Web of Data: Business domain translation of problem spaces. Semantic Business Integration (WIP draft)

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

Introduction

The goal is to streamline and augment with analysis and knowledge discovery capabilities enhanced declarative and reactive event driven process flows of applications between frameworks, protocols and tools via Semantic Web backed integration and Big (linked) Data applications enhancements. Perform EAI / Semantics driven Business Integration (BI).

Provide diverse information schema merge and syndicated data sources and services interoperability (for example different domains or applications databases). Translate behavior in one domain context into corresponding behavior(s) in other context or domains via aggregation of domain data into knowledge facts.

Aggregate knowledge into component metamodels which can interact between each other (databases, business domain descriptions, services, etc) into a dialog-enabling manner via dataflow / activation protocols which activates referrer / referring contexts with knowledge enabled from participating models.

Enable seamless integration of different data sources and services from different deployments and platforms to interoperate. A system A, for example, in the domain of CRM (Customer Relationship Management) and, a system B, in the domain of HMS (Healthcare Management System) would be able to be plugged into a ‘bus’. Then, actions (CRUD, service invocations) in one system should have ‘meaning’ for the other (potentially many) systems.

Keywords

Semantic Web, RDF, OWL, Big Data, Big Linked Data, Dataflow, Reactive programming, Functional Programming, Event Driven, Message Driven, ESB, EAI, Inference, Reasoning.

Current landscape

Document Web vs. Data Web.

The current Web (and the applications built upon it) are inherently ‘document based’ applications in which state change occurs via the navigation of hyperlinks. Besides some state transitions on the server side by means of ‘application’ servers, not much has changed since the web was just an ‘human friendly’ frontend of diverse (linked) resources for representation.

Even ‘meta’ protocols implemented over HTTP / REST are layers of indirection over this same paradigm. At the end we are all ending up spitting HTML in some form or another. And much of this seems like a workaround over another while we still trying to get some juice from ‘documents’.

The Web of data is not going to change this. And it’s not going to be widely adopted because the only thing it has in common with ‘traditional’ Web is the link(ed) part. No one will ever figure out how to build Web pages with ‘Semantic’ Web. It’s like trying to build websites with CSV files. That’s not the role in which SW will shine.

The Data Web

Semantic Web is a set of representation (serialization) formats and a bunch of (meta) protocols which excels for the modeling and accessing of graphs. Graphs of… well, graphs of anything (anything which may have an URI, at least). Let’s call a graph a set of nodes and a set of arcs between nodes, these are Resources. A triple is a set of three Resources: a Subject Resource, a Predicate Resource and an Object Resource (SPO: node, arc, node). A Triple may have eventually a fourth Resource, a Context, then the Triple (Quad) has the form: CSPO.

Now imagine a CSV file (or a database engine) that given this input files could relate it with a lot of other files, ‘calculates’ missing fields or figures out new ones. It may also figure out new ‘rows’. This is what Semantic Web has of ‘semantic’ and exactly what it has not of ‘Web’, in the traditional sense.

So, SW is a data exchange mechanism with formats and protocols. For user agent consumption it has to be rendered into some kind of document, like it is done for a (graph) database. But the real power of using the SW approach is for machine consumption. We’ll be using it that way in our example approach of EAI / Business Integration as a metamodel encoding facility which will entail aggregation and reasoning of business domains facts and flows. We’ll be using ‘ontologies’. An ‘ontology’ is for SW format representations as a database schema is for queries / statements only that this schema is modelled as SW resources as well.

Objectives

Inputs:

Syndicated datasources, backends of diverse applications databases, services interfaces (REST / SOAP / JMS, for example) should be aggregated and merged (matching equivalent records, for example) via the application of ‘Semantic’ metamodels thus providing via virtualization and syncing interoperability between those applications.

Features:

Once consolidated metamodels of the domains involved into the business integration process are available, services metamodels come into play providing alignment (ontology matching, augmentation and sorting), transformations and endpoint features.

Connectors: The goal, once source data / services are consolidated and aligned is to provide APIs for different languages / platforms which enable consumers of those data / services retrieve rich ‘augmented’ and enhanced knowledge that was not present in the original (integrated) backends.

Goals:

Given applications (big or microservices) there should be a way, given its schemas, data and behavior to ‘infer’ (align into a semantic upper ontology) what this applications do (semantically speaking).

Proposal / Solution

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 metamodels 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 meet a given purpose.

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.

Declaratively stated ‘purposes’ (an ontology of task related flows, roles and goals) should abstract producer / consumer peer interfaces for interactions in message exchanges. An ESB implementation of metamodels representing various domains of knowledge (databases, services, learning and inference engines) and message routing between them will exploit the ‘semantic’ capabilities while keeping ‘representation friendly’ consumer agent exchange mechanisms.

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) for semantically business integrated and enhanced applications.

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

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 and interact with them via some of the following features:

Feature: Data backends / services virtualization (federation / syndication / synchronization). Merge of source data and services.

Any datasources (backends / services) entities and schema regarded as being meaningful for a business domain translation and integration use case, regardless of their source format or protocol.

Business domain translation (dynamic templates). ESB customization features BI by means of abstract declarative layers of sources, processing and formatting of knowledge.

Feature: Schema (ontology) merge / match / alignment. Attributes / links inference. Contextual arrangement (sorting / comparisons). Metamodel services.

Diverse domain application data with diverse backend databases and services and diverse sources of business data (linked data ontologies, customers, product and suppliers among others) are to be aligned by merging matching entities and schema, once syndication and synchronization are available.

Examples: different names for the same entity, entity class or entity attribute. Type inference.

Identity alignment: merge equivalent entities / instances.

Attributes / Links alignment: resolution of (missing / new) attributes or links. Relationship type promotion.

Order (contextual sorting) alignment: given some context (temporal, causal, etc.) resolves order relations (comparisons).

Goal: ‘enrich’ applications actual services with domains knowledge. Query this knowledge by means of a dedicated endpoint for ad-hoc interaction contexts enhancements.

Goal: survey existing applications and components ‘semantic’ descriptors for integration in context of an ESB deployment which will provide their knowledge and behavior via ‘facades’ of those systems, which in turn will interact with the systems themselves.

MDM. Governance (client connectors). For schemas, data and behavior (plug in component / services for developing custom ad-hoc task apps integrated. platforms app devel toolkits).

Framework

Leverage existing solutions. Current needs / problems:

Current enterprise / business applications implementation technologies range from a very wide variety of vendor products or frameworks which handle different aspects of behavior and functionality needed for implementing use cases.

