Business domain translation of problem spaces (WIP draft)

© 2017. Sebastian Samaruga (ssamarug@gmail.com)

Abstract:

Keywords: Semantic Web, RDF, OWL, Big Data, Big Linked Data, Dataflow, Event Driven, Message Driven, ESB, EAI.

Objectives: Streamline and augment analysis and knowledge discovery capabilities for enhanced declarative and reactive event driven process flows between frameworks, protocols and tools for Semantic Web backed integration and Big (linked) Data applications.

Achieve diverse information schema merge and interoperability (for example for different domains or applications databases). Translate behavior in one context (domain) into corresponding behavior in other context. Aggregate diverse domain data (facts) into corresponding domains state.

Description

Purpose driven (declaratively stated) systems integration and application building and deployment framework through consumer / producer abstractions of input and output platforms bindings using nodes and services.

Inputs are declaratively stated through the use of templates and filters instantiating nodes and bindings via their services. Metamodels of nodes reflect in their instances the intended behaviors previously declared through the use of patterns and routing mechanisms.

Features of deployed nodes and services implements templates and filters declaratively stated behaviors via alignment of types, resources, contexts and attributes. Augmentation, inference. TBD.

Outputs provides clients with multi-platform bindings of actionable knowledge from the features phase. Dynamic clients platform bindings provides APIs which resembles domain's use cases (CRUD / merge-sync, flows, etc) exposed as services (REST, SPARQL, SOAP, EJB/JPA etc.) in a DDD (Domain Driven Development) fashion by the rules of the MVC / DCI design patterns of object oriented programming.

Templates and filters (at core, protocol or domain levels) also enables for the exposition of a HATEOAS protocol in which output models are ready to be consumed as input ones (see Protocol section).

Example input: multiple data stores sync.

Example features: unified services API. TBD.

Example output: leaf domain node of a complex purpose organization. TBD.

Considerations:

Given a business domain problem space (data, schema and behavior for an application, for example) a ‘translation’ could be made between other domain(s) problem spaces which encompasses a given set of data, schema and behavior instances (objectives) to be solved in the other domains in respect of the problems in the first domain.

This shall be accomplished by means of an event-driven architecture in a semantics aware layer which leverages diverse backend integration (sync) and schema merge / interoperability. Also, a declarative layer is provided for aggregation and composition of related event flows of various objectives to met a given purpose.

An example: In the healthcare domain an event: flu diagnose growth above normal limits is to be translated in the financial domain (maybe of a government or an institution) as an increase of money amount dedicated to flu prevention or treatment. In the media or advertisement domain the objectives of informing population about flu prevention and related campaigns may be raised. And in the technology sector the tasks of analyzing and summarizing statistical data for better campaigns must be done.

Purpose: Domain translation of problem spaces. Trays, flows. Messaging. Goal patterns. Prompts for state completion. Purpose goals driven domain use cases (DCI) application platform (services: data / schema for facts, information, behavior). Scoped contexts flows hierarchies.

Features:

The proposed application framework to be implemented as mentioned in this document is thought to provide means for full stack deployments (from presentation through business logic to persistence).

The core components are distributed and functional in nature: REST endpoints, transformations layer, functional metamodels / node abstractions (profile driven discovery and service subscriptions).

An important goal will be to be able to work with any given datasources / schemas / ontologies without the need of having a previous knowledge of them or their structures for being able to work with them via some of the following features:

Node IO via plain RDF serialization to-from any service / backend through Bindings (below). Message wrappers of data / information / knowledge (schema-behavior) layers.

Ontology (schema) merge. Type inference. Attributes and relationships alignment / augmentation. Index service.

Order inference. Contextual order inference alignment / augmentation (temporal, causal, containment, etc.). Registry service.

Identity and instance equivalence inference. Determine whether two subjects (differents schema / identifiers) refer to the same entities. Naming service.

Infer business domain process semantics and operations / behavior from schema and data (and services). Aggregate events, rules, flows.

Node metamodel models statements into layers, each one having its own 'abstractions' for the roles each statement parts play and each one having its schema / behavior associated:

Data layer:
Resource (schema) / Event (behavior).
Data example: aProduct, price, 10;
Metamodel example: TBD.

Information layer:
Kind (schema), Rule (behavior).
Information example: aProductPrice, percentVariation, +10;
Metamodel example: TBD.

Knowledge (behavior) layer:
Class (schema), Flow (behavior).
Knowledge example: aProductPriceVariation, tendencyLastMonth, rise;
Metamodel example: TBD.

Node Resource metamodel activation: given REST endpoint semantics messages consumed activates publishing of matching quads: for a CSPO quad matching CS, corresponding CSxx quads messages will be emitted.

Knowledge, information and data layers behavior (REST CRUD) entails ‘propagation’ into upper / lower metamodel aggregated levels of statements updating them accordingly via dataflow graphs (below).

Service Features:

Alignment / augmentation features a (contextualized) Node should provide.

Align identity: merge equivalent entities (with different URIs). Align types (schema / promotion: infer type due to role in relation).

Align attributes / links: augment knowledge (properties and values) about entities (because of type / role alignment).

Align ordering: sort entities regarding some context / axis (temporal, causal, composition and other relations).

Implement functional query / transformation API language: for a given entity / concept (Monad) being able to browse / apply a function which entails some other result entity (Monads).

Resource endpoints. Node RESTFul interfaces for metamodels interaction. Provides / consumes (feeds / streams) events / messages declared via dataflow engine. The datastore should be a HATEOAS web application. DCI pattern (messages: data, subscriptions: context / roles, interactions).

Implement XSL Driven declarative dataflow engine: Streams ‘pipes’ declaratively stated in XSL for reactive behavior definitions. Functional query / transform semantics. Datastore application publish / subscribe to this (request / response 'filters'). Each metamodel concept has its own templates for backend, model and domain declarative model aggregations.

Implement a 'Node' abstraction: Integration of diverse (wrapped into metamodels) datasources / backends / services / protocols via the implementation of service contracts and exposing a plain RDF IO (dialog protocol) interface. A Node Binding to other Nodes shall allow to integrate via dataflow semantics diverse datasources / datastores. Contexts: parent, siblings, child nodes (domains).

Provide discovery services for data / schema / behavior (bus / stream events actors and roles). Bind Profiles by ontology metamodel alignment: Registry: Ordering alignment, Naming: ID and instance matching alignment, Index: Links and attribute alignment. Big Data and EAI integration patterns. Reactive adaptive dataflow containers (criteria instead of hardcoded addresses).

