Summary of Modeling with RDF

RDF and Tabular Data

Consider the *Product* database relation below.

ID	Model No.	Division	Product Line	SKU
1	ZX-3	Manufacturing support	Paper machine	FB3524
2	ZX-3P	Manufacturing support	Paper machine	KD5243
3	B-1430	Control Engineering	Feedback line	KS4520
4	B-1430X	Control Engineering	Feedback line	CL5934

We consider how this data can be represented in RDF. Each row describes a single entity, and the type of each entity is the table name (*Product*). We must give each row a distinct URIref. The table gives us a locally unique identifier: the primary key (*ID*). To get a globally unique identifier, use the URI for the database itself, say, http://www.acme.mfg.com; we can use the prefix `mfg`. For an identifier for a row, concatenate the table name (*Product*) with the unique key and express the result in the `mfg:` namespace: `mfg:Product1`, `mfg:Product2`, …

The column labels correspond to properties. For globally unique identifiers, use the same namespace as for individuals. For local names, concatenate table name and column label: `mfg:Product_ModelNo`, `mfg:Product_Division`, …

Overall, there is one triple per table cell plus one triple per row for type information. The triples could be spread across the Web; if so, we would use a common base to makes the URIrefs consistent. Some of the property values (objects) could be resources, identified by URIrefs, instead of literals. Some such resources could be foreign keys (unique identifiers in other relations) and would correspond to the globally unique identifiers for rows discussed above.

RDF and Inferencing

In the Semantic Web, consistency constraints on the data can be expressed in the data itself. This lets us model smart data, data that can describe how they should be used. The Semantic Web stack includes a series of layers on top the RDF layer to describe consistency constraints on the data.

The key here is the notion of inferencing: given some stated information, we can determine related information that can be considered as if stated. As an example, consider the *Type Propagation Rule* (an inference pattern):

If *A* `rdfs:subClassOf` *B* and *x* `rdf:type` *A* then *x* `rdf:type` *B*

This statement is the entire definition of subClassOf in RDFS and is consistent with informal notion of *broader term* in a thesaurus or taxonomy. Basing the meaning of constraint terms on inferencing provides a robust solution to understanding the meaning of novel combinations of terms.

No mater how a triple is available, an inference engine treats it in the same way. *Asserted triples* are those asserted in the original RDF store (possibly merged from multiple sources) while *inferred triples* are the additional triples inferred by one of the inference rules governing a given inference engine. (If an inference engine infers a triple already asserted, we consider the triple asserted.)

As an example, suppose one information source provides a list of members of the class `PassengerVehicle` and another provides a list of members of class `Van`. To find a list of all `MotorVehicles` , we use the type propagation rule, essentially merging the graphs. There are two fundamental components in this data integration example, the first being a *model* that expresses the relationship between the two data sources. Here the model consists of a single class having both the classes to be integrated as subclasses; this involves the single concept of `subClassOf`. The other fundamental component here is the notion of *inference*; inferencing applies the model to the two data sources to produce a single, integrated answer.

RDFS and Meaning

Modeling in RDF is about graphs—it creates a graph structure to represent data. In RDFS (which provides a way to talk about the vocabulary used in an RDF graph), however, modeling is about sets.

RDFS is like other schema languages in providing info about how we describe data, but it differs from other schema languages in important ways. The mechanism by which the schema in RDF helps provide meaning to the data is inference. Inference lets us know more about a set of data than what is explicitly expressed in the data and thus explicates the meaning of the original data. The additional info is based in a systematic way on patterns in the original data. RDFS provides detailed axioms that express exactly what inferences can be drawn from given data patterns.

Most modeling systems have a clear distinction between the data and its schema, but many modern systems model the schema in the same forma as the data. For RDF, the schema language was defined in RDF from the start as all schema information is defined with RDF triples. It is thus easy to give a formal description of the semantics of RDFS: give inferences rules that work over patterns of triples.

