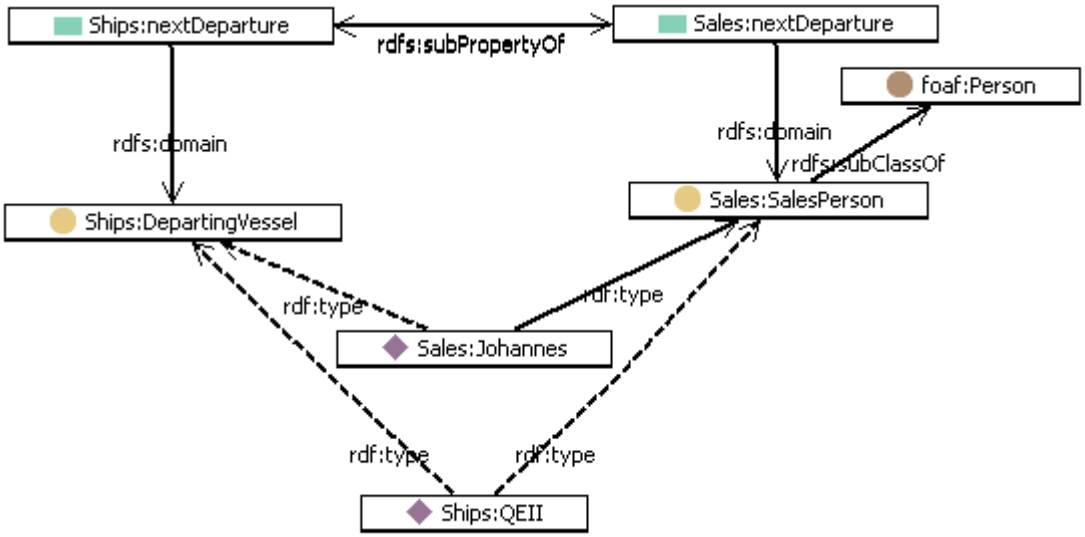

Figure 6. Inferences resulting from merging two notions of nextDeparture.

It is easy to lay the blame for this unfortunate behavior at the feet of the definition of *rdfs:domain*, but to do so would throw out the baby with the bathwater. The real issue in this example is that we have made a modeling error. The error resulted from the over-zealous urge to jump to the conclusion that two properties should be mapped so closely to one another. The ramifications of using *subPropertyOf* (or any other RDFS construct) can be subtle and far-reaching.

In particular, when each of these models stated their respective domain and range statements about *sales:nextDeparture* and *ship:nextDeparture* respectively, they were saying, "whenever you see any individual described by *sales:nextDeparture* (resp. *ship:nextDeparture*), that individual is known to be of type *sales:SalesPerson* (resp. *ship:DepartingVessel*)" This is quite a strong statement, and should be treated as such. In particular, it would be surprising if two properties defined so specifically would not have extreme ramifications when merged.

So what is the solution? Refrain from merging properties? This is hardly a solution in the spirit of the Semantic Web. Avoid making strong statements about properties? This will not help us to make useful models. Change the RDFS standard so that we can't make these statements? This is a bit extreme, but as we shall see, OWL does provide some more subtle constructs for property definitions that allow for finer-grained modeling. No, the solution lies in understanding the source of the modeling error that is at the heart of this example: we should refrain from merging things, like the two notions of *nextDeparture*, whose meanings have important differences.

Using the idioms and patterns of RDFS shown in this chapter, there are more things we can do, depending on our motivation for the merger. In particular, we can still merge these two properties, but without making such a strong statement about their equivalence.

If, for instance, we just want to merge the two notions of *nextDeparture* in order to drive a calendar application that shows all the departure dates for the sales force and the ocean liner fleet, then what we really want is a single property that will provide us the information we need (as we did in the property union pattern, section 6.4.4). Rather than

mapping the properties from one domain to another, instead we map both properties to a third, domain-neutral property, thus:

```
ship:nextDeparture rdfs:subPropertyOf cal:nextDeparture .
sales:nextDeparture rdfs:subPropertyOf cal:nextDeparture .
```

Note that the amalgamating property *cal:nextDeparture* needn't have any domain information at all; after all, we don't need to make any (further) inferences about the types of the entities that it is used to describe. Now we can infer that

```
sales:Johannes cal:nextDeparture "May 31, 2008" .
ship:QEII cal:nextDeparture "Mar 4, 2010" .
```

A single calendar display, sorted by the property *cal:nextDeparture*, will show these two dates, but no further inference can be made. In particular, no inferences will be made about considering the *QEII* as a member of the sales force or *Johannes* as a sailing vessel.