Architecture:

Nodes / Services (Metamodel, Functional API, Message IO, API: XSL declarative routes, ASM provider functional interfaces).

Node: Functional API / Metamodel. Resources: Metamodel, API activation signature. Node roles: Node Resource implementations: persistence, protocol, IO, alignment, peer, service, aggregation, etc.

Bundles: declarative resources / node bindings (bundle metamodel). Bundle resource implementation role (wraps nodes).

Nodes:

Nodes: declaratively instantiated. Message IO. Pub / sub producer and consumer. Reactive metamodel.

Node profiles: discovery by metamodel profiles. Declarative bindings. Aggregated node ‘pattern’ (domain of knowledge).

API: interface implementing declared instance. Streams API.

Nodes rely upon Services to provide a series of features by the means of Functional Services / Services declarative interfaces implementations: for example, Functional DOM / ASM (management object / message API).

Services provide for internal normalized metamodel (upper ontology) shared in common with other nodes regardless their purpose or role (binding, alignment, augmentation, etc.) instantiating an Application Services Model (ASM). Nodes are contextualized by their domain of knowledge into a hierarchical structure.

Nodes has Functional API which leverages ASM with query / filter / transform semantics.

Dataflow reactive event / message driven nodes. Nodes act as server / client peers in a protocol / dialog implementation. Exposed via Message aware REST endpoints.

Messages prompts (request) or augment (response) knowledge in intervening nodes conversation scopes. Metamodels aggregate or refer to new knowledge.

Message encoding: XML Metamodel Flow serialization. Metadata (ie.: services / alignment).

Message flow / routes: publisher / subscriber subscriptions and processors (DCI) stated in declaratively composable XSL templates. Pattern based discovery / matching. Discovery by profile: criteria / patterns instead of endpoint addresses. Bus: dataflow streams (pipes). Actor / role pattern.

Node ASM implementation interfaces: determines API needed to be implemented for a Node to fulfill its purpose / role.

Services

Functional state. Business logic. Routing / dispatch. Behavior (unified API). Template metamodel encoded. Discovery (‘operations’) functional invocation.

Index, Naming, Registry, etc. Align, augment. TBD. Distributed / reactive. Alg.: name - obj.

API: interface implementing declared instance. Streams API.

Services provides enhancements to a node’s IO and metamodel. Their invocation is performed in a self arranged manner consuming input or producing output for their node. Services are dynamic and pluggable or discoverable in a functional fashion.

Services: Node has services API (Index, Naming, Registry) used for alignment / augmentation and by the Node (or other contexts nodes) for endpoint interactions resolution.

Scopes (levels): knowledge, information, data in schema / instance (behavior) Message encoded pairs. Message encoding (message, flow, rule, event, class, kind, resource). Node data model: Node contexts / scopes: discovery / resolution. Domain translation.

Contexts: Node aggregation (parents, siblings, children dataflow: domain oriented translation) into hierarchical domains (Nodes).

Message and Node services based communication flows. Ontological (domain / schema) and instances driven grouping and arrangement of Nodes, messages and services. Query / discovery / binding / alignment for Nodes exposing some 'schema' and 'instance' (in all three data / info / knowledge levels for schema and behavior).

Service interface: uniform, declarative. Performs behavior according to Node (IO) messages / discovery / routes.

Node Service: A Node is another Node Service (by Node address binding / discovery).

Metamodel Service.

Ontology Service.

Index Service (property resolution).

Naming Service (semantic patterns).

Registry Service (context patterns).

Alignment Services (ID, property, order).

IO / Persistence Services:

JBoss Teiid / Apache Metamodel. Messaging. Apache Jena.

Reactive dataflow engine (Web endpoint): REST HATEOAS (HAL / JSONLD) XSL Declarative contexts / subscriptions.

Node structure: Services interfaces implementations + Declarative Configuration. Interacts / discover other services / Nodes (by data / schema / behavior of facts / information / knowledge descriptions).

Node deployment: WAR (JEE) or OSGi Bundle (Apache ServiceMix) archetype implementation (Node kinds). Deployment: Node bindings, address discovery / registry (WebApp / Bundle endpoints).

Metamodel

Metamodel: Node instance specific Model monad endpoint (backend / triplestore sync / aggregation) wrapped. Specific Statement and Resource implementations. Bundle declarative settings. Model Statement / Resource Functional / CRUD: aggregate one object instance per each triple store quad.

Hierarchy: Model : Statement : Resource.

Resource(T extends Source): Quad. Example: DatabaseResource.

Statement(T extends Resource): Quad.

Model(T extends Statement): (Endpoint, Rsrc, Rsrc, Stmt);

Model: (Database / Service, Table / Op, Row / Args, Resource);

Aggregated resources: extends Model. Example: Fact(T extends Statement / FactStatement);

Metamodel: Apache Jena backed triple store for Model. Aggregated Resources: instantiated from Model (Model hierarchy).

Metamodel: Aggregation: Fact, types (dimensional), CEP. Rules. Flows. Streams (aggregate, align, reason).

Statement / Table: (Resource, Resource PK, Resource Col, Resource Val);

Resource: (Player, Occurrence, Attribute, Value);

Model / Resource / Statement monads wraps specific node roles implementations of IO (specific models, Database, has specific resource and specific statement instantiated). Provides sources CSPO IO / CRUD. Then aggregates Fact, Kind, Class, Event, Rule, Flow (Resource monads wrappers of their players).

Fact (dimensionally aggregated from SPO): (Subject, Table, Column, Value);

Kind / Table: (Table, Subject, Column, Class);

Class: (Class, Table, Column, Value);

Event: (Event, Fact, Kind, Class); Fact occurring.

Rule: (Rule, Event, Kind, Flow); Aggregated from Events.

Flow: (Flow, Rule, Class, Class); Resulting attribute class flows.

Application / Binding: (Binding, Input, Match, Output); Bound functions.

Statement types wraps player resources. Resource wraps player aggregated statements.

Architecture: implementation view

Resource: (Name, Input, Feature, Output);

Node: Resource kind. Metamodel. Functional interface.

Bundle: Declarative arrangement of Nodes.

