1. Introduction

1.1. Peers. Datasources.

A Peer is a repository of semantic data, information and knowledge. It performs loading plain RDF triples from enabled datasources and interacting with other Peers sharing knowledge. Datasources have 'drivers' which enables them via an events mechanism to keep in sync with Peer's operation. Lastly, Peers offer a set of APIs & ports to many protocols exposing internal (augmented) knowledge.

Population of Peer's internal models is done through decomposition and aggregation of input triples. Apache Jena models are kept for input sources (provenance) and for aligned models (ports).
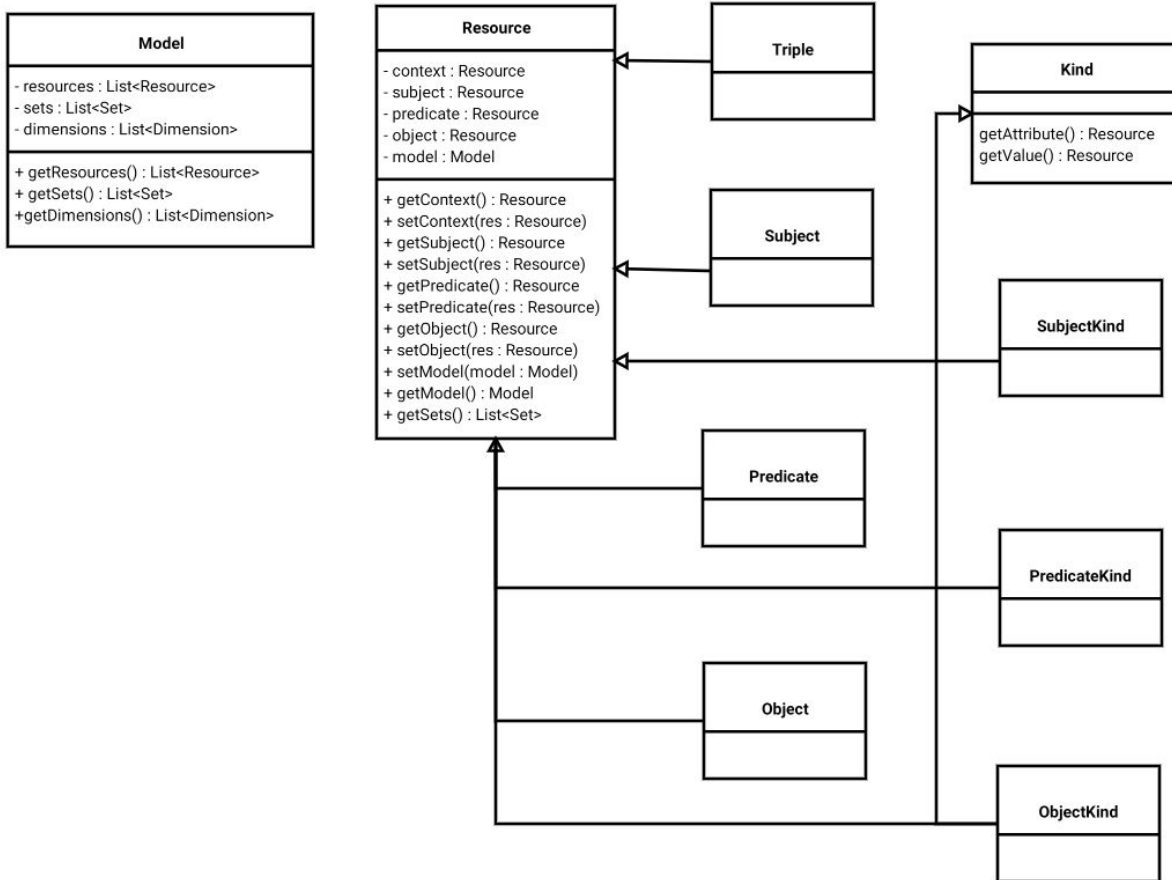
1.2. Type inference

Given the fact the only input a Peer expects is 'plain' RDF (no RDFS, no OWL) it's necessary for a given set of input triples to assert some kind of type information for each SPO part of a statement. The approach taken here (better explained in sets section below) is to regard, for example for a given Subject in a set of statements, the Predicates and Objects of those statements as describing a 'class' for this kind of Subject maybe shared with other Subjects of the same kind. In fact, for a Subject sharing a set of Predicate properties we say it's of some 'class' and given the value of those properties it is of some 'metaclass' being both class and metaclass the Kind of the Subject. Same applies for Predicates and Objects being class and metaclass the corresponding for their position in a statement.