The current approach seems to be a ‘divide and conquer’ one. There is much effort being done in decomposing big applications into ‘microservices’ ones. But there remains being (small) black boxes which do ‘something’ (small). The semantics and discovery interoperations are left to the developer building apps that way.

The proposal here is to ‘syndicate’ (homologue) diverse systems backends and services for the sake of ‘knowledge propagation’ between diverse domains of diverse systems so behavior in one system is reflected in behavior in other domains.

Deployment

A Metamodel abstraction takes care of semantically represent and unify, by means of an API and interfaces, the different data sources, backends, services, features (alignment, augmentation, reasoning, etc.) that may get integrated into an ESB deployment. Each Metamodel is plugged into contexts with pipes of message streams with endpoints collaborating for each Metamodel to perform its tasks plus aggregation and other Node(s) knowledge propagation.

Messages abstract Metamodels state in ‘semantic’ form. A metamodel has an ontology of resources which get ‘activated’ from messages and ‘activates’ (fires) new messages.

Metamodels provide (via Messages):

Datasource, backend, service bindings: IO. Virtualization, consolidation / syndication. Synchronization and schema align and merge.

Core Functional Alignment:

Ontology (schema) merge. Type inference. Attributes and relationships alignment / augmentation. Relationship promotions.

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

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

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

Data, information and knowledge layers:

Data layer:
Example: (aProduct, price, 10);
Metamodel: TBD.

Information layer:
Example: (aProductPrice, percentVariation, +10);
Metamodel: TBD.

Knowledge (behavior) layer:
Example: (aProductPriceVariation, tendencyLastMonth, rise);
Metamodel: TBD.

Data: ([someNewsArticle] [subject] [climateChange]);

Information: ([someMedia] [names] [ecology]);

Knowledge: ([mention] [mentions] [mentionable]);

Metamodel layers:
Facts, Information, Knowledge.

TBD.

Existing deployed solutions could leverage of the benefits the stated approach. Existing clients and services could retrieve knowledge augmented data from a service in the context of their interactions.

Applications ‘plugged’ in this semantic ‘bus’, their processes could trigger or be triggered from / to another applications processes (maybe orchestrated by some domain translation declarative template).

Dados los mecanismos actuales de manejo de información y de la gestión de los procesos asociados a dicha informacion se pretende proveer a las herramientas actuales de medios aplicativos de la llamada gestión de bases de conocimiento que brinden insights en tiempo real tanto de análisis como de explotación de datos que ayuden a enriquecer con mejoras tanto la utilización del conocimiento como la toma de decisiones.

Infer business domain processes semantics and operations / behavior from schema, data and services (interfaces). An ontology (domain description schema) should be provided / interpreted for new or existing applications and services. Aggregated metamodels (events, rules, flows) based on existing or newly deployed behavior are driven from this schemas.

P2P: Purpose and capabilities discovery driven domain translation of business problem spaces. Enterprise bus of pluggable ontology domains, topics and peers providing features as backends (Big Data), alignment, rules, workflows, inference, learning and endpoints. Due the distributed nature of SOA (ESB) a P2P Peer in some form of protocol could be implemented via messaging endpoints. ‘Templates metadata’.

As mentioned before an existing deployed application could benefit from this framework integrating its data and services when plugged into the bus. It then may be enhanced using actual flows to consume augmented knowledge or being available to / for consumption by other applications or services.

Example

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.

Metamodels

A Metamodel renders schema / behavior / instances of data, information and knowledge layers for a given domain (connector). There are domain / schema Metamodels (Templates) and data Metamodels (Resources) aggregated from (meta) data of given domain connectors. Template hierarchies (‘code’: schema / behavior transforms) are aggregated from domains (connectors) instance data (Resources) and Runtime configuration.

A Runtime configuration only needs its URI (input / output of SPO statements) endpoints to be provided and perform merge aggregation and alignment of Resources and Templates (from DBs, services, etc.). Specific deployment configurations implements the needed interfaces for, for example, RDBMS, REST and other interactions.

For enabling different sources of data and services behaviors to be treated in an uniform manner a fine grained metamodel of such artifacts should be built / aggregated as for enabling description and discovery of functional marge, aggregation and ‘alignments’ which are the means for one element understanding each other element behavior (meaning, identity, attributes, contexts).

Functional monadic stream hierarchies

Metamodel instances of RESTFul HATEOAS endpoints streams. Templates (XSL) for endpoints message exchanges functional composition.

A Metamodel Resource is a (functional) monadic type which wraps a reference of its quad’s context resource URI (resource of which it has occurrences into quad’s subjects) and a (dynamic) list of its occurrences (parent / children).

TBD: Functional DOM. Monad interface (Type), Bound Function interface (Member), Application interface (Type / Members instances declaration). Functional Application / Binding Statements: (Application, Monad, Function, Monad); Bound functions. DCI.

TBD: Metamodel: semiotic encoding, reification, layers (aggregate inputs): Reify statements of the form (Statement, Classes / Metaclasses (kinds), SPOs). Aggregate into Metamodel(s).

Monad (Player, Occurrence)

Function (Attribute, Value)

Application : Monad, Function (Player, Occurrence, Attribute, Value).

Monadic functions: merge / apply / fmap / filter / query / transform / etc.

Domain functions: Reactive. Observable T. Events. Buses. Streams (Promise) types result function. Composition. Flows. Extractors.

URI : Endpoint. Representation. REST URIs reactive API. Uniform resource interfaces (DOM, HATEOAS HAL / JSON-LD. Activation).

The ‘class’ hierarchy of a Metamodel, modelled as RDF quads has the following pattern:

Classes and their instances are modelled as a quad hierarchy of OOP classes:

ClassName : (instance URI, occurrence URI, attribute URI, value URI);

A quad context URI (instance / player) identifies an instance statement (in an ontology) of a given OOP class. All ontology quads with the same context URI represent the same ‘instance’ which have different attributes with different values for different occurrences.

An instance (player) may have many ‘occurrences’ (into different source statements / quads). For example: a resource into an statement, a kind into a fact, etc.

Kinds / Grammars

Kinds: For whatever occurrences an URI may have into an statement (for example being the subject of one statement) there will be a corresponding set of ‘attributes’ and ‘values’ pairs determining (predicate and object, respectively, in this case), that the aggregated pairs of those occurrences, in common with other instances conforms the ‘subject kind’ of the resource, thus performing basic type inference. There may also be ‘predicate kinds’ and ‘object kinds’ with their corresponding SPO attribute / value pairs.

Upper ontology. Reified Metamodel plus encoding (IPs). Classes / Function / Kinds as Resources.

Grammars. Kinds Statement patterns.
Sustantivar (S); Verbalizar (P); Adjetivar (O); Pasión, Acción, Estado.