Node roles (by their Resource kind):

Backend / persistence:
- Inputs
- Features
- Outputs

Alignment / augmentation:
- Inputs
- Features
- Outputs

Port / endpoint (CRUD / behavior flows):
- Inputs
- Features
- Outputs

Binding: Client platform endpoints
- Inputs
- Features
- Outputs

Bundle: Declarative Resource Templates

- Internal declarative Metamodel representing deployment Resources. Reactive dataflow activation graph (distributed).
- Services, Nodes, Peers, Ports, Bindings Resource declarations. Subscriptions / transforms.

Node: Functional API / Metamodel. Resources: Metamodel, API activation signature. Node roles: Node Resource implementations: persistence, protocol, IO, alignment, peer, service, aggregation, etc.

Bundles: declarative resources / node bindings (bundle metamodel). Bundle resource implementation role (wraps nodes).

Resource interface IO: message dispatch, routes, endpoints, content type / format / contextual resolution of consumer (from subscription patterns). Protocols (quad / REST HATEOAS). Dialog message augmentation.

Message routing via Resource activation signature (resource, in, feature, out) pattern. Message IO coming from inner / outer Resource layers.

Resource resolution: index, naming, registry. Patterns. Templates.

Resources: Each Resource has its metamodel / functional / dataflow endpoint / interface (templates). Implemented reactive behavior according role (Service: persistence / alignment, Node: merge / augment, etc.): patterns & templates, IO (Resource functional implementation). Declarative Bundle description metamodel: instances and bindings of Resource(s).

Client platform bindings (augment services dialogs via events API over CRUD). Query client contexts over augmented state regarding schema / data of facts, info, knowledge. Common API: standard displaying / protocol (activation). JAX-RS, JAX-WS, JCA.

Example: Persistence Service over Apache Metamodel / JBoss Teiid via D2RQ. Node binding service links federated deployments. Port / Binding Resources expose services through protocol spec + endpoint service (IO).

Metamodel encoding: TensorFlow models. Aggregation. Layers. Reactive / Functional Node API.

Java platform binding: JCA / JavaBeans Activation Framework / XML Beans serialization (DataContentHandlers over standard generic model bean: REST / functional transform verbs over content type). XML / JSON HAL bindings. Export schema for DCI / ORM like bindings.

Bundles deployed as Apache ServiceMix / Red Hat Fuse OSGi bundles.

Implement ServiceMix OSGi Blueprint DSL (namespace) for Node, Metamodel and Bundle bindings implementations. Message subscriptions and pattern routing. Aggregation and Resource backends.

Blueprints archetype: Node metamodel impls. (backends, alignment, endpoint). Aggregation. Camel (reactive) routes: bindings / subscriptions, topics. Transforms. CXF (reactive JAX-RS) endpoint for endpoint metamodel resource monad. Client platform: JCA over endpoint protocol. Streams ETL. Request backend for specific patterns.

Metamodel: (Resource, Occurrence, Attribute, Value);
Metamodel: (Statement, Resource, Attribute, Value);
Metamodel: (Model, Statement, Attribute, Value);

Metamodel: backend Resource: IO (messages, persistence) / aggregate.

Resource API:
Resource.resource(String URI);
Static / factory. URI: Resource monad backend (JDBC, REST, SPARQL, etc.) Resource listens to / publish to. Apache Jena persistence interceptor / cache.

Resource (monadic instance) wraps their occurrence instances sets (occurrences, query, apply transforms).

Resource.occurrences() : Statement.

Statement.occurrences() : Model.

Metamodel aggregation hierarchies subclass monad wraps superclass instances. Flow : Rule : Event : Class : Kind : Fact : Model.

Resource.apply(Resource pattern) : Resource. Transform. Update. Apply pattern query / match: add / modify corresponding occurrences to player context resource.

Resource.query(Resource pattern); Apply to Model. Quad pattern matches. If none then build Resource from monadic resource factory. Performs resource activation (messages transform results, apply occurrences).

Metamodel messages: (match, apply) CSPO quads for each Resource hierarchy new instance: quads message. Apply occurrences to each local matching CSPO. Context of each applied CSPO: complement triple (i.e.: CPO for S) resources history. Metamodels aggregate new occurrences.

Resource history: invoked (match, apply) transforms in contexts until base resources. Complement based ID encoding.

Metamodel

Dimensional statements:

Type inference: aggregated attributes / values (resources / kinds), resource representation (parse statements / contexts: (700 / Units, Order / Product, Qty.)). Encode relations:

Resource statements (Facts): (aTravelFact, travel, distance, 60km); (aTravelFact, travel, origin, placeA); (aTravelFact, travel, destination, placeB);

Kind: (Kind, Fact, Class, Resource);

Rule: (Rule, Resource, Class, Resource);

Class: (Class, Kind, Attribute, Value);

Flow: (Flow, Value, Attribute, Value);

Fact statements (Events): (60km, placeA, distance, placeB);

(placeA, placeB, destination / origin, placeB);

(O, O, P, O) : Fact statements.

(For resource statements, kind, class, facts build: event, rule, flow).

Event: (60km, placeA, distance, placeB);

Rule: (Resource / Family, Resource / Peter, Class / Brother, Resource / John);

Flow: (Value / State, Value / Argentina, Attribute / Capital, Value / BuenosAires);

State is (an inferred) value for a class attribute of Argentina. Reified kinds / class / etc may play resource / attribute / value roles.

In La Plata; In Buenos Aires; In Argentina; The x of y (from specialized to generalized): The capital of the place.

Reification. Grammar layer. Sets (occurrence, attribute, value) predicates.

Domain / Range Kinds aggregation (grammars). Business domains (specific grammars templates).

Metamodel Service: Classes / Aggregation

Alignment: abstract upper ontology. Primitives. Protocol comparison / rels contexts. Concept lattice / FCA. Resource encoding.

Type, order, attribute metadata aggregated as kinds / classes.