1.3. Resources, Sets, Services

1.3.1. Resources

Resources extracted from input triples are Subjects, Predicates, Objects, SubjectKinds, PredicateKinds, ObjectKinds and Triples. All resources are reified so, for example a Subject in the source graph is represented as a resource which have this Subject as its subject part and which identifies that Subject across occurrences and contexts with a unique name (for example for align & merge purposes)

Model

- resources : List<Resource>
- sets : List<Set>
- dimensions : List<Dimension>

+ getResources() : List<Resource>
+ getSets() : List<Set>
+getDimensions() : List<Dimension>

Resource

- context : Resource
- subject : Resource
- predicate : Resource
- object : Resource
- model : Model

+ getContext() : Resource
+ setContext(res : Resource)
+ getSubject() : Resource
+ setSubject(res : Resource)
+ getPredicate() : Resource
+ setPredicate(res : Resource)
+ getObject() : Resource
+ setObject(res : Resource)
+ setModel(model : Model)
+ getModel() : Model
+ getSets() : List<Set>

Triple

Subject

Predicate

Object

Kind

getAttribute() : Resource
getValue() : Resource

SubjectKind

PredicateKind

ObjectKind

1.3.2. Sets

Sets representations of Resource (meta) models provide the means for functional query, extraction and traversal of entities. Sets population, due Set definition predicates, provides for Resource class hierarchy instantiation and Services layer population. Sync and callbacks between models.

Models are instances of other (meta) models, initially SPO triples. Given a metamodel, all its SPOs are taken as the Subjects of the next model, the Kinds are taken as the Predicates and the Triples as the Objects conforming new (Set) triples from which to infer new Kinds.

In the Sets models diagrams attribute holds for corresponding occurrences Kind's class and value for Kind's metaclass.

SPO (Initial) Metamodel:
This model holds initial input data.

SPO Model (Facts)
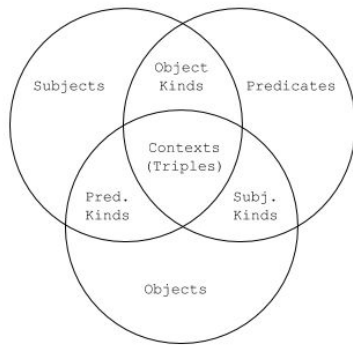
Occurrence	Attribute	Value
Subject	Predicate	Object
Predicate	Subject	Object
Object	Predicate	Subject

Triples:

```
Occurrences (Subject ex.):
[context / time] [SubjectURI] [classID] [metaClassID]

Kinds:
[metaClassID] [classID] [attribute] [value]

Contexts:
[context / time] [Subject] [Predicate] [Object]
```
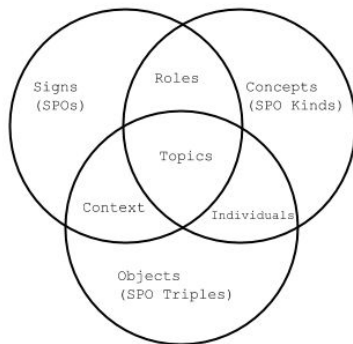
SCO (Sign, Concept, Object) Metamodel:
This model aggregates semiotically the SPO Metamodel.

Semiotic Model (SCO, Contexts)

Occurrence	Attribute	Value
Sign	Concept	Object
Concept	Object	Sign
Object	Concept	Sign

Triples:

```
Occurrences (Object ex.):
[context / Topic] [ObjectURI] [classID] [metaClassID]

Kinds:
[metaClassID] [classID] [attribute] [value]

Contexts:
[Topic] [Object] [Concept] [Sign]
```
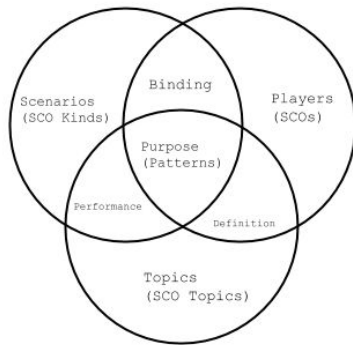
Behavior Model:
This model attempts to aggregate previous concepts in a DCI design pattern like form.

