* Project:
* Environment:
* Frameworks / Libraries. Docs.
* Runtime. Tools. Services:
* Jersey / CDI. Spring.
* RDF4J (OpenRDF Sesame).
* R2RQ. Teiid. Metamodel. Any23.
* Predictions: ML / BigData (Spark).
* Index: Lucene / Solr. Quads VSM.
* Registry (Zippers): hierarchical key / value. Map Reduce.
* Naming: DIDs Services. [schema.org](http://schema.org), [sameAs.org](http://sameas.org). Matching (mappings). WordNet, OpenNLP, Wikipedia.
* Reactive (Bus): Vert.x
* Dataflow: SCDF Pipelines.
* Designer: OpenRefine. Protege Web. Editors / browsers (Forms / Flows).
* Protocols (SAILs): SPARQL, JSON-LD, JDBC (OGM / OData: Driver DatabaseMetadata). Messages (Functional / Augmentations) Declarative Services (templates).
* Clients:
* Debug Console.
* Declarative UI: ZK / XUL Templates (Content types / Components activation).
* Services APIs: Declaratively stated services: templates / queries.
* IDE. Runtime Deployment (Application).
* Clients: OData / JDBC / WS-\* (CXF) / HATEOAS / etc.
* Lectures. Topics.
* Schedule. Topics.
* JXTA. DHT. Kafka. Event sourcing. W3C DIDs. Smart Contracts. SoLiD. StratML.
* Triple store backend. RDF4J:
* Quad Layers SAIL. Core layers schema. IO: RDFS / OWL (classes, types, sameAs, etc.). SPARQL. Upper Ontology.
* Objects Layers SAIL. IO: Objects (Layers / DOM / OGM) HATEOAS.
* Functional Layers SAIL. IO: Augmentations: Browse / Navigate Model (Objects / Monads). Message Driven.
* Augmentations / Domain Services / Dataflow SAIL. IO: Messages built in Functional SAIL / Augmentation Contexts results streams. Contexts: Model Contexts, Domains Services REST Contexts URIs (APIs), connector / clients plugins (signatures). DIDs Domain Services: internal URIs. DIDs / URI mappings / APIs. Signatures. Bus.
* Augmentations: Perform Aggregation, Activation, Alignment according input Message. Update Model Contexts (upwards), Occurrences (downwards) according Augmentation stream results (Contexts). Enqueue further dataflow messages.
* Dataflow (Predicates, Mappings, Functions) domain / range (signatures). SAIL registry (types / kinds bus). Bus: topics, queues. Reactive interfaces. Dispatcher.
* Services SAIL I/O: Connectors / Clients. Distributed nodes Connector sources.
* URI: Jersey / CDI APIs. Persistence interface template methods.
* Layers APIs.
* Statement URI: Layer URI + Layer URI instance ID.
* Layers CRUD tests.
* Functional APIs.
* Functional tests (Inside Flat Map: dataflow over Object APIs):
* Predicate tests. Activation.
* Mapping tests. Aggregation.
* Function tests. Alignment.
* (MessageDomain, Predicate, Mapping, Function);
* Flat Map: Dataflow chaining over Message transforms and dataflow results.
* Augmentations performed over individual Message roles in dataflow and in dataflow with other Message roles (APIs / contexts).
* Message: Functional wrapper of Layers Contexts hierarchies statements. Hierarchy: template methods, predicate, mapping, function behaviors (inside flat map and on message statement role positions: internal dataflows).
* Predicate, Mapping, Function, URI, Statement, Value. Root hierarchy types.
* (Statement, OldEmployees : Predicate URI / Activation, SalaryUpdate : Mapping URI / Aggregation, Percentage : Function URI / Alignment
* (Domain / Context matching signatures (async streams / topics), Employee : Aggregation / Predicate, Developer : Activation / Mapping / Attributes, ProgrammingLanguages / Alignment / Function / Values);
* Monad::of(URI instance : hier);
* Monad::flatMap(Statement) : Monad (of Statement hier).
* Dispatch: each layer instance consumes matching or forwards upwards (layers signatures / zippers) incoming messages. Resulting Message enqueued for further processing.
* URI::onMessage (template methods: context, occurrence, attribute, value, role, contexts, occurrences, etc.)

**Items:**

**Ontology Matching:**

The typical use case I have in mind is the declaration of equivalence between a flat statement and a property chain, as in the two patterns below:

@prefix : <http://example.org/> .

@prefix ex: <http://ontology.org/example> .

ex:book ex:hasauthor "John".

:book :hasbeencreated :creation\_event .

:creation\_event :carried\_out :person .

:person :is\_identified\_by :appellation .

:appellation rdfs:label "John" .

another important type of equivalence, but slightly different, which I would like to declare is the one between these two patterns:

@prefix : <http://example.org/> .

@prefix ex: <http://ontology.org/example> .

ex:Architect rdfs:label "John" .

:Person :classifiedAs <http://vocab.getty.edu/aat/300024987> ;

rdfs:label "John".

<http://vocab.getty.edu/aat/300024987> a gvp:Concept ;

rdfs:label "architects"@en .

The first one declare an instance of the class Artist according to an ontology(x), the second classify as artist, using a controlled vocabulary term, an instance of a person declared using the ontology (y).