Upper ontology: exchange, merge, align. ISO 4D (object, time, place, attribute, value). Exchange of reified and instance metamodels (schema + instances). Vocabulary: semiotics sign + concept + language.

TBD: Grammars (class / type, metaclass / occurrence in kinds).

Uniform interface treatment of Templates / Resources

TBD: Metamodel abstractions are wrapped into Functional Programming ‘monads’ with reactive dataflow event driven semantics. Dataflow graph encodes dependencies / dependents for activation graphs tracking (input, state, output). Endpoint publisher / subscriber, subscriptions and exchanges semantics follow a DCI design pattern.

The ‘monadic’ type which homologue hierarchies is of the form:

Resource<Observable<T extends URI>> : T occurrences

T : URI. REST Semantics. HATEOAS / HAL, JSON-LD API.

Reactive functional uniform API (verbs). Activation, Representation (DBURIs, ServiceURIs).

Metamodel modeling: dimensional / tabular aggregation

Metamodels forms part of a Node knowledge regarding some domain. The typical scenario for building Template / Resource Metamodels is through aggregation of (meta) data coming from Node Connector instances.

TBD: Dimensional aggregation. X is Y of Z in W (contexts / dimensional statements modelling). Versions, contexts, scopes (access control). Attribute / links rel / scope (referrer context).

TBD: Tabular aggregation. Table, PK, Column, Value quads input.

Aggregation: Infer / aggregate Kinds / Instances of compound URIs. Reify higher layers statement quads into plain statements.

URI is the parent class of, for example, DB URI, Rest URI, SPARQL URI, Soap URI, etc. which are the types that, wrapped by a monadic Resource, provides the connection mechanisms to the underlying protocols.

Protocol: Dialog (events, commands).
IoC: Track dependencies / dependants. Flows: activation graphs. Encoded in Metamodel DCI Subscriptions (dependencies / dependants).

Class descriptions:

A Metamodel is anything that could be modelled or represented by URI statements.

A Statement (reified) represents a Context Subject, Predicate, Object (CSPO) structure.

A Topic is a set of Statements ‘talking’ about the same subject.

A Class is the aggregated set of Topics ‘talking’ about the same kind of resources.

A Behavior represents an abstract ‘action’ which may be carried.

A Flow is a concrete instance of a Behavior, for example a transition between Class attributes.

An aggregation algorithm is to be provided so it is possible to infer / aggregate topmost Flow(s) from original URI / Statement(s). Also, as Resource is a reactive designed monadic wrapper this algorithm performs in a publisher / subscriber manner and it is able to ‘react’ the underlying model given activation or inputs matching consumers which will propagate changes to its subscribers.

Metamodel hierarchies (Template / Resource)

Statement: Data / Data (Metamodel occurrence):
(Statement, URI, URI, URI);

Metamodel: Interaction (query) / Interaction (Topic occurrence):
(Metamodel, Statement, URI, URI);

Topic: Context (session) / Instance (Class occurrence):
(Topic, Metamodel, Statement, URI);

Class: Connector / Actor (Behavior occurrence):
(Class, Topic, Metamodel, Statement);

Behavior: Node / Role (Flow occurrence):
(Behavior, Class, Topic, Metamodel);

Flow: Peer / Performance:
(Flow, Behavior, Class, Topic);

Endpoints (publisher / subscriber) Dataflow semantics

Distance: Dimensional metadata of Functional comparison result in context. Comparable (order): X is Y of Z in W (contexts / dimensional statements modelling).

Dimensional (semiotic) statements pattern: reify X is Y of Z in W statements. A statement will hold references to this dimensional data in their occurrences.

Protocol: Functional Metamodel endpoints interface (REST / Activation)

TBD.

Metamodels: Resources / Templates

A Runtime (Peer library) JAR is to be configured with Templates (domains / ‘code’) and Resources (backends / ‘data’) streams which will be handled functionally to achieve the aggregation, alignment and inference needed mechanisms for the ultimate purpose of integration of disparate systems / services by functional merge of schema / data and behavior encoded in Metamodels.

Connector (API) and deployment environment (application server, services, etc) wrapper libraries are to be provided for a specific instantiation of a Runtime configuration into a knowledge-aware environment of sources and endpoints for consumption and provenance of semantic services.

Peer’s Template and Resource Metamodels (instantiated declaratively through configurations) streams are ‘merged’ via model composition giving insights on new knowledge and populating helper models (Semantic Object Mapping via upper ontologies, for example) to provide proper aggregation, alignment and inference.

Runtime:

TBD: Templates (Resource merge): Parser combinators / apply Resource as Function (transform).

Hierarchies (configuration) provider (publisher / consumer Resource / Template endpoints). Monadic wrapper hierarchy API for functional composition, activation (REST) and transforms of Metamodels.

Project archetypes / issues. Connectors.

Packages / APIs / Interfaces:

Blueprints (abstract bundle interfaces).

Endpoints (bundles).

Metamodels (Resource / Templates) are wrapped into a functional stream of (hierarchical) Messages which provide the means for accessing knowledge.

TBD: Protocols: Message interchange are wrapped into exchanges. Routes / Endpoint activation. Dialogs (reactive resolution of patterns / primitives: variables, wildcards, placeholders in streams).

Routes. Layers (contexts, domains, scopes).

Messages (activation / scopes / domains).

Templates (domains / dataflow subscriptions / contexts). Templates metamodels (domain / purposes) dataflow translation of domain behaviors.

Connectors (RDF IO: Activation, Patterns, CRUD / Versioning). Connection, Session, Dialog.

Deployment (library configuration): Messages, Endpoints, Protocols, Routes, Activation (reactive dialog), Templates, service providers / factory interfaces. Default / pluggable implementation of service providers. DSL. JavaScript. Templates (domains).

TBD: Transforms: declarative XSL stylesheets bound to domain’s exchanges.

Peer deployment: Container, configuration (Nodes), API / Protocol / Clients (Peers, Client Nodes). OSGi Bundle (blueprints), WAR / EAR (CDI / JAF / JCA), Spring, Jigsaw runtime (services), etc.

Metadata / Models. Inferred functions from domain. Functional ‘DOM’ (Data Object Model). Java / XSL Templates.
‘CRUD’ operations inserts new Resources (axes, contexts).

Metamodels Runtime (streams processing)

Functional application. Activation.

Templates (Resource merge): Monadic parser combinators / apply Resource as Function (transform). Build combinators for each Metamodel hierarchy layer that accepts Template (context) / Resource (data) as arguments. Retrieve Templates / Resources from Templates, Template (applicable) / Resources from Resources. Apply combinators from parent / child contexts to context childs. Outputs hierarchy of Resource / Template (applicables).