Main Metamodel and upper ontology classes and instances are modelled following a simple principle for mapping RDF quads to OOP classes and objects:

Classes are modelled as a quad hierarchy of OOP classes:

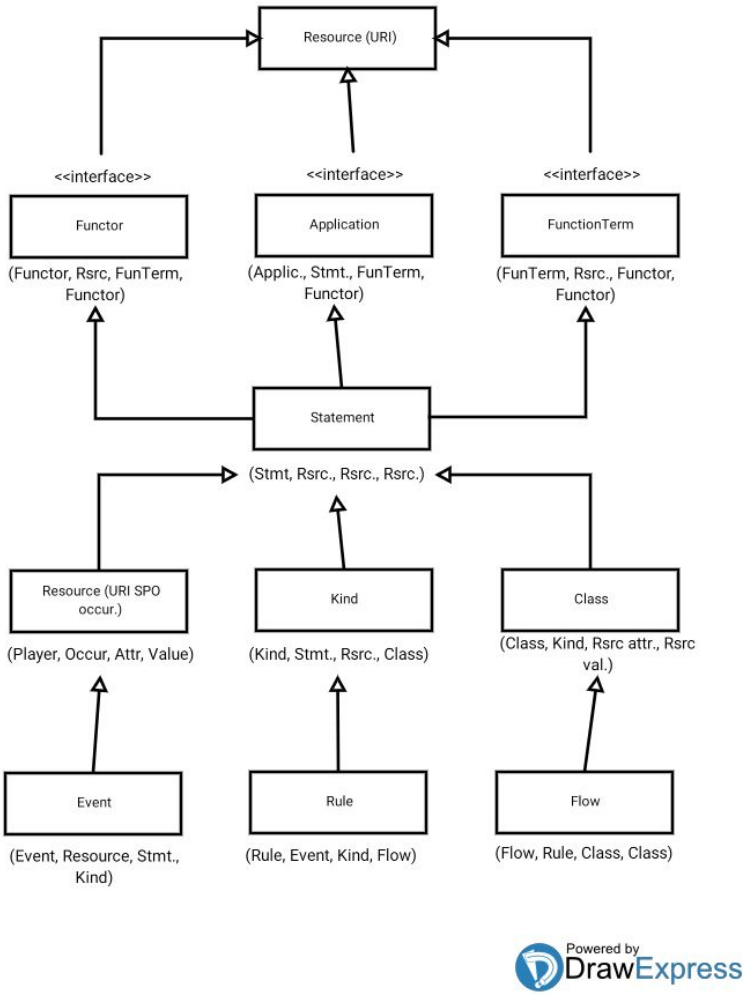
ClassName : (playerURI, occurrenceURI, attributeURI, valueURI);

A quad context URI (player) identifies an instance (in an ontology) of a given OOP class. All ontology quads with the same context URI represent the same ‘instance’.

An instance (playerURI) may have many ‘occurrences’ (into different source quads). For example: a resource into an statement.

For whatever occurrences a player instance may have there will be a corresponding ‘attribute’ and ‘value’ pair being, for example, if the player is having a subject role in an statement then the attribute is the predicate and the value is the object of the given statement.

For input statements aggregation is done into Resource(s), Kind(s), Class(es), Event(s), Rule(s) and Flow(s).



Given a Resource a Kind states it has a given Class for its occurrence into an Statement:

Peter : Employee in
(Peter, worksAt, IBM).

Given an Event a Rule states that some Flow is being accomplished for a given Kind (role):

GoodEmployee : Promotion is
SalaryRaiseFlow (for Employee Kind salary attribute update).

Update Metamodel for Flow nesting / Message encoding via quad players occurrence.(TBD. Example):
(Event, Class, Resource, Statement);
(Class, Kind, Resource, Resource);
(Kind, Resource, Statement, Class);

Data (instance / class aggreg.): Resource (schema), Event (behavior).

Information (instance / class aggreg.): Kind (schema), Rule (behavior).

Knowledge (instance / class aggreg.): Class (schema), Flow (behavior).

Layers. TBD.

Functional DOM:
Functor (Type).
FunctorTerm (Member).
Application (Type instance members). Reified Metamodels.

Node : URI (schema / behavior)
Quads it produces / consumes when activated). Resource activation graphs.

Binding : (Binding, Node producer, Message pattern, Node consumer);

Binding via XSL declared subscriptions.

Message: encode / aggregate interaction state.

TBD.

Metamodel Service: Inference / Layering / Reification

Type inference: basic type inference (for aggregation and datasource syndication / merge) is performed when aggregating (layers of) metamodels. Augmentation service nodes are expected to provide more complete alignment facilities.

Typing via Resource Kinds: each SPO in a Statement has its corresponding Kind which has an ‘attribute’ and a 'value’ corresponding to its occurrence in this position in a triple.

For example, in the triple:

(Peter) (worksAt) (IBM)

Has a 'SubjectKind’ of (worksAt, IBM) for its Subject (Peter). Subject’s attribute and value are (worksAt) and (IBM) respectively. This metadata could be used for basic type inference by aggregating ‘Employees’ (worksAt domain) and specific employees (which work at IBM) or ‘range’ (metaclass). This way Kinds aggregate Classes with their occurrences having a Kind, maybe, multiple Classes aggregated.

The same holds for classifying Predicates and Objects into their types and their corresponding meta-types. Different Model layers (discussed below) have its own (Predicate based) SPO Sets definitions and, thus, their own Statement and Kind structures.

Kinds reification may be performed given, for example, in this situation: 'Employee’ SubjectKind becoming a 'Employee’ Subject (resource). This way attributes and links may be stated for the 'Employees’ set in general (Grammars model).

Domain / Range Kinds aggregation (grammars). Business domains (specific grammars templates).

Aggregation: Data, information, knowledge layers.

Data, information, knowledge example: price, price variation, price tendency.

Knowledge aggregated in models should be capable of being abstracted in such a way that general knowledge may be obtained from specific knowledge. Richer query / browsing and inference capabilities should arise from such schema.

Data: [someNewsArticle] [subject] [climateChange]
Information: [someMedia] [names] [ecology]
Knowledge: [mention] [mentions] [mentionable]

In La Plata; In Buenos Aires; In Argentina; The x of y (from specialized to generalized): The capital of the place.

OntoClean. Primitives. Upper ontology. Bit map coding. Octal comparison results mask (order rel expressed in three bits). FCA Concept lattice.

(Fact1, Peter, wife, María);

(Fact2, Man, relationshipWith, Woman);

Patterns. Comparisons. Order / contexts. Comparing two facts (resources) in a given context must shield a third resource (alignment, discovery by distance).

Knowledge to compare Fact1 as instance of Fact2. Knowledge to compare two facts (in functional context) and sort them ('causeOf', 'after', 'before', 'partOf', 'equality' example contexts) according resulting resource / distance.

Model Layers:

The first Model (Facts) is built from raw input triples (SPO Resources) and its Events, Kinds, Rules, Classes and Flows are aggregated according its Statements quads and Resource occurrences.

