Business domain translation of problem spaces (WIP draft)

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

Abstract:

Abstract. Keywords. Introduction (business domains integration use cases). Schema merge and interoperability.

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.

Considerations:

Given a business domain ‘problem space’ a ‘translation’ could be made (between other domains) which encompasses a given set of problem Objectives to be solved in one domain (maybe because of Events in that domain) for a requirement into another set of co-requirements in other domains which triggers new Objectives into its flow which shall be accomplished for ‘a’ global Purpose to be met.

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.

Features:

Although the proposed application mentioned in this document is thought to be implemented in an end to end full stack manner (from presentation through business logic to persistence) the core components meant to be used here are distributed and functional in nature.

Implementations of the framework shall allow for the discovery and publishing of profile driven endpoints.

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 to be able to work with them via some of the following features:

Achieve plain RDF serialization to-from any service / backend through Node Bindings (below). Ontology (schema) merge. Temporal, order (in contexts), Identity, Type and attributes / relationships alignment / augmentation.

Model statements into layers, each one having its own 'abstractions' for the roles each statement parts play:

Data layer: Infer type information, Classes / Kinds roles. Properties (domain / range role Kinds). From facts layer statements (data) infer which property kinds holds for which domain / range role kinds:

Who is a Person.
Who is a Business.
Which relations (properties) holds between them.

Information (schema) layer: Events which entail Flows given certain statements (relations) roles.

aPerson (hired), worksFor, someCompany (hires);

Knowledge (behavior) layer: Given certain Rules some Messages are ‘activated’.

betterJobOffering, employeeBehavior, jobSwitching;

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

Data example: aProduct, price, 10;

Information example: aProductPrice, percentVariation, +10;

Knowledge example: aProductPriceVariation, tendencyLastMonth, rise;

Features should be provided by the framework / datastore (alignment):

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

Resource endpoints. RESTFul interfaces for metamodels interaction. Provides / consumes (feeds / streams) events / messages declared via dataflow engine. The datastore should be a HATEOAS web application.

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').

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.

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:

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

Node:

Internal normailized metamodel (upper ontology) shared in common with other nodes regardless their purpose or role (binding, alignment, augmentation, etc.) instantiating an Application Services Model (ASM).

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.

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: determine interfaces needed to be implemented for a Node to fulfill its purpose / role.

Metamodel: classes / aggregation:

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:

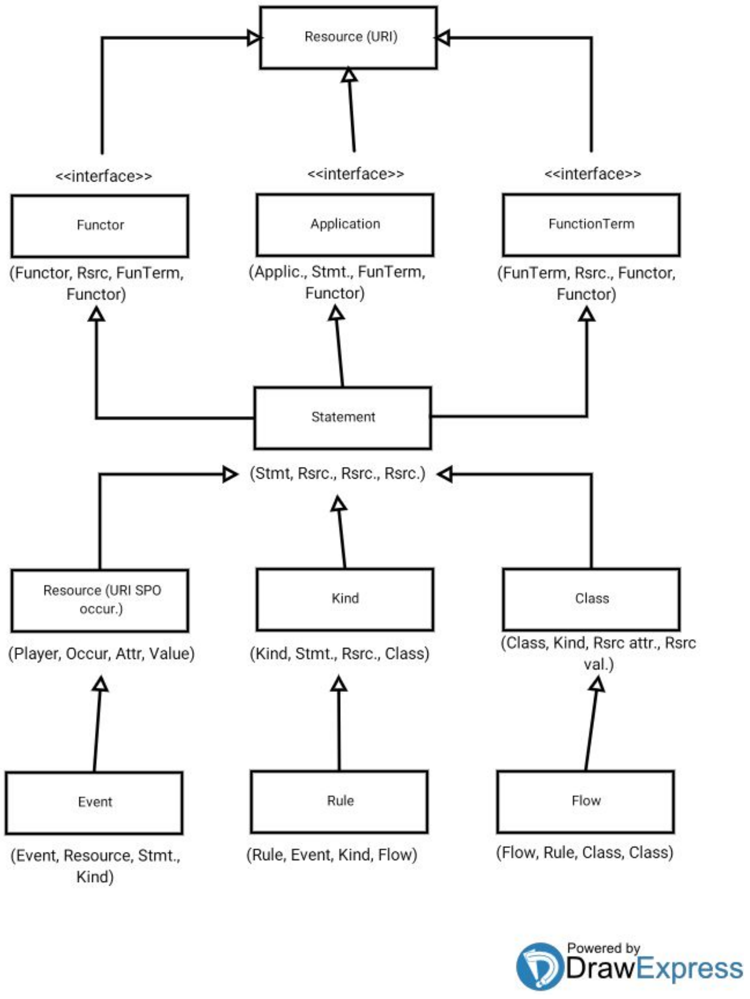
ClassName : (playerURI, occurrenceURI, attributeURI, valueURI);

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’ (different source quads). For example: a resource into an statement.