Templates generated from domain Resources knowledge aggregation (schema / roles / behavior inference).

Combinators: apply domain transform templates, aggregation, alignment (populate models).

Parser monads: Metamodel layers (of Resource of layer type) parsers. Parser result AST: Metamodel (Parsers of Resource hierarchy, combinators into Parser of Metamodel). Templates generated from domain Resources knowledge aggregation (schema / roles / behavior inference).

Resource Metamodel: equivalent to XML document.

Template Metamodel: equivalent to XSL templates.

Runtime: aggregate Resources Metamodel (Metamodel Message layers streams). Data.

Runtime: aggregate Templates Metamodel (Metamodel Message layers streams). Transforms.

Parser combinator (of Metamodel Templates / Resources monadic type. Outputs Resource Metamodel). Combinator Parser of Metamodel Template / Resource layers.

Parsers of Metamodel layers. Inputs: (context : Template layer, data : Resource layer).

1. Parser Match Template layers (hierarchically).

2. Parser Match Template layers Resources (hierarchically).

3. Parser Match Resources applicable Templates (recurse to 1).

Combinators: apply aggregated domain transform Templates, Alignment (populate models).

TBD: Templates (Resource merge): Parser combinators / apply Resource as Function (transform).

Monadic wrapping of hierarchy elements allow for the application of the above alignment mentioned methods by means of functional composition.

- Core functions:
- Aggregation functions.
- Alignment Type / ID (attrs. promotion) inference algorithm.
- Alignment Links / Attributes inference algorithm.
- Alignment Context / Order (class hierarchy expressed as an order relation) inference algorithm.
- Getters CSPO (extractor functions)
- Wrapped URI HATEOAS / HAL, JSON-LD Functions.

By means of monadic application of different alignment functions, Ontology Alignment and Algorithmic Alignment may be used, the first to perform the desired alignment operation on Resource(s) and the later for the comprobation of correctness in the reasoning / inference process.

Core alignment functions are: Identity alignment (to identify equivalent entities), Link and attribute augmentation and contextual ordering / sort function. Also CSPO 'getters' functions and RESTFul HATEOAS functions (by means of HAL or JSON-LD) should be provided.

Because Metamodel(s) aggregate knowledge in such a way, 'model' functions may exist which, for example, reifies the application of a Behavior to an entity instance (a Flow).

Semiotic Function: Languages: terms, semiotic mappings. Augmented dictionary (a is b in ctx for d). Grammars (ontology Kinds aggregation).

Resource Functions: repository of functions (Dataflow specs, XSL DSLs). Core Functions / declarative functions namespace for retrieving from exchanges and templates.

- Distance: Dimensional metadata of Functional comparation result Resource. Comparable (order).
- Apply (fmap): Function<T, M>
- Sequence. Apply F to list.
- Chained functions over monadic wrapper (getAddress, getStreet).
- Pipelines / Flows (Promises).
- Predicates.
- Selectors (patterns).

- Streams operations.

- Functional domain
- Functions available from Java / XSL / Camel templates.
- Resource<Function<T, M>>
- Runtime: merge Resource streams (URIs, Functions).

Monads: everything as a Resource (of Observable of aggregated T). Unify treatment of data coming from any datasource into streams of aggregated layers from source data.

Activation (query / match). Monadic transforms.

Reified Metamodels (upper ontology templates). Primitives, roles. Bitmap coding. Quad encoding. FCA. TBD.

TBD: Functional DOM: Monad interface (Type), Bound Function interface (Member), Application interface (Type / Members instances declaration). Implemented by Resource, Statement and Kind (Flow, Rule, Class inherits).

Functional selectors. Dependencies. Functional selectors. Dependants. Activation.

API Interfaces: Template methods (implementing hierarchy).

Type / Attribute promotion (relation): Ontology templates.

Domain translation: (dynamic) templates.

Endpoints / Dataflow

Metamodel resources (Resources / Templates) follows a reactive Publisher / Subscriber (domains) pattern as Endpoints. Runtime configuration. Streams. Dataflow.

TBD: Routes are determined by aggregation and knowledge domains (protocols / scopes / contexts). Transforms.

Merge / Apply: Parser combinators (Templates / Resources). Runtime Dataflow.

Resource API: Activation. Reactive. Dataflow.

Metamodel representations interfaces: Browseable hypermedia (REST HATEOAS HAL / JSON-LD).

Aggregation

In the context of an exchange of dataflow messages aggregation of Metamodel layers is performed.

Merge

In the context of an exchange of dataflow messages merge (functional composition of Templates and Resources) of Metamodel hierarchies is performed.

Alignment

In the context of an exchange of dataflow messages alignment of Metamodel layers by algorithms is performed.

Message Exchanges

Protocols: Routes / Dialogs / Scopes

TBD: Dynamic routing of Messages due to Observable pattern. Contexts, scopes, sessions (dialog) domain namespaces.

Transforms

Domain driven ‘code’ (Java, JS, XSL) applying composite functional transformations.

Functional application of Metamodel transforms (streams). Alignment, Augmentation and inference algorithms / models application.

Functional query:

Functional transform:

Metamodel: Template / Resource hierarchies (message quads):

Statement: Data / Data (Metamodel occurrence):
(Statement, URI, URI, URI);

Metamodel: Interaction / Interaction (Topic occurrence):
(Metamodel, Statement, URI, URI);

Topic: Context / Instance (Class occurrence):
(Topic, Metamodel, Statement, URI);

Class: Connector / Actor (Behavior occurrence):
(Class, Topic, Metamodel, Statement);

Behavior: Node / Role (Flow occurrence):
(Behavior, Class, Topic, Metamodel);

Flow: Peer / Performance:
(Flow, Behavior, Class, Topic);

Messages:

Entire hierarchy encoded (Resources / Templates). Streams (from messages): Template / Resource streams. Apply entire message from Template to Resource / from Resource to Template (i.e.: Behavior application over a Metamodel results in an Statement stream).

Layer encoded aggregation of transform / merge changes ‘propagation’ (stream events).

- Dialog. Protocol (schema / instances). Patterns, wildcards, placeholders. Layers (sync):
- Data
- Information
- Knowledge

Dataflow / Runtime transform hierarchies (configuration / streams):

Subscribe (recursively) to Template hierarchy Template streams from Runtime and to each Template Resource sub streams (and recursively observe Resource Templates). Apply parent Template message to observed Resource streams. Applied hierarchies result into new streams.