The second level (Signs) is the result of building statements via reification of Fact triples as subjects, reification of Fact Kinds as predicates and reification of Fact resources as objects.

The third level (Behavior) is done performing the same procedure done in the second level in respect to the first one (reification).

All levels aggregate Kinds, Classes, Events, Rules, Flows the same way as the first level.

Metamodel Service: Functional API semantics

Functional properties: idempotence. Versions. Topics. Streams. Protocol headers (ordered Etag: order relationship aligns contexts: attributes, identity / types).

In context of an attribute comparison: having same attribute / same value. In context of type / identity comparison: being same type / same instance.

Comparison result values holds metadata (ie.: 0 being equal, negative / positive values indicating encoded 'distance'). Contexts (type / instance / attributes) occurring in a (temporal / revision) context.

Encode Functor / Monads, Function / Terms and Applications (flatMap, filter, etc.) as Resources (XSL Templates bindings).

Functional DOM (ASM / Upper ontology):
Functor (Type).
FunctorTerm (Member).
Application (Type instance members).

Reified Metamodels (upper ontology).

Resource URI scheme / format for proper feature / activation Node patterns.

Reactive dataflow. Message (events) driven. Actor / Role pattern. DCI (Message, Publisher, Subscriber: Subscription, Interaction: producer / consumer), Message Type (discovery bindings by data kinds, roles and contexts).

Message semantics:

Message monads / patterns.
Referenced metamodel resources in Message body (links).

Message routing. Bindings, node as resource activation graphs: node quad, resource aggregation activation graphs: build Message matching CSPOs to complete matching statements and populate / dispatch contextualized Message result which applies to corresponding bindings.

Activation (aggregation) example: attribute / value: activates Class. Class: activates Kind(s). Kind: activates Statement(s). Message 'posted' in reply (bindings / dispatch) aggregated data, information, knowledge schema / behavior.

Node : URI (schema / behavior)
Quads it produces / consumes when activated). Resource activation graphs.

Binding : (Binding, Node producer, Message pattern, Node consumer);

Binding via XSL declared subscriptions.

Message: encode / aggregate interaction state.

Functional API: Service ASM

Dataflow activation graph. Templates instantiate nodes.

Metamodel ASM (AOM: data Application Object Model) should be to RDF / Semantic Web as DOM
(Document Object Model) plus JavaScript is for HTML / XML.

DOM: Functor, Term and Application interfaces models OO representations of metamodels. Dynamic Object Model.
Data. Variables. Placeholders. Upper ontology alignment.

All hierarchy classes (including Functor, Application and Term interfaces) are defined in terms of RDF
Quads (OOP mapped).

Data input (event / message stream), Reactive activation (dataflow graph: templates), Definitions (producer types).

AST / Monadic parser combinators on statements input / aggregation.

The Quad statements are of the form:

Quad : (Player, Occurrence, Attribute, Value);

Functor (functional wrapper of Resource):
(Functor, Resource, Term, Functor);

Term (bound function / dataflow op wrapper of Resource):
(Term, Resource, Functor, Functor);

Application: TBD.

Example: (‘Someone’ : Resource).flatMap(Employee : Kind) : EmploymentKind : Someone’s jobs (reified Kind).

TBD.

Message IO: Encode Metamodel Flows:

(Resource hierarchy). Alignment metadata (attributes / values). Reactive: message / event driven dataflow.

Reactive dataflow. Message / Event driven. Actor / Role pattern. DCI (Publisher, Subscriber,

Subscription, Interaction: producer / consumer instances, Message Type (bindings by data kinds, roles and contexts).

Message encodes data / information / knowledge schema and behavior for a given event based endpoint interaction. A Message (part) is susceptible to a Functional API query / transform in the context of declarative stylesheets bindings (XSL).

Message as monads. Message patterns matching in bindings. Nesting of Metamodel resources in Message body (links: attributes / values). Message routing (declarative bindings, node as resource activation graphs: node quad, resource aggregation activation graphs).

Build Message matching CSPOs to complete matching statements and populate / dispatch contextualized Message which applies to corresponding bindings).

Activation (aggregation) example: attribute / value: activates Class. Class: activates Kind(s). Kind: activates Statement(s). Message 'posted' in reply populated. Bindings dispatch Statements (within Message context).

The most elemental Message Flow is a Subject having an Object as its Property: SPO Statement Message.

Given a Resource a Kind states it has a given Class for its occurrence into an Statement:

Peter : Employee in
(Peter, worksAt, IBM).

Given an Event a Rule states that some Flow is being accomplished for a given Kind (role):

GoodEmployee : Promotion is
SalaryRaiseFlow (for Employee Kind salary attribute update).

Metamodel Message flow (XML):

```
<Message>
          <Flow>
                    <Rule>
                              <Event>
                                        <Class>
                                                  <Kind>
                                                            <Resource>
                                                            </Resource>
                                                  </Kind>
                                        </Class>
                              </Event>
                    </Rule>
                    <Property>
                    <Attribute>xyz</Attribute>
                    <Value>123</Value>
                    </Property>
          </Flow>
</Message>
```

Message: XML like with nesting structure / repetitions / links / references (for XSL declarative pub / sub pipes). Metadata. RESTFul HATEOAS (HAL / JSONLD).

Layers:
Data: Resource (schema), Event (behavior).
Information: Kind (schema), Rule (behavior).
Knowledge: Class (schema), Flow (behavior).

Encode (multiple) quads attributes / values (for a resource with the same parent).

Update Metamodel for nesting / encoding via quad players occurrence.(TBD. Example):
(Event, Class, Resource, Statement);
(Class, Kind, Resource, Resource);
(Kind, Resource, Statement, Class);

Metadata:
Flow encoding: order, contexts encoding.
Rule encoding: schema (attributes / links) type promotion encoding.
Event encoding: Data identity encoding.

Nodes: retain 'dialog' state (Metamodel + Messages) vars / placeholders. Message 'model': data, schema / transforms, behavior. Lambda (server less) ‘runat’ features. Activation graph.

Resource REST endpoints (in / out Node templates). Declarative bindings (in templates) with Functional API features (markup, reified Term, Functor, Application). Message IO: augmented dialog / prompts protocol.

Message IO: Publish / Subscribe:

Dialog protocol: (post / prompt) pattern based (endpoint, queue, topics) semantics.

Transport: Nodes setup / discovery / dataflow bound by declaratively composed pipes (streams) in XSL (patterns over bus / queue / topic endpoints). Functional API ‘callbacks’ (reactive typed / bus pattern based publishers / subscribers) into templates.