For whatever occurrences a player should 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.

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


Resource (URI)
<<interface>>
Functor
(Functor, Rsrc, FunTerm, Functor)
<<interface>>
Application
(Applic., Stmt., FunTerm, Functor)
<<interface>>
FunctionTerm
(FunTerm, Rsrc., Functor, Functor)
Statement
(Stmt, Rsrc., Rsrc., Rsrc.)
Resource (URI SPO occur.)
(Player, Occur, Attr, Value)
Kind
(Kind, Stmt., Rsrc., Class)
Class
(Class, Kind, Rsrc attr., Rsrc val.)
Event
(Event, Resource, Stmt., Kind)
Rule
(Rule, Event, Kind, Flow)
Flow
(Flow, Rule, Class, Class)
Powered by DrawExpress

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

TBD.

Metamodel: inference / layering / reification:

Type inference: basic type inference (for aggregation and syndication) 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, 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 later) have its own (Predicate based) SPO Sets definitions and, thus, their own Statement and Kind structures.

Kinds reification may be performed given, for example, 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).

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]

Model Layers:

The first Model (Facts) is built from raw input triples (SPOs) and its Kinds are aggregated according its Statements 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. Functional API semantics:

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

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

XSL API: Common transforms.

Functional API:

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

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).

Materialize / augment order rels, comparisons. (Complement, Start, Finish).

TBD.

Message IO: Encode Metamodel Flows:

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

Dynamic Object Model.
Data. Variables. Placeholders.

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

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 nesting (Flow):

(Flow (Rule (Event (Class (Kind (Resource))))));

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

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 endpoints (in / out Nodes). Interface methods (as templates in XSL over metamodel + Functional API) + Message logic for processing request / response. Declarative 'request / response': criteria over publish / subscribe. Filter messages.
Node: Bindings Node. Discover if Node has knowledge regarding Message. Emit knowledge regarding Message (syncing if necessary). Apply Metamodel + XSL transforms. Endpoints 'wrapping' datasource / protocols CRUD (Message encoded semantics determine operations, criteria and arguments).

Message IO: Publish / Subscribe:

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

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

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.

Message IO: XSL Based routing:

(reactive distributed dataflow activation graphs). Dialog protocol: augmentation / enhancements.

Node reactive dataflow integration setup (bindings, routes, transforms):

XSL Declarative reactive streams configuration: DCI (Message, Subscriptions, Processor).

Functional transforms declared as resource composition in XSL templates. Access Functional API from templates via resource representations.

Events pipes / routing. Resource 'activation': addressable. Pattern based. DCI. Cactoos.org.

RESTFul Message Metamodel layers Flows.

Node API:

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

Implement various Node (ASM / Declarative XSL) Kinds:

Node (Metamodel + XSL / Functional + Message API): Apache ServiceMix (like) archetypes (RX Camel).

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

ASM: Semantic ORM: (Dynamic Object Model, Actor / Role, DCI, JAF).

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

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

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

Binding Node archetype: backend syndication / sync Node implementation. RDBMSs / services / protocols datasources.

Alignment Node: ID / Merge, Order / contexts, Attribute / link knowledge enrichment and augmentation.

Service Node: index, naming, registry.

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

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 datasources.

Message (Flow) driven IO / ETL syndication / sync to - from any backend / datasource / service / protocol from which to obtain RDF statements IO.

Other Node(s) ‘conversations’ with this kind of Node fetch and publish knowledge encoded as Message Flows.

Align Node: ID / Merge

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

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

Naming services.

Align Node: Ordering / contexts

Contextual comparison augmentation. Temporal, causal, others.

Comparison results helps to augment other alignment methods.

Registry services.

Align Node: 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 Node:

Index services.

Naming services.

Registry services.

Learning services.

Inference services

Reasoner services.

Dimensional services.

Analysis services.

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. 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.

Appendix: Monads, DCI. Functional API

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.