Runtime streams (messages: protocol / routes via Observables): Templates, Resources. Lambdas:

a) Apply Template to Template Resources stream: Template stream.
b) Apply Resource to Resource Templates stream (populated by configuration and a): Resource stream.
c) Apply (merge streams) a / b corresponding result Resources, Templates (apply a / b again): Result Resource / Template streams (populate further Template / Resources streams).

Streams: Observable / publisher with corresponding protocol / layer, instance, context / scope qualifiers for Runtime contexts streams arrangements (i.e.: Template / Resource lambdas).

Template / Resource streams for each context.

Template / Resource streams for each aggregated parent.

Template / Resource streams for each occurrences of Resource / Template in each other (Resources matching Templates, Templates applicable to Resources.

Dataflow / Runtime transforms (exchanges / apply merge):

Session / Dialog API. Facade / APIs (Class / Behavior / Flows).

Apply aggregation.

Apply alignment.

Populate models.

Lambdas example:

Template(cook);
Resource(Ingredients):
Template(serve);

Resource(ingredients);
Template(cook):
Resource(meal);

Metamodels: Models (grammars, object, others), Services (resolution), Algorithms (Alignment / aggregation of hierarchies): Runtime configuration (Templates / Resources interfaces sources / APIs: i.e.: ML) transform / apply at Message level (interactions) in dialog contexts.

In the context of a Data dialog given matching Information a Knowledge template could be matched which activates a Rule Flow (pattern / transform) which updates players LHS with RHS.

Purpose Metamodel (Templates): Task accomplishment services / QA. Over Domain Metamodels. Over Backend Metamodels.

Purpose Metamodel (Templates): aggregation ontology, layer scopes (facts, concepts, roles / contexts : Data IO, dialog state, behavior templates).

Layer scopes for Purpose Metamodels:

Purpose Metamodel (Templates): Connector populates behavior templates layer from dialog state from previous facts and other Metamodels. IO activates dialog state to / from behavior templates and aggregation augments roles / contexts. IO parses / renders facts aggregated to / from dialog state in respect to behavior templates contexts.

Purpose facts (Resources) Connector: Connection IO (render hierarchical flows, prompts, confirmations from dialog state relative to current inputs) into facts to / from dialog state in respect to behavior templates (NLP Connector).

Templates (aggregated knowledge) determines what to prompt for input and what output facts apply for the current dialog (Information) ‘session’ (Data). Other Metamodels augments and get augmented from these interactions (retrieve database records, invoke services, alignment ‘interprets’ user input) thus integrating QA into a broader set of integration use cases.

Current input fact in respect to dialog context resolves next behavior template to be populated into dialog context (question / question, question / answer).

Current output fact in respect to dialog context resolves to expected behavior template(s) to be populated into dialog context (answer / question).

Dialog state session mediates between facts and knowledge in question / answer scenarios (hierarchical flows, prompts, confirmations).
Purpose Metamodel Connector (NLP, parser / renderer) handles representations of dialog facts 'questions' and 'answers' in the context of available question / answer sets (behavior templates knowledge) in the context of a dialog session.

Normalized message format which may encode / activate behavior in any route(s) layer.

Messages: Protocol agents (reactive activation / paths routes). Referrer. Representation. (browse / aggregate). Encode Resource (aggregation). Routes.

Messaging: Messages flow through dynamic routes between Component Metamodels and Components. Synchronization and propagation of knowledge by means of activation.

Message: Protocol agent. Encode routes / payload for different Component Metamodel layers (endpoints: Semantic URIs):

Message: Encodes Component Metamodels routes / payloads for a given activation. Example: Purpose occurrence corresponds to domain / backend occurrence. Compares ‘distance’ (alignment) calculating next routes / payloads to be delivered.

Message: Determines next routes / payloads for its Component Metamodels routes / payloads. Aggregates activated knowledge (routes / payloads) in its exchanges (browsing / discovery agent). Relative to its ‘referrer’ (Context: ‘Developer’, ‘Peter’, ‘Work’ -> ‘Peter Working as a Developer’).

Message: Metamodel instance (Data routes facade, Information and Knowledge payload body / template). Activation / aggregation determines Message facade state for further routing /

payloads. Goal: fulfill specific template.

Domain Flows (scenario / context, roles, state, transitions). Ontology aligned Messages. DCI: Data (Metamodel, Message), Context (sharing scopes, subscriptions, roles inferred from data / referrer), Interactions (declaratively stated in routes APIs).

Algorithms / Models

Aggregation, Alignment, Inference / Entailment.

Alignment of ontologies is performed via different Representations (ontology, algorithmic, aggregation alignment) plus grammars / object model / upper ontology (via Facades which holds Representations) of a Metamodel.

- ID / Type align augmentation: ResourceID patterns Functions
- Resource.fmap(AlignFunction<Resource, Resource>).fmap(MergeFunction<Resource, Resource>) : returns merged Resources.

- Order in context: ResourceIDs order in context / role (attribute, value axis) Functions
- Resource.fmap(OrderFunction<Resource, Resource>).fmap(ContextFunction<Resource, Resource>) : returns first.

- Attribute / Link: augmentation Functions
- Resource.fmap(TypeFunction<Resource, Resource>).fmap(AttributeFunction<Resource, Resource>) : returns attributes.
- Semiotic Function: Languages: terms, semiotic mappings. Augmented dictionary (a is b in ctx for d).

Models:

Grammar models.

Alignment models.

Object models (type object upper ontology).

- Upper ontology. Reified Metamodel plus encoding (IPs). Classes / Function / Kinds as Resources.
- Grammars. Kinds Statement patterns.
- Sustantivar (S); Verbalizar (P); Adjetivar (O); Pasión, Acción, Estado.

- Metamodel.
- Alignment models: Ontology, Algorithmic, Aggregation.
- Grammars (primitives / objects: ontology alignment).
- Services: Resolution. Index (identity, ‘content’), Naming (links / rels, IDs), Registry

(context positions).
- Object model (dynamic type object upper ontology / ISO / OWL).

- Peer interface (URI / Metamodel level).
- Connector interface (Statement level).
- Node interface (Topic level).
- Facade interfaces (APIs). Class, Behavior, Flow levels.
- Grammar models.
- Alignment models.
- Object model (ISO / Type object upper ontology).

- Metaclass: classes this class is attribute of.
- Class: attributes.
- Instance: attribute values.
- Kinds metadata: Class aggregate attributes. Class aggregates attributes values. Instances of a class summarization by ML models / seed. Resource IDs.