Dialog / Flow Messages (IO: variables, wildcards, placeholders).

Example: Bindings posts its ‘schema’ on creation (Flow Message).

Augmentation / Client eventually ‘ask’ Binding for ‘data’ in its schemas. Client gets ‘augmented’ Flow(s) for its
requests. Nodes keep their Metamodel(s) updated (with references to other Nodes models / resources).

Hypermedia (REST HATEOAS: HAL / JSONLD encoding) Messages holds links / patterns (discovery) to other Nodes model resources / Messages.

TBD.

Resources:

Encoding: TBD (AlphaRGB).

IDs, expressions, distance. Predicates. Comparison results.

Dimensional encoding: resource / facts statements.

Message IO: XSL Based routing:

Node + metamodel + endpoints reactive distributed dataflow based activation graphs. Dialog protocol: augmentation / enhancements.

Node reactive dataflow integration setup (bindings, routes, transforms in XSL declarative reactive streams configuration): DCI (Message, Subscriptions, Processor)

Functional (API) transforms declared as resource instances into metamodel for markup composition in XSL templates. Access Functional API from templates via resource representations. DOM.

RESTFul Message metamodel layers endpoints Flows.

Business Domain Translation of Problem Spaces:

Domain translation: fulfill (dynamic) template.

Dashboard: actionable domain translation of problem spaces flows.

TBD.

Node API:

Node components: Services
Metamodel
Functional API
Message IO
API: XSL declarative routes
Persistence / Bindings
Alignment
Others

Node Services:
Index service: Attribute / link (promotion) inference.
Naming service: type / identity inference / resolution.
Registry service: Contexts and ordering resolution (comparable).

Services complements functional interfaces / DOM to provide an application model.

Contexts: Discovery for various resource kinds, endpoints, message patterns.

Endpoints: each resource class instances into a Node is bound to a RESTful endpoint which translates requests and responses (asynchronous Message events) by means of declarative configuration via the use of XSL stylesheets and an HATEOAS Web Application framework.

Node Archetypes:

Binding: Implement Node for exposing data sources / Node syndication / bindings (RDBMSs, services, protocols, formats, endpoints, etc.). Synchronization.

Augmentation: Implement Node interfaces for knowledge enrichment (data enhancement / linking / augmentation, reasoning, inference, alignment learning, etc.). Services.

Client: implement user interaction, services, protocols (endpoints: REST, SOAP, SPARQL, MVC, Web). Contexts.

Node deployment: Node contexts. Discovery (Subscription) resolver. Contexts. Message flows. Domains.

Binding Node:

The main idea is to be able to merge diverse data sources (from existing applications databases for example) and from they and their metadata expose 'declarative' application models which can be used for domain driven front ends or services.

Bindings are meant to provide Node syndication / sync interactions with backend data sources by the translation of data sources schema and instances into a common metamodel and an upper alignment ontology (Functional DOM).

Message flow: Nodes contains declarative logic for knowledge aggregation, augmentation and enhancements via the concept of a RESTful driven HATEOAS dialog protocol.

Alignment Node Service: ID / Merge (type / instance)

This kind of Node Service is meant to provide equivalence / identity knowledge regarding subjects in the scope of a conversation. Type inference also should be provided as means of query able knowledge regarding layers.

Infer identity: subject equivalence (URIs in different namespaces refer to the same objects / resources).

Infer types: subject playing same roles in same contexts (promoted) to the same types. Attributes / links of the new type should also be considered.

TBD: Encode type / identity relationships as order (comparison) of resources (ie.: set / superset of attributes / values).

Naming services. Resolution.

Alignment Node Service: Ordering / contexts

Contextual comparison augmentation. Temporal, causal, others.

Provide means to compare subjects regarding some context. Example: in a dayOfWeek context, friday is after monday. In an ‘academic’ context graduate is after undergraduate.

Comparison results helps to augment other alignment methods.

Registry services. Resolution.

Alignment Node Service: Attributes / links

Property / value attribute and links augmentation. Promotion of types and properties due to a subject joining a new relationship.

Example: someone being hired, type promotion: employee; attribute / link augmentation: salary, manager.

Example: (Ctx.: rel, Peter, Joe) : where neighbors, then friends, and then partners. Transitions. Truth values (temporal, for ordered 'names').

Index services.

Service Nodes:

Index services.

Naming services.

Registry services.

Learning services.

Inference services

Reasoner services.

Dimensional services.

Analysis services.

TBD.

Client Node (service, protocol, app):

Implement Node to provide (platform specific) bindings to provide facilities for implementing

clients (Web App, expose service endpoints, etc.).

Declaratively bound (subscriptions) with other Nodes specifying ‘backend’ providers.

TBD.

Application (Binding Node):

General purpose dashboard. Reports over heterogeneous data sources (ASM).

Application (Client Node):

General purpose client / browser for Binding Node Application Services Model (ASM).

Business Domain Translation of Problem Spaces: TBD.

Generic client (browser).

Application Demo: Dashboard (CRUD enabled) of aligned / merged syndicated data sources. Inferred business use cases (domain driven development) frontend. Flows.

Implementation Details

Functional expressions as Resources (Functor, Term, Application). Pattern IDs, variables, placeholders, dialogs (dataflow: encode graph).

Actor / Role (class / metaclass). Context / Interaction. DCI reactive / dataflow. Promotion (event roles).

Graph encoding. Naming scheme (Profile IO translation)

Dimensional reified quads: (Dimension, Fact, Measure / Unit, Value);

NLP: Substantive, Verb, Adjective (computer, computes, computed).

Upper (internal) OWL / RDFS ontology. Restrictions based alignment.

Infer equivalent properties by equivalent property domain / range (kinds). Domain / range kind alignment, ID, links, order inference by equivalent property kinds. Encode inferred schema as upper OWL / RDFS Statements.

Reactive Streams: Subscriber, Publisher, Subscription, Processor <T> (Web, XSL Declarative Resource Dataflow Pipes).

Resource: Data. Subscription(s): Context (actors / roles). Processor(s): Interactions (behavior instances).

Discovery. Internal upper ontology. Sust, Verb, Adj. Reify layers entities. Terms (reified

grammars, dimensional, rules). Constraints. Shapes. Domain / range, ordering rels comparisons inference. Schema merge / sync.

Domain / Range Kinds aggregation (grammars). Business domains (specific grammars templates).