Do you know how can I express such alignments?

I heard about EDOAL, but sincerely I did not fully grasped how to actually use it.

@prefix : <http://example.org/> .

@prefix ex: <http://ontology.org/example> .

ex:book ex:hasauthor "John".

:book :hasbeencreated :creation\_event .

:creation\_event :carried\_out :person .

:person :is\_identified\_by :appellation .

:appellation rdfs:label "John" .

In the scenario above, what is the relation between ex:book and :book, and can you define some rule on how to create one from the other? Is it "same-local-name-but-different-namespace"?

If ex:book and :book are identical then a SHACL sh:equals constraint can be used

ex:BookShape

a sh:NodeShape ;

sh:property [

sh:path ( :hasbeencreated :carried\_out :is\_identified\_by rdfs:label ) ;

sh:equals ex:hasauthor ;

] .

<https://www.w3.org/TR/shacl/#EqualsConstraintComponent>

Note that SHACL includes a syntax for SPARQL-like property path expressions, and the value of sh:path above is a property chain (represented as a simple rdf:List). Other types of paths are supported too: <https://www.w3.org/TR/shacl/#property-paths>

@prefix : <http://example.org/> .

@prefix ex: <http://ontology.org/example> .

ex:Architect rdfs:label "John" .

:Person :classifiedAs <http://vocab.getty.edu/aat/300024987> ;

rdfs:label "John".

<http://vocab.getty.edu/aat/300024987> a gvp:Concept ;

rdfs:label "architects"@en .

I am also unclear whether you want to use alignments to validate constraints (e.g. "does pattern 2 exist for pattern 1"), or to construct/infer one pattern out of the other.

The complexity of the scenario above indicates that you may want to use SPARQL, because then you can more easily look up values through matches, e.g. to match "architects"@en to ex:Architect using some look-up table, and because SPARQL gives you a maximum of expressiveness.

- If you want to validate constraints, you could use SHACL-SPARQL constraints: <https://www.w3.org/TR/shacl/#sparql-constraints>

- If you want to construct target triples, you could use SHACL-AF rules: <https://w3c.github.io/shacl/shacl-af/#rules>

You can write the alignment rules in RIF, then translate them to SPARQL

and use as needed, either as part of a runtime query, or to materialize

your own "equivalence" triples (using SPARQL INSERT or CONSTRUCT).

Rules written in RIF are easier to analyze and document, for example by

expressing them in the RIF XML notation and using XSLT.

I used this approach to align and validate part master and product

structure information in a large dataset derived from different PLM

systems.

What correspondences do you want to express exactly?

From your first example, I understand that you want to express that the

property:

ex:hasauthor

from the first ontology corresponds to the property chain:

:hasbeencreated o :carried\_out o :is\_identified\_by o rdfs:label

in the second ontology.

From your second example, I understand that you want that the class of

ex:Architect in the first ontology correspond to the class of people

that are classified as http://vocab.getty.edu/aat/300024987 .

I wrote these correspondences in an alignment file at:

https://www.emse.fr/~zimmermann/edoal-example.xml

The alignment format from Inria's alignment API is meant to represent

correspondences in a way independent from how the correspondence may be

used. There are different ways of interpreting and using a

correspondence in an alignment:

1. as an ontological axiom

2. as data transformations

3. as schema constraints

4. as "bridge rules" between descriptions of different contexts

Additionally, ontology alignment correspondences may have a "measure"

assigned to them that can be interpreted as a degree of confidence, or

as a fuzzy value, or as a probability, or something else. They also have

additional metadata that makes it clear that they are relating something

from an ontology to something from another ontology. In comparison, a

logical axiom, even if it uses URIs from different namespaces, does not

make this clear.

So, depending on your use case, you may want to use Holger's suggestion

(SHACL) or Paul's (RIF + SPARQL), or something else, but you may also

postpone the decision for later (or leave it to someone else) and just

write an EDOAL alignment like I did. The alignment file can also serve

other purposes, such as alignment evaluation, composition, and enrichment.

**Zippers:**

Zippers: car / cdr on tree / list structures with predicates / iteration / recursion (reference model, contexts / occurrences). Shape Monads. E.g.: Uncle (reified relation predicate and reified Relation w./ roles / attributes). Dynamic Shape Monad on Kinds. Aggregation, Activation, Alignments.

Example: Marriage (TBD).

Predicates:

:aHusband :marriedWith :aWife

:marriedWith rdfs:domain :Male

:marriedWith rdfs:range :Female

Relationship:

(aMarriage : Relation, anStatement : marriageStatement, aKind : husbandRole, aResource : aHusband);

(aMarriage : Relation, anStatement : marriageStatement, aKind : wifeRole, aResource : aWife);

(Marriage : Relationship, Marriages : Relation, anStatement : marriagesStatements, aKind : marriageRole);

Predicates / Relationships, Relationships / Predicates entailment. Dimensional: inference / relation types / restrictions.

Encode order / hierarchies / relations (parent / child, prev / next, etc.) / iterations / conditionals / jumps.

Dimensional Domain: dimensions, units, measures, values. Comparisons, relations. State. Events (marriage example). Verbs (action, passion, state). Order (data / schema / behavior).