- Metamodel reified: URI : Metamodel. Metamodel statements parent, attribute, value : Metamodel. For each URI hierarchy resource statement Metamodel statement. Alignment / Mapping upper ontologies.
- Normalize URIs hierarchy:Normalize Metamodel hierarchy into Resource IDs aggregable statements.
- Reify Metamodel hierarchy into Resource IDs aggregated statements.
- Infer / aggregate Kinds / Instances of compound URIs.
- Reified Metamodel statements now could be aggregated for Resource ID inference.

TBD: Resolution.

Ontology alignment

Objective: Merge (align / blend) graphs with equivalent ‘records’ without common keys algorithm. Identity resolution / merge.

Datasource interoperability

Normalize input statements (URIs):

Translate any datasource to the following tabular roles:

Tabular roles: (Table, PK, Column, Value);

Statement roles: (Player, Occurrence, Attribute, Value);

Traversing quad elements:

Previous element in quad: metaclass.

Current element in quad: class.

Next element in quad: instance.

Aggregation: Same URIs parent / child of previous / next URIs.

Ontology alignment: (Metaclass, Class, Instance, Occurrence);

Instance (next): current role URI / previous role aggregated URIs. Final: datatype / primitive. Example: age.

Kinds (previous): current role same URIs / next role aggregated URIs. Example: Person. Final: Top (Object)

Alignment roles: (Kind / Instance, Kind / Instance, Kind / Instance, Datatype / Primitive).

From 2 datasources the final (object) datatype / primitive 'aligns' sources statements (1 and 2 different sources to be merged):

Alignment roles 1: (Carpenter / Guiseppe, Grandparent / Giuseppe, Parent / Peter, Person / Joseph);

Alignment roles 2: (Carpenter2 / Giuseppe2, Grandparent2 / Guiuseppe2, Parent2 / Peter2, Person2 / Joseph2);

Context Kind / Context Instance: should be in both sources as the value of the player role of the statement. Metaclass of subject Kind.

Sliding (disambiguation):

Datatype / Primitives: regarding Context: Carpenter / Profession ‘slides’ to subject position, Sex / Male occupies context (sliding till reach primitives).

Chained Kind (SPOs): Identify lowermost (top superclasses, objects) of Kind hierarchies. Aggregate Kind / sub Kind hierarchies. Sub Kinds: attributes are superset of those of super Kind.

Assign instance, class, metaclass and player roles. Player role: Datatype / Primitive or Kind / Instance.

If player role ends in Datatype / Primitive alignment could be performed.

If player role ends in Kind / Instance a new statement should be made ‘sliding’ player to class and recalculating hierarchies.

Match Datatype / Instance are the same values: Training. Mappings: learn hierarchies for known matching players. Semiotic alignment (sign is object for context in concept).

Semiotic Function: Languages: terms, semiotic mappings. Augmented dictionary (a is b in ctx for d).

Match Datatype / Instance are the same values: same Kind / Instance hierarchies taxonomy (graph shape). Abstract graph of models nodes (Kind roles: previous, current, next). Merge aggregated graphs augmenting Kind with attributes (Instance lattice). Attribute similarity function.

Alignments (merge):

- ID / Type merge: matching resources: same entities.
- Link / Attribute merge: matching resources sharing unknown attributes in context augmentation.
- Order / Context merge: matching resources with 'wildcards' in contexts / CSPO axes which respect Kind / Instance hierarchies. Wildcards will be provided with a comparator function.

Reifying Metamodel: Normalize Metamodel hierarchy into Resource IDs (Metamodel) aggregable statements.

Update:

Ontology align: aggregate inputs

(class / instance / metaclass, class / instance / metaclass, class / instance / metaclass, class / instance / metaclass);

Context: objects hierarchy; Object: primitives; classes / metaclasses: kinds; Identity merge: equivalent objects / primitives (kinds) shapes / paths. From equivalent primitives traversal (slides) to equivalent objects.

Class: Kind relative to Resource attributes / values.

Metaclass: Kind relative to Resource (class) occurrences as value of attribute of other Resource(s).

Ontology example:
(class / "Computer Supplies" / metaclass, class / "Notebooks" / metaclass, class / "Lenovo" / metaclass, class / "Lenovo XYZ" / metaclass);

Ontology example (slide left):
(class / "Notebooks" / metaclass, class / "Lenovo" / metaclass, class / "Lenovo XYZ" / metaclass, class / "Lenovo XYZ S/N 1234" / metaclass);

Ontology example (slide right):
(class / "Inventory" / metaclass, class / "Computer Supplies" / metaclass, class / "Notebooks" / metaclass, class / "Lenovo" / metaclass);

Algorithmic alignment

Objective: Succinct representation of quads for graph encoding / exchange and validation of ontology alignment.

Objective: infer links / rels missing or calculated through ontology.

Objective: encode URI statement quads into numeric quads (ResourceIDs) for, for example, Machine Learning algorithms or virtual network addressing (IPs) for resources.

The idea is being able to use a fixed length quad of fixed length components being able to 'reuse' a same numeric identifier with another meaning in another quad / component position. This could be used for virtual network addressing of statements for enabling protocols and operations.

As quads components are themselves metamodels (quads) this process could recursively encode a vast amount of information in a reduced fashion and enable, by the use of an identification algorithm, the use of known Machine Learning tools for processing large amounts of data.

Statement (C, S, P, O):
(meta:(super:(class:inst))

- inst: Primes sequence count (InstanceID)
- class: InstanceID product by corresponding InstanceID next primes (ClassID).
- super: ClassID product by corresponding InstanceID next primes (SuperID).
- meta: SuperID product by corresponding ClassID next primes (MetaID).
- Next prime is in relation with parent prime: parents aggregate child primes count.
- If element repeats, same prime is used as in first occurrence.

- Example
- (Carpenter, Grandfather, Father, Person);
- (FirstBorn, Brother, Son, Person);
- FirstBorn and Carpenter shares Person factor(s). Different identifiers in different contexts will preserve this characteristic.

- Reduce quad products disambiguating common factors.
- Primes may be mapped to their position in primes list: (1, 2), (2, 3), (3, 5), (4, 7), (5, 11), etc. Encode / decode from mappings for succinct representation.

Ontology Alignment from Algorithmic Alignment:

Reify Metamodel hierarchy into Resource IDs aggregated statements. Each Metamodel subclass entity has its corresponding (normalized) aggregable Metamodel statement.

Inferir / aggregate Kinds / Instances of compound URIs.

Example:

Topic: (Statement, URI, Metamodel);

Topic: Amor (Persona / Metamodel, Amante / URI, Pareja / Statement);

Reified Metamodel hierarchy statements now could be aggregated for Resource ID inference. Alignment with (upper) ontologies could be done using ID, Link, Context ordering functions.