The subclass relationship is like IF/THEN of programming languages:

IF something is a member of the subclass THEN it's a member of the superclass

But nothing in RDFS corresponds to the ELSE clause—you cannot infer things from the lack of asserted membership.

The RDFS Inference Patterns

RDFS consists of a small number of inference patterns, each provides type info on individuals in a variety of circumstances.

Relationship Propagation through `rdfs:subPropertyOf`

The rule: For properties *P* and *R*, we have

If *P* `rdfs:subPropertyOf` *R*, then, if *x* `P` *y*, then infer *x* `R` *y*.

`rdfs:subPropertyOf` lets us describe a hierarchy of properties. Whenever a property in the tree holds between two entities, so does every property above it.

Typing Data by Usage—`rdfs:domain` and `rdfs:range`

The rules: For property *P* and classes *D* and *R*, we have

If *P* rdfs:domain *D* and *x P y*, then *x* `a` *D*.

If *P* `rdfs:range` *R* and *x P y*, then *y* `a` *R*.

In RDFS, we cannot assert that a given individual is not a member of a given class. In fact, there's no notion of an incorrect or inconsistent inference

Combination of Domain and Range with `rdfs:subClassOf`

The rules: Where *P* is a property and *D* and *C* classes,

If *P* `rdfs:domain` *D* and *D* `rdfs:subClassOf` *C*, then *P* `rdfs:domain` *C*.

If *P* `rdfs:range` *D* and *D* `rdfs:subClassOf` *C*, then *P* `rdfs:range` *C*.

In general, whenever we have domain or range information about a property and a triple involving that property, we can draw conclusions about the type of any element based just on that triple.

Relation to OOP

In RDFS there is no notion of inheritance per se—the only mechanism is inference. The RDFS inference rule closest to the OO notion of inheritance is the Subclass Propagation Rule. 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. The RDFS relations specify which inferences can be correctly made about it in particular contexts

RDFS Modeling Combinations and Patterns

The effect of the few, simple RDFS inference rules can be quite subtle in the context of shared info in the Semantic Web.

Set Intersection

There is no explicit RDFS modeling construct for set intersection (or union). When, however, we want to model intersection (or union), we need not always have to model it explicitly. Often we need only certain inferences supported by these notions, and sometimes these inferences are available through design patterns that combine the RDFS primitives in specific ways. If we have $C \subseteq A \cap B$ (where A, *B*, and *C* are classes), then from *x* a *C*, we can infer *x* a *A* and *x* a *B*. This inference is supported by making *C* a common subclass of *A* and *B*:

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

We cannot, however, express $A \cap B \subseteq C$.

Example: Hospital Skills

Suppose we are describing hospital staff. A surgeon is a member of the hospital staff and a qualified physician:

Surgeon $\subseteq$ *Staff* $\cap$ *Physician*

But not every staff physician is a surgeon—the inclusion goes one way.

Property Intersection

One inference is based on one property being an intersection of two others: P $\subseteq$ R $\cap$ S. We assert

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

Then we have the rule

Given *x P y*, infer both *x R y* and *x S y*.

Example: Patients in Hospital Rooms

When a patient is logged into a room, we know that

- He is on the duty roster for that room

- His insurance is billed for that room

So we assert

```
:loggedIn rdfs:subPropertyOf :billedFor .
:loggedIn rdfs:subPropertyOf :assignedTo .
```

Set Union

We express $A \cup B \subseteq C$ by making C a common superclass of A and B:

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

Then we have

Given `x a A` or `x a B`, infer `x a C`.

Summary

$C \subseteq A \cup B$ by making C a subclass of A and of B.

$C \supseteq A \cap B$ by making both A and B subclasses of C.

Example: All-Stars

In determining candidates for the All Stars, a league's rules could state that

- MVPs are candidates
- Top scorers are candidates

So we assert

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

Property Union

If two sources use properties P and Q similarly, a single amalgamated property R can be defined as

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

Then (for resources x and y),

Given `x P y` or `x Q y`, infer `x R y`.