Deployment

Deployment: Jena RDF / OWL backend. Parsed functional Metamodel. Nodes.

XSL Files (data, information, knowledge declarative schema-behavior bindings mapped into metamodel).

RESTful Message IO:
For each metamodel entity class: Resource metamodel class, patterns and domain specific templates with transforms encoded / parsed from metamodel.

Message flow: Filter handler (Servlet) request-response (async) through declarative template transforms.

Templates referring other endpoints (template aggregation). Apply templates (reactive async, request / response).

Topics. Streams (ordered events). Distributed contexts / scopes.

OWL upper ontology. Reified metamodel.

Protocol

The possible protocol to be regarded here is a kind of meta-protocol in the style of the OData protocol being implemented over HTTP. Thus, for example, will implement a representation framework (as HTML is for a ‘document’ web oriented application stack) via a dynamic hypermedia aggregation of path traversals.

It should to be able to retrieve knowledge, information, facts about ‘subjects’: 4D (where, when, who, state dimensions: categories / profiles). Patterns. Node responses. Index servive.

Hierarchical contexts provides query / assertion / reply interfaces. Graph / nested contexts (tree / lists) models. Paths. Node endpoints dynamically allocated into graphs according identity, attributes and contextual comparison dimensional alignments compose resolvable paths semantics. Registry service.

Dialog: ask / assert / reply pattern in context path location. RDF Message based representations. Dialog variables, placeholders. Pattern based subscriptions. Naming service.

‘Accounts’ (Consumer / Subscriber context, interactions): backend, user clients and agents consolidated and syndicated dispatch of ‘gestures’ (paths messages plus command statements) translated to any given protocols / IO. Context graph ‘reactive’ dataflow activation.

Paths / contexts / messages. State IO. Prompts (request / response) ‘runat’ lambdas.

Flows / scenarios (contexts / roles / state / transitions). Discovery. Subscriptions. Ontology alignment.

Protocol layer (over HTTP):
URI scheme (paths, node patterns, subscriptions HATEOAS HAL / JSONLD browseable applications).
Representations: RDF Message.
Command Statement (via HTTP methods defines how representation messages are handled):

Protocol 'semantics' (Discovery, Subscriptions, REST):
Subscription: REST Resource (feed / queue), declaratively stated patterns (Binding).

Nodes. Resources / Patterns. Data (Fact, Event), Information (Kind, Rule), Knowledge (Class, Flow) instance / schema dataflow activation (metamodel reactive IO).

Contexts. Endpoints. Paths. Hierarchical / graph aggregation of node resources identifiers. Resolvable 'patterns' to nodes in hierarchy according context and contents. ‘Posting’ (requests) occur in the scope of a context and ‘activates’ (async, message oriented) reactions with corresponding data, information and knowledge handled by the node.

Accounts (contexts). Dialogs (interactions). Subscriptions (data). Declaratively stated by 'patterns' (resource templates with 'paths': model data, application information and domain knowledge levels).

Protocol: submit 'paths' to 'paths' (functional semantics). CSPO Statements reified / encoded as 'paths'. Return 'paths', rel discovery by comparison alignment (referrer). CRUD is performed on the requesting side by means of returned results augmenting node metamodel. Intermediate requests augments requested nodes metamodel.

Example: from ‘Peter’ resource in ‘Employment’ (referrer context) to ‘Country’: all Peter’s Countries and the relationship with them (i.e.: countries where Peter has worked in).

Example encoding:

C: Context / Path (instance identifier aggregates SPO into pattern.

SPO: /subjectPath[path]/predicatePath[path]/objectPath[path]

De referenceable resource: aggregated Message (IO).

Domain translation: fulfill (dynamic) template.

Representations (requesting client metamodel resources) are built upon aggregating and aligning protocol dialog 'path' resources into data (Fact, Event), information (Kind, Rule) and knowledge / behavior (Class, Flow) in the requesting node, maybe by multiple ‘posts’ / traversals of activated contexts. Those are the same models which get 'activated' in the requested side by means of async messages IO.

Comparison result values holds metadata (ie.: 0 being equal, negative / positive values indicating encoded 'distance'. Octal results). Contexts (type / instance / attributes) occurring in a (temporal / revision) context.

Encode Functor / Monads, Function / Terms and Applications (flatMap, filter, etc.) as Resources (XSL Templates bindings).

Functional DOM (ASM / Upper ontology):
Functor (Type).
FunctorTerm (Member).
Application (Type instance members).

Reified Metamodels (upper ontology).

OntoClean. Primitives. Upper ontology. Bit map coding. Octal comparison results mask (order rel expressed in three bits). FCA Concept lattice.

(Fact1, Peter, wife, María);

(Fact2, Man, relationshipWith, Woman);

Patterns. Comparisons. Order / contexts. Comparing two facts (resources) in a given context must shield a third resource (alignment, discovery by distance).

Knowledge to compare Fact1 as instance of Fact2. Knowledge to compare two facts (in functional context) and sort them ('causeOf', 'after', 'before', 'partOf', 'equality' example contexts) according resulting resource / distance.

URI encoding (SPO instance, occurrence):

/resource[attr, val]/context[attr, val]/resource[attr, val]#instanceId

Example:

person[name, Doe]/country[code, ar]/employment[salary, highest]#johnJobs

Resources: comparable / function (Doe, highestSalaryJobs) : Jobs (high salary);

Context: referrer (Doe, highestSalaryJobs, Argentina) : Jobs (high salary, Argentina);

Application layer (Message encoded as RDF. Reactive dataflow). Bindings: Client APIs (DOM / JAF / DCI).

Node: Template methods. Implementation specifics of implementing declared subclasses of general / complex algorithms (query, services, functional). XSL Templates rely upon this.

Key / Value storage: (distributed) URI activation graphs. Map(Type, Map(Id, Val)). Functional applications.

ID: Encodes aggregated attribute / value of SPOs. Aggregated by instance ID (Context : ID): Relevant SID, PID, OID for CID.

(CID: (ID, ID)*, SID: (ID, ID)*, PID: (ID, ID)*, OID: (ID, ID)*);

RO: (Resource, Occurrence, Attribute, Value); Resource Occurrence.

(RO, RO, RO, RO);

Aggregated / Activation from RTL.

Peter, fact1, worksAt, IBM.

John, fact2, traineeOf, Peter.

Peter, fact3, promotionTo, seniorDeveloper.