Aggregation alignment

Objective: Context alignment (resolve x is y of z in w); IO / Activation / Reasoning.

Encoding (occurrences: context, player, attribute, value): S(P, O); P(S, O); O(S, P); C(S, P, O); Shapes / grammars (kinds).

Resolution: (name, player); (context, attribute); (index, value); IO / Activation / Reasoning.

Facades. APIs.

Facades. Interfaces (functional declarative queries: schema / data / behavior). Metamodel encoded queries. Queries API translation by Peers alignment. Resolution services.

Client (Facade) Models: APIs. Transforms from Facade interface queries (Client Metamodels).

Object model (type object / upper ontology). APIs. Facades. Grammars.

APIs:

Facades: Metamodels / Models.

Metamodel / Templates. Endpoints (publish / subscribe layers Messages). Transforms (exchanges):

Metamodels hierarchies: Templates, Resources. Stream through Templates hierarchically, then through for each Template hierarchy. Resources subscriptions via Templates.

Interface Endpoint (publisher / subscriber: protocols Peer, Node, Connector, Facade, etc. methods / instance, context, scope).

Interface Message (RDF / aggregated Metamodel / scope, context). Template / Metamodel hierarchy propagation (correlations).

Interface Exchange (Templates / protocol scope / context Metamodel transforms / routes, publish).

Runtime: Endpoint publish / subscribes to Endpoint streams (Peers, Connectors, Nodes, etc. Templates hierarchically), Messages, Exchanges sub-streams / protocols (DCI, Dialog protocol). Configure Peers, Nodes, Connectors, Facades, etc. Streams activates when event occurs in Endpoint / child (Peer / Node when Node Message publishes). Dataflow: Facade(s) subscribes to related events (Metamodels) according Templates.

Models.

Aggregation, alignment and inference (exchanges).

Solution architecture

Solution architecture Federated domains back end virtualization. Aggregation, Alignment, Inference. API interfaces for plugin schemes of sources and clients.

The proposed solution architecture is the implementation of custom extensions to an ESB (Enterprise Service Bus) platform which allows for the deployment of custom URI(s) (DB, service, connector, etc.) implementations metamodel components.

By means of declarative 'templates' the platform will be allowed to specify and compose diverse source / service components (local or remote) enabling them to interact with each other. Interactions will implement the 'domain translation' pattern thus enabling gestures / behaviors in one domain to be translated into corresponding actions into other domains.

Declarative DCI Templates: Java / XML - XSL implementations (www.cactoos.org).

Layers (schema / instances):

Data
Information
Knowledge

Protocol:

Routes
Dialog

Activation Graphs (reactive)

Strategy patterns in functional composition.

Server less (distributed command message pattern).

Integration with documents / office platforms. Forms. Templates.

Monads: TBD.

Type constructor.
Unit.
Map / flatMap
Sequence (apply to list)
Chaining of mappings of functions.
Predicates.
Selectors (patterns).
Stream operations.

Functions:

Domain (Resource(s) URI hierarchy).
Namespaces (Java / Templates).
Resource(s) as functions: Inferred functions from domain. runtime merges function / Resource streams.

Activation

Dataflow

Functional ‘DOM’ (Java / Templates)

CRUD: Immutable Resource(s). Changes creates new entities in new contexts (axes).

Implementation roadmap:

Core Bundles: Peer. Node. Metamodel (messaging, blueprint, containers).

Core Peer. Customized Camel routing messaging schemes / exchanges (APIs plugins / templates).

Core Node. Blueprint declarative API. Endpoints (backend / connectors) plugins.

Core Metamodel. Align. Node's model container (Peer's / Node's Message dispatch: reactive activation dataflows).

Java / ServiceMix. XML / XSLT. Javascript (Node.js Bundles) plugins / templates.

Deployment

Runtime. (client) Connectors. Wrappers (facades / containers).

Deployment use case

Library (Peers). Connectors (implementations). Wrappers (deployment). Peer runtime configurations.

Core Bundles: Peer. Node. Metamodel (messaging, blueprint, containers).

Java / ServiceMix. XML / XSLT. Javascript (Node.js Bundles) plugins / Templates.

Deployment of the solution is aggregated into three kind of nesting entities scopes (ESB bundles): Peers, Nodes and Metamodels which, through messaging, templates and endpoints constitute the solution proposed runtime.

Templates are declarative entities which may be ‘plugged’ in any message exchange at any entity level scope (Peer, Node, Metamodel). Implementation may vary from Java classes implementing some interface, XSLT templates or ‘special’ Node.js bundles.

Templates are plugged declaratively through Node(s) declaration blueprint or may be registered and invoked by an event filtering mechanism. Templates may use any of the Functional namespace registered operations / alignments and also ‘selector’ predicates over Resource monadic type.

Template(s) metadata is the notion of a domain business, its data, information, knowledge and behavior schema / instances represented as a ‘purposes upper ontology’ Metamodel.

Scopes, domains, purpose, etc. Metadata share / merge (TBD).

Alignment, Functional domains and aggregation are performed through templates.

Peers:

Core Peer. Customized Camel routing messaging schemes / exchanges (APIs plugins / templates).

P2P Bundles. Aggregates Node(s). Define custom ESB URI scheme for handling of routes and exchanges of entity scopes layers.

Discovery. Advertisements. Dynamic bindings (containers) of other scopes entities. Peer scope (dynamic) routes (topics / queues + templates). URI namespaces.

Peer Templates: ‘hooks’ according each exchange semantics. Inputs, functional transforms, outputs (TBD).

Nodes:

Core Node. Blueprint declarative API. Endpoints (backend / connectors) plugins.

Endpoint / Peer binding scopes. A declarative blueprint in the ESB instantiates a Node which, through an Endpoint, populates / sync and provides reactive activation of a Metamodel.

Namespaces: Node ‘scopes’ which determines metadata of, for example, domains of business of Node’s Metamodels for enabling metadata interoperability (templates metadata).

Scopes (Message flow IO):

Peer namespaces.
Node Connector (Endpoint).
Metamodel Connection.

Connector (Domain. Template metadata).
Session (Connection).
Dialog (Metamodel).

Metamodels:

Core Metamodel. Align. Node's model container (Peer's / Node's Message dispatch: reactive activation dataflows). Facade.

Metamodels are the internal representation of an instance of a Node regarding one Endpoint as declared into a Peer.

Instantiation. Factory. Population. Aggregation. Activation / Sync / IO (TBD).

Messages / Event IO (metamodel layers, peer components, functions / alignment). (TBD).

Templates: activation routes. Template API spec over ESB routes (class / pattern). Transforms, filter, etc. Functions available in contexts (Java, XSLT, Node.js).