Example: Merging Library Records

One library uses property `borrows` to indicate that a patron has borrowed a book, and another library uses `checkedOut` for the same relationship. If we are sure the two mean exactly the same, we make them equivalent:

```
Library1:borrows rdfs:subPropertyOf Library2:checkedOut .
Library2:checkedOut rdfs:subPropertyOf Library1:borrows .
```

If we are not sure they are exactly the same but have an application that wants to treat them the same, we use the Union pattern to create a common super-property:

```
Library1:borrows rdfs:subPropertyOf :hasPossession .
Library2:checkedOut rdfs:subPropertyOf :hasPossession .
```

Property Transfer

In modeling information from multiple sources, a common requirement is:

If two entities are related by some relationship in one source, then the same entities should be related by a corresponding relationship in the other source.

Given property P in one source and property Q in another, we can say that all uses of P are to be considered uses of Q with

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

Now we have

Given *x P y* infer *x Q y*.

It is perhaps strange to have a design pattern consisting of a single triple, but this use of `rdfs:subPropertyOf` is so pervasive it merits being distinguished.

Example: Terminology Reconciliation

A growing number of standard information representation schemes are being published in RDFS. Information developed in advance of a standard must be retargeted to be compliant. For example, suppose a given legacy bibliography system uses the term `author` for the creator of a book. But this is what `dc:creator` is for. We make the system's data conform to Dublin Core with a single triple

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

Challenges

We look at modeling scenarios that can be addressed with the patterns we have seen.

Term Reconciliation

The issue here is resolving terms used by different agents who want to use their descriptions together in a federated application

Challenge 2

We wish to enforce the assertion that any member of one class is automatically treated as a member of another

Solution

Take the case where a given term in one vocabulary is fully subsumed by a term in another. For example, a *researcher* is a special case of an *analyst*. We would like RDFS to infer from the fact that *x* is a researcher to that *x* is an analyst:

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

Challenge 3

Suppose there is considerable semantic overlap between the concepts *analyst* and *researcher*, but neither concept is defined sharply: some analysts might not be researchers and vice versa. Still, for the federated application, we want to treat these concepts as the same.

Solution

We use the Union pattern and define a new term, `Investigator`, for the federated domain that is not defined in either of the sources. We effectively define `Investigator` as the union of `Researcher` and `Analyst` using the super-property idiom:

```
:Analyst rdfs:subPropertyOf :Investigator .
:Researcher rdfs:subPropertyOf :Investigator .
```

Challenge 4

Suppose we have determined that the two classes really are identical. In terms of inference, we want any member of the one class to be a member of the other and vice versa.

Solution

RDFS does not provide a primitive statement of class equivalence, but achieve it with `rdfs: subClassOf`:

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

Instance-Level Data Integration

Suppose we have answers to a single question coming from multiple sources. For example, a command-and-control mission planner wants to know where ordinance can't be targeted. Several sources of information contribute. One source provides a list of facilities and their types, some of which—civilian facilities—must never be targeted. Another provides descriptions of airspaces, some of which are off-limits. A target is off-limits if excluded by either of these sources.

Challenge 5

Define a single class whose contents include all individuals from all these sources (and any ones later added)

Solution

We use the Union construction to join the two sources into a single, federated class:

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

Readable Labels with rdfs:label

Resources on the Semantic Web are specified with URIs, which are not particularly meaningful to people. In contrast, `rdfs:label` gives a standard way for presentation engines (e.g., web browsers) to display a printable name of a resource. There might be another source for human-readable names for any resource, and we can use a combination of the property union and property transfer patterns to have these names be values of the `rdfs:label` property. For example, suppose we have imported RDF information from an external form (e.g., database or spreadsheet) defining two classes of individuals: `Person` and `Movie`. `Person` has property `personName`, and `Movie` has property `movieTitle`.