What can we take from this example into our general semantic web modeling practice? Even with a small number of primitives, RDFS provides considerable subtlety for modeling relationships between different data sources. But with this power comes the ability to make subtle and misleading errors. The way to understand the meaning of modeling connections is by tracing the inferences. The ramifications of any modeling mapping can be worked through by following the simple inference rules of RDFS.

6.7 Non-modeling Properties in RDFS

In addition to the properties described so far, RDFS also provides a handful of properties that have no defined semantics; that is to say, that there are no inferences that derive from them. We already saw one example of such a property, *rdfs:label*, in section 6.5.3. No inferences are drawn from *rdfs:label*, so in that sense it has no semantics.

Nevertheless it does by convention have a sort of procedural semantics, in that it describes the ways in which display agents interact with the model.

6.7.1 Cross-referencing files: rdfs:seeAlso

Every resource in a semantic web model is specified by a URI, which can also be de-referenced and used as a URL. In the case where this URL resolves to a real document, this provides a place where defining information about a resource can be held.

In some contexts, it might be useful to include some supplementary information about a resource for its use in a certain context. This might include schematic information (e.g., if the resource itself is a property, the supplementary information could specify its subproperties), or related data (e.g., if the resource corresponds to a table from a database, the supplementary information could be the other tables from the same database). For such cases, *rdfs:seeAlso* provides a way to specify the web location of this supplementary information. *rdfs:seeAlso* has no formal semantic, so the precise behavior of any processor when it encounters *rdfs:seeAlso* is not specified.

6.7.2 Organizing vocabularies: rdfs:isDefinedBy

Just as *rdfs:seeAlso* can provide supplementary information about a resource, *rdfs:isDefinedBy* provides a link to the primary source of information about a resource. This allows modelers to specify where the definitional description of a resource can be found.

6.7.3 Model documentation: rdfs:comment

Just as in any computer language (modeling languages, markup languages, or programming languages), sometimes it is helpful if a document author can leave natural

language comments about a model for future readers to see. Since RDFS is implemented entirely in RDF, the comment feature is also implemented in RDF. To make a comment on some part of a model, simply assert a triple using the property *rdfs:comment* as predicate. For example:

```
sales:nextDeparture rdfs:comment "This indicates the
next planned departure date for a salesperson." .
```

6.8 Chapter Summary

RDFS is the schema language for RDF; it describes constructs for types of objects (Classes), relating types to one another (subClasses), properties that describe objects (Properties) and relations between them (subProperty). The Class system in RDFS includes a simple and elegant notion of inheritance, based on set inclusion; one class is a subclass of another means that instances of the one are also instances of the other.

The RDFS language benefits from the distributed nature of RDF by being expressed in RDF itself. All schema information (classes, subclasses, subproperties, domain, range, etc.) is expressed in RDF triples. In particular, this makes schema information, as well as data, subject to the AAA slogan; anyone can say anything, even about the schema.

The semantics of RDFS is expressed through the mechanism of inferencing; that is, the meaning of any construct in RDFS is given by the inferences that can be inferred from it. It is this simple but powerful mechanism for specifying semantics that allows for the short and elegant definition of subclass and subproperty.

RDFS also includes the constructs *rdfs:domain* and *rdfs:range* to describe the relationship between properties and classes. The meanings of these constructs are given by very simple rules, but these rules have subtle and far-reaching impact; the rules are simple, but the statements are powerful.

Even with its small set of constructs and simple rules, RDFS allows us to resolve a wide variety of integration issues. Whenever you might think of doing a global find-and-replace in a set of structured data, consider using *rdfs:subPropertyOf* or *rdfs:subClassOf* instead. It may seem trivial to say that one should only merge entities from multiple sources that don't have important differences. Using the inference mechanism of RDFS, we can determine just what happens when we do merge things, and judge whether the results are desirable or dangerous. While RDFS does not provide logical primitives like union and intersection, it is often possible to achieve desired inferences by using specific patterns of *subClassOf* and *subPropertyOf*. RDFS provides a framework through which information can flow; we can think of subClassOf and subPropertyOf as the IF-THEN facility of semantic modeling. This utility persists even when we move on to modeling in OWL. In fact, the use of *subClassOf* in this way provides a cornerstone of OWL modeling.

When used in careful combination, the constructs of RDFS are particularly effective at defining how differently structured information can be used together in a uniform way.

6.9 Fundamental Concepts

The following fundamental concepts were introduced in this chapter:

rdfs:subClassOf: Relation between classes, that the members of one class are included in the members of the other.

rdfs:subPropertyOf: Relation between properties, that the pairs related by one property are included in the other.

rdfs:domain and ***rdfs:range***: Description of a property that determines class membership of individuals related by that property.

Logical operations (Union, Intersection, etc.) in RDFS: RDFS constructs can be used to simulate certain logical combinations of sets and properties.