Function encoded as Resource. Parser. Materialized results (patterns / grammars).

Function: resulting Resource semantics. Contained result Metamodel type. Concat function expressions.

Endpoints (backend connectors / client connectors)

APIs: JDBC / JAF / JCA. DCI. REST HATEOAS (HAL / JSON-LD).

Kind of ‘drivers’ for Node Metamodel(s). IO Connectors (backend / client) interactions modelled as service interactions (homologous). Rendering services (bundles) for UX.

Messaging: handle sync / knowledge propagation (TBD).

Connector examples:

JBoss Teiid endpoint.
Apache Metamodel endpoint.
RDF/ OWL / SPARQL endpoint.
REST endpoint (HATEOAS, protocols).
SOAP endpoint.
LDP / SoLiD endpoint.
Java EE (JCA / JAF).
JavaScript (browser).
JavaScript (Node JS).
PHP.
.Net (LINQ).
RDBMSs.
LDAP / JNDI / JCR.
Streams (Big Data).
BI: OLAP / Mining.
Indexing.
Machine learning
Rules engine.
QA.

Peer, Node, Metamodel Template API

Message exchanges between scoped entities (Peer, Node, Metamodel) handled by custom URI namespaces (defined at Peer level) are ‘augmented’ by ‘wrapper’ Template(s) regarding the inputs, transforms and outputs / routing results of the exchange.

Activation and Dialog Protocol: Messages ‘activates’ Metamodel Resource(s) in respect to matching of input patterns, which produces into the Message exchange response the ‘activated’ Resource(s) to be provided by means of Template(s) manipulation.

A ‘Dialog Protocol’ is established in the connection / session / dialog scopes Message exchanges which allows for wildcards, variables and placeholders to be progressively ‘populated’ in a request / response cycle. (TBD: Functional alignments, aggregation and patterns).

Normalized message format which may encode / activate Resource / behavior in any route(s) layer. Message determines routes itself. Referrer semantics (contexts). Aggregates activated knowledge (routes / payloads) in its exchanges (browsing / discovery agent). Relative to its ‘referrer’ (Context: ‘Developer’, ‘Peter’, ‘Work’ -> ‘Peter Working as a Developer’).

Message: Metamodel Resource instance (Data routes facade, Information and knowledge payload body / template). Activation / aggregation determines Message facade state for further routing / payloads. Goal: fulfill specific template.

Template metadata: Upper Ontology. Domain Flows (scenario / context, roles, state, transitions). Ontology aligned Messages. DCI: Data (Metamodel, Message), Context (sharing scopes, subscriptions, roles inferred from data / referrer), Interactions (declaratively stated in routes APIs). (TBD).

Dialog Protocol

Scopes. Wildcards. Variables. Placeholders. QA request / response.

Domain Metamodel. Connection driven. Shared (scope context attribute). Template metadata driven exchanges.

Purpose: Notion of Template metadata upper ontology Metamodel. Shared (scope context attribute).

Metamodel layers: Data (Resource, Flow), Information (Kind, Class), Knowledge (Statement, Rule). FIXME. TBD.

Activation: Dataflow graph from Resource parents / occurrences. Dialog flow (apply / templates).

Activation: Dependencies. Selector. Events.

Activation: Dependants. Selector. Events.

Messages, Routes, Activation / Templates

Between Peer(s), Node(s), Metamodel(s). TBD.

Topic / Queue / PTP. P2P Transparent.

Distributed Dataflow Activation Graph:
Input, State, Output graph (dependent / dependants). Encoded in Metamodel DCI Subscriptions (dependencies / dependants).

Location transparent (Docker, Kubernetes). Dynamic IP space mapped to CSPO Resource. Algorithmic ResourceIDs as RESTFul addresses.

Events.
Command pattern.
Normalized Message Resource Monads.
Protocol 'agents': activation, routes, payload is Message encoded logic.
Sync: listeners (dependencies / dependants).

Change propagation: aggregate (ctx:msg) tuples for which if a tuple holds the other tuples must hold too.

Purpose / Metadata (domains): Change propagation + Template metadata.

Semantic activation patterns:
Event source (criteria)
Event destination (profile)

Dialog: Placeholders, wildcards, variables: resolve from context (previous named field inputs, referrer, history).

Metamodel: immutable Resource state. Activation and exchanges provenance / state transitions. Message history.

Activation, Aggregation, Alignment, Layers and Functional domains: Dynamic upper ontology

Metamodel Resource(s) get 'activated' (maybe in some 'degree' or 'distance' which is another Resource) when matched with another Resource (pattern) obtaining this way some kind of 'criteria' or 'comparison' result.

A full match returns the same Resource when other degrees of matching depends on comparisons between CSPO components of Resources in a given sort of criteria (order or comparison of quad elements and then sorting, for example).

<u>Dynamic upper ontology</u>:

If models shared by Peers share some common underlying ‘upper’ ontology, this could be aggregated, aligned and matched by only Metamodel data itself. That means no ‘hard-coded’ upper ontology but one that is dynamically shared and exported to other formats if necessary.

Upper ontology: Aggregate Metamodel(s) Metamodels. Peer Node level. Reify Metamodel hierarchy into Metamodel statements for each class. 'Dynamic': alignment methods between Peer(s). Align to OWL / RDFS models (SPARQL / endpoint).

Access control. Scopes. Domains. Roles (upper purpose ontology).

Upper purpose ontology: Modelled in Metamodel (reification). Constraints. Restrictions. Temporal / dimensional parts. ISO like ‘templates’, OWL, RDFS Metamodel IO export / import (endpoint).

Metamodel Object Model (endpoints / connectors APIs). Metamodel facade: Object graph / RDF APIs / others. Upper ontology aligned. Grammars (dialog alignment). Facade messaging / activation / utils.

Classes / RDF literals / Primitives (parse type kind assignment: Address, Age, etc.) Alignment algorithms.

UX: Grammars. Possible domain / range / properties. Dialog alignment.

Functional / Reactive programming. Resolution interfaces: domain / range input / results (scope, contexts, levels, content / activation). Index (identity, ‘content’), Naming (links / rels, IDs), Registry (context positions).

Activation / REST / Pipelines (P2P, addressing, message self routing, Resolution). Templates. Provenance. Addressable 'transmissions' (exchanges, listeners). Materialized transforms / operations / routes (cache / patterns / shapes / grammars).

Shape expressions (upper ontology, grammars, facade, activation). LDP. SoLiD. REST HATEOAS (HAL, JSON-LD) Resource(s) addressing / interface generation (Resolution interfaces).

(TBD).