John, fact4, replaces, Peter.

(fact1, fact2, fact3, fact4);

Kinds and patterns activation. Temporal / contexts order. Facts, Events, Kinds, Rules, Classes, Flows.

Resource Occurrences: Fact, Statement, Kind, Class.

Occurrence Statements: (Fact, Statement : Event, Kind : Rule, Class : Flow).

Type inference: aggregated attributes / values (resources / kinds), resource representation (parse statements / contexts: (700 / Units, Order / Product, Qty.)).

Resource statements (Facts): (aTravelFact, travel, distance, 60km); (aTravelFact, travel, origin, placeA); (aTravelFact, travel, destination, placeB);

Kind: (Kind, Fact, Class, Resource);

Rule: (Rule, Resource, Class, Resource);

Class: (Class, Kind, Attribute, Value);

Flow: (Flow, Value, Attribute, Value);

Fact statements (Events): (60km, placeA, distance, placeB);

(placeA, placeB, destination / origin, placeB);

(O, O, P, O) : Fact statements.

(For resource statements, kind, class, facts build: event, rule, flow).

Event: (60km, placeA, distance, placeB);

Rule: (Resource / Family, Resource / Peter, Class / Brother, Resource / John);

Flow: (Value / State, Value / Argentina, Attribute / Capital, Value / BuenosAires);

State is (an inferred) value for a class attribute of Argentina. Reified kinds / class / etc may play resource / attribute / value roles.

Contextual semantic markup: Resource, fact statements contexts aggregated from 'dialog' scopes (from representations of interactions with node resource protocol). Example: parsing a document or processing records from a database and emitting or 'aggregating' context from messages.

Purpose: Domain translation of problem spaces. Trays, flows. Messaging. Goal patterns. Prompts for state completion. Purpose goals driven domain use cases (DCI) application platform (services: data / schema for facts, information, behavior). Scoped contexts flows hierarchies.

Lambda: submit / response of 'runat' codat. Serverless / stateless: context state in each request

(representation, functional). Quad coding: GPU / AlphaRGB (TensorFlow).

Message driven. Publish / subscribe. Reactive streams. DCI / Metamodel declarative dataflow flows / activation graphs modelling (activation, events: facts / resource statements). Lambda functional encoding (functors) declarative (pattern / interface) matching of behavior and data. Sequences.

Appendix: Implementation details. Monads, DCI. Functional API

Apache Jena, Lucene, JCR, Stanbol.

Apache Metamodel. JBoss Teiid.

Cactoos.org (Declarative DSLs).

Node: Apache ServiceMix archetypes (RX Camel).

Monads:

Parameterized type M<T>.

Unit function (T -> M<T>).

Bind function: M<T> bind T -> M<U> = M<U> (map / flatMap: bind & bind function argument returns a monad, map implemented on top of flatMap).

Join: liftM2(list1, list2, function).

Filter: Predicate.

Sequence: Monad<Iterable<T>> sequence(Iterable<Monad<T>> monads).

Order by contextual comparator (registry).

DCI (Data, Context, Interaction):

Functional API: TBD.

Message (Flow): HATEOAS: JSONLD / HAL encoding.

ASM: Semantic ORM: (Dynamic Object Model, Actor / Role, DCI, JAF). DOM: Functional Functor (Type), Term (Member), Application (Type / Member instance binding).

Declarative discovery / activation. Reactive platform bindings (Java, JavaScript ASM) over RESTFul Client Node Resource endpoints.

TBD.

Apache ServiceMix / JBoss Fuse implementation

Factories: Apache Camel custom component: “metamodel:nodeType” like namespaces. Pipes (Resource hierarchy): Blueprint Camel contexts. Prefixes for each InOut endpoint, resource hierarchy: metamodel:fact, metamodel:kind, etc.

Metamodels: Ontology (Apache Jena backed) service implementations (for each type of backend). Aligned / augmented / aggregated repositories / registry for Factory Message exchanges. Example: DB / Service Backend. MetamodelService provides features. Service binding from route contexts. Archetypes for development.

Pipes (Message IO) between each Resource hierarchy layers. Factory contexts activate on inputs over Metamodel service. Index service. Routing (dynamic, languages) Aggregators. Transforms (via dynamic resource in context plus template): enrich / filter (ID alignment), normalize (attribute / link alignment), sort (context alignment).

Message (hier parent):
MsgID: URI (history).
Headers: CSPO URIs.
Body: DOM Document (deep / rel links). Parsed for parents, occurrences IO in subclasses.
Attachment: Message representation.

Metamodel: Resource(Message) mapping: URI: MsgID, CSPO: Headers, Parent, occurrences: parse body / aggregate, transforms pipes for CSPO, factory. Representation: Type handlers for activation, REST command maps from metamodel metadata.

Resource API:
Resource(Backend, T extends Resource):
getParent() : T
getOccurrences() : T super Resource
getFactory() : Context
retrieve(CSPO pattern)
from(pattern parent, pattern occurs) add / set parent
occurrences(pattern parent?) : subcls
match(parent, pattern)
apply(occurs ctx, pattern newOccur) : adds occurrence
history(pattern parent)

Implement functional methods via Message transforms (XSL Templates, XPath, XQuery).

Metamodel: Message(Backend); Message / Backend IO. Backend occurs: Messages. Backend parent: Endpoint / Connection.

Resource arg: Super class.
Resource occurrences: Sub classes.

Resource(Backend) : Message
Statement(Resource)
Model(Statement)
Fact(Model)
Kind(Fact)
Class(Kind)
Event(Class)
Rule(Event)
Flow(Rule)

Metamodel service: Export hierarchy interfaces. Flow occurrences: Message / Resource.

Example context:
from=”metamodel:fact”
to=”metamodel:kind”
InOut endpoints / route. Pub / Sub producer / consumer Rx / async.

Transforms: Custom component pipes (metamodel namespace) routes for resource hierarchy and custom metamodel service implementation. Enrich (aggregate), filter (ID), normalize (attrs.) and sort (ctxs.) in rel to ctx resource via functional Resource API.

Dialog example (fact / kind):

1. Fact - Kind (Fact w/o Kind aggregated)
2. Retrieve Kind occurrences in Fact (pattern) context (existing / created classes)
3. Create / retrieve matching Kind. Apply Kind to matching Facts (Kind occurs).

Component URI invocation example of Resource API:
metamodel:kind[selector]:fun(args / patterns)