Challenge 6

Want to use a generic display mechanism, using `rdfs:label`, to display info about these people and movies

Solution

We define each property as a subproperty of `rdfs:label`:

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

Data Typing Based on Use

A shipping company manages a fleet of vessels, including vessels under construction, being repaired, currently in service, and retired from service. Table 1 shows some information the company might keep. In RDF, each row corresponds to a resource of type `ship:Vessel`. The following are properties of `ship:Vessel`:

```
ship:maidenVoyage
ship:nextDeparture
```

```
ship:decommissionedDate
ship:destructionDate
ship:commander
```

And the following are subclasses of `ship:Vessel`:

`ship:DeployedVessel`—has been deployed sometime in its lifetime

`ship:InServiceVessel`—currently in service

`ship:OutOfServiceVessel`—currently out of service (for any reason, possibly retired or never deployed

Table 1 Ships

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

Challenge 7

We want to automatically classify a vessel into a specific subclass from information in Table 1:

- Has had a maiden voyage ➔ `ship:DeployedVessel`
- Next departure set ➔ `ship:InServiceVessel`
- Has a decommission or destruction date ➔ `ship:OutOfServiceVessel`

Solution

We enforce these inferences using rdfs:domain

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

Challenge 8

The above inferences concern the subject of the rows—the vessels. We can also draw inferences about what is in the other table cells. For example, we want to express that the commander of a ship has the rank of captain.

Solution

We express ranks as classes:

```
ship:Captain rdfs:subClassOf ship:Officer .
ship:Commander rdfs:subClassOf ship:Officer .
```

Etc.

Then we express that a ship's commander has rank captain:

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

Filtering Undefined Data

We can select individuals for further processing based on information defined for them.

Challenge 9

Those vessels with `nextDeparture` dates are selected as input to, e.g., a system scheduling group tours

Solution

We define the class that is the domain of `nextDeparture`:

```
ship:DepartingVessel a rdfs:Class .
ship:nextDeparture rdfs:domain ship:DepartingVessel .
```

RDFS and Knowledge Discovery

Because of the inference-based semantics of RDFS, domains and ranges are not used to validate information but rather to determine new info based on old. `domain` and `range` are basically tools for knowledge discovery not knowledge description. On the Semantic Web, we do not know in advance how info from somewhere else should be interpreted in a new context; `domain` and `range` let us discover things about our data based on its use.

Modeling with Domains and Ranges

In the shipping example, we have two definitions for the nextDeparture domain:

```
ship:nextDeparture rdfs:domain ship:InServiceVessel .  (1)
ship:nextDeparture rdfs:domain ship:DepartingVessel .  (2)
```

Any vessel for which a `nextDeparture` is specified is inferred to be in the intersection of the classes in the domain statements. This is counterintuitive from the object-oriented programming point of view, where, if a "property" is associated with two classes, we can use it with members of either class, that is, it may be used with their union.

To see the ramifications of this intersection behavior, suppose a company manages a team of traveling salespersons, each with schedule of trips:

```
sales:SalesPerson  rdfs:subClassOf  foaf:Person .
sales:nextDeparture  rdfs:dmain  sales:SalesPerson .
```

We now merge the information for the sales force management with the schedules of the ocean liners. A simple connection to make between the two models is to link their `nextDeparture` properties:

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

Two of the triples we can now infer are

```
sales:Johannes  a  ship:DepartingVessel .
ship:QEII  a  foaf:Person .
```

The issue is a modeling error. We jumped to the conclusion that two properties should be mapped so closely, whereas we should avoid merging things whose meanings have important differences.

If we just want to merge the two notions of `nextDeparture` for a calendar application, we should map both properties to a third, domain-neutral property:

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

The amalgamating property, `cal:nextDeparture`, doesn't need any domain information.