Behavior Model (TSP)

Occurrence	Attribute	Value
Scenario	Topic	Player
Player	Scenario	Topic
Topic	Scenario	Player

Triples:

```
Occurrences (Topic ex.)
[context / Purpose] [TopicURI] [classID] [metaClassID]

Kinds:
[metaClassID] [classID] [attribute] [value]

Contexts:
[Purpose] [Topic] [Scenario] [Player]
```

1.3.3. Services

Services layer provides a functional abstraction over Resources and Sets layers. Basically provides mappings between the main three kinds of entities the framework provides: URIs: SPOs (Names), Kinds ('Content' Types) and Triples (Representations).
It also provides 'grammars' for those mappings being a grammar the representation for a Content Type (Kind). Mappings and grammars are encoded as Triples / Quads. A 'runtime' or environment may be enforced by the use of OWL and an upper ontology in which to align resources given their abstract (grammar) structure.

1.3.3.1. Index

Index mapping: idx(name: ctx, representation) : type
Index grammar:
(Kind : parent) (Kind : name) (Kind : representation) (Kind : type)

1.3.3.2. Naming

Naming mapping: nam(representation : ctx, type) : name
Naming grammar:
(Kind : parent) (Kind : rep) (Kind : type) (Kind : name)

1.3.3.3. Registry

Registry mapping: reg(type : ctx, name) : representation
Registry grammar:
(Kind : parent) (Kind : type) (Kind : name) (Kind : representation)

2. Features

2.1. Links type / instance inference

Relationship (links) inference example: X coworker Y (same employer). Develop discover algorithms. Infer link types (grammars). Use Kinds, classes, metaclasses (Kinds) relations.

Infer attributes / rels from class (emp, sal, dept, manager) from links. Mgr. is emp's dept. leader.

Infer type by contents: Occurrence having other Kinds in other contexts. Grammar (abstract) occurrences of subject, context merge. Sort Kinds: Grammar hiers (parent). Adult - CanDrive. Employee must be Person & Student.

2.2. Type grammar inference

Discover primitives (metamodels). Aggregate Kinds and Kinds of class / metaclass. TBD.

2.3. Align & merge of ontologies

Similarity from grammar equivalence (equivalent grammar graphs). TBD.

2.4. Ordering of triples & events

Triple context (Quad context) holds temporal relationships in metamodels. Query for specific time range, specified interval (bounds) may fire 'events'. Events may be materialized into models. Mappings grammar could specify 'listeners' and templates for goals / purposes.

3. APIs

3.1. Services REST API

Service layer provides functionality for seamless implement a REST HATEOAS API directly over the model. Roughly, protocol would be like clients requesting, previously potentially sending a state set of triples, and retrieving an 'index' triple(s) Services representation. Then client chooses a triple and a name in that triple to submit. Then it obtains a type (Kind) for that name in that context and given this Kind it can query for further properties and retrieve them again in the form of new triples.

Client(triples) - Server(Triples)
Client(name) - Server(type rep.)
Client(type name) - Server(triples)

3.2. Functional (Dimensional) API

Functional (monads) interface for uniform Resource operations. Query, filter, traversal, predicates, assertions. TBD.

3.3. Ports

3.3.1. RDF(S) / OWL

3.3.2. SPARQL

3.3.3. OData

3.3.4. SOAP

3.3.5. Solid

3.4.6.Activation Bundles (DOM Model)

ORM + Services like bindings for specific platforms. Export bundles (JAR files? JS?) with concrete APIs. TBD.

4. Lab

4.1. Encoding & addressing

4.2. Octal order rel. encoding

4.3. Node, containers

5. Application

5.1. Dashboard example

Services Dashboard matrix
X Axis: Names (SPO URIs) : DCI Data
Y Axis: Representations (Triples) : DCI Context
Points: Content Types (Kinds) : DCI Interaction

Tool for analysis, discovery & mining. Develop views through the use of facets. TBD.