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

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

I) Abstract

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 components (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.

II) Keywords

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

1) Introduction

Current landscape: Document Web vs. Data Web applications.

1.1) Description

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.

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.

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.

1.2) 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.

Considerations:

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).

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.

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.

1.3) Proposal

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.) resolve 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.

2) Solution

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).

2.1) 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.

2.2) Architecture

Logical features. Implementation features. Lambda. Runat.
Aggregation of knowledge layers: facts, concepts, roles / contexts). Examples.
Semantics + Semiotics.

Logical features.
Implementation features.
Characteristics (BI): IO, syndication, alignment, layers, activation, functional.

A Metamodel abstraction takes care of semantically represent and unify, by means of an API and interfaces, the different datasources, 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.

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.

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);

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

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

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

Metamodel layers:
Facts, Information, Knowledge.

TBD.

2.3) Enhanced deployments

Existing deployed solutions could leverage of the benefits of any of the two previously stated approaches. 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.

This third approach fits into what could be called ‘strict’ Data Web and is discussed in the section 4: Protocol.

As mentioned before an existing deployed application could benefit from this framework integrating its data and services 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.

3) Deployments

3.1) Instantiate Project bundle

A project instance starts instantiating one of the Project Bundle Maven archetypes. TBD.

3.2) Select Metamodels / Connectors

An application instance Project bundle consists of a set of chosen Connectors (for back end services, databases, etc) which instantiate Connections. This, along Metamodels (Backend, Domain and Purpose Metamodels) are the building blocks of an integration solution.

Connectors are the way an existing or new deployed application communicates to the ESB and performs knowledge aware operations. Connectors are kind of drivers for databases, web services, etc. They handle communication with the outside world. They may be mostly sources of data (a read only file, for example) or a bi directional sink where data could be read or written.

Connector APIs should handle the notion of ‘context’ as the requests and responses provided by them are to be meaningful in the scope of these interactions.

More information regarding specific languages / platforms API bindings of connectors is in the implementation section.

For a specific kind of Metamodel (service) to be instantiated and bound into bus pipes and endpoints implementations should provide with an archetype fulfilled with corresponding implementations of a metamodel artifacts (classes, interfaces, templates, driver).

Connector Driver is the most low-level integration interaction / IO component and is the factory of the monadically wrapped objects of the top-level Metamodel hierarchy class (interface): Resource.

Section 3.3.3 describes details of some specific core Metamodel implementations in more depth.

3.3) Clients. Visualization. Browser. Domain Use Case

Enterprise business applications integration: Unified augmented workbench. QA (Wizard) style.

Domain, use cases: Music & Movies (plus DBPedia) retail, record, artist / publisher frontends. Core business cases plus enhancements. Integration with existing APIs.

Music / Movie store (buy, rental). BBC Music, IMDB, DBPedia, Geo / Mondial DB, Store DB (accounts, transactions, etc.). Amazon, iTunes, Netflix, Spotify, etc. Endpoints. Alignment (catalog abstract resource / concrete item resource: roles in context, ID, attributes in transactions / browsing). Platform endpoint (Java EE JCA / REST HATEOAS HAL / JSON-LD protocol nodes).

Features: linkeddata.org / Freebase / DBPedia (async) augmented. Time, places, etc.

API designed for custom implementations of metamodels (over the core ones) with extension points enabling instances of the framework to behave according business objectives: Templates, Transforms, formats and other application specific extensions which integrates with other (custom) metamodels.

Business integration / business translation templates / transform. Dynamic ‘clients’ expecting customized ‘standard’ state exchanges (representations, schema, linking conventions). HATEOAS HAL / JSON-LD. Translation / rendering.

Tiles. XUL dynamic templates.

Visualization: Messages, Resources. Nested (context) tiles. Knowledge interfaces (activation operations).

UX: ZK / ZUL Templates & transforms from endpoints schema metadata / instances (tiles). JCA / JAF / DCI / REST. Activation domain browser.

API designed for custom implementations of metamodels (over the core ones) with extension points enabling instances of the framework to behave according business objectives: Templates, Transforms, formats and other application specific extensions which integrates with other (custom) metamodels.

Business integration / business translation templates / transform. Dynamic 'clients' expecting customized 'standard' state exchanges (representations, schema, linking conventions). HATEOAS HAL / JSON-LD. Translation / rendering.

4) Implementation

4.1) ServiceMix / OSGi container: Peers / Project Bundles

The integration framework proposed here is implemented as a set of Apache ServiceMix / JBoss Fuse APIs Maven bundle archetypes for being executed into an OSGi container as Apache Karaf (the one provided with both frameworks).

An OGGi blueprint Metamodel namespace is to be provided for the instantiation of the different metamodel services implementations (Components). Connector Driver and declarative metadata regarding Metamodel implementation must be provided here.

A provided Apache Camel custom component (with pipes / contexts bound to a Metamodel service) will have endpoints bindings exposing different metamodel prefix URIs (one for each Metamodel hierarchy level).

Messages: Metamodel layers hierarchy normalized to one Message format. Encoding. Routing. Aggregation.

Message IO: Exchange between parent / child Resource hierarchy layers context endpoints. Metamodel, custom component bound, resources activated / activates on input / output. Routing, Processor, Transforms, Aggregators: align (ID: enrich / filter, Attrs.: normalize, Contexts: sort). Templates (Exchange plus context resource).

A set of deployed pipes for Metamodel hierarchies comprise a Peer which can be consumed by any of the means of using Connectors in an application or that also may be integrated into a P2P deployment by the use of ‘discovering’ able Metamodel Connector drivers (section 4: Protocol).

API designed for custom implementations of Metamodels (over the core ones) with extension points enabling instances of the framework to behave according business objectives: Templates, Transforms, formats and other application specific extensions which integrates with other (custom) metamodels.

Business integration / business translation templates / transform. Dynamic ‘clients’ expecting customized ‘standard’ state exchanges (representations, schema, linking conventions). HATEOAS HAL / JSON-LD. Translation / rendering.

4.1.1) Apache Camel custom component ('metamodel:' URI prefix)

For ease of development an Apache Camel custom component will be provided which will handle low-level exchange of context bound metamodel services. The URIs will have a ‘metamodel:’ prefix and a Metamodel resource type class name after that.

Routes will be provided between resource types URIs adjacent to each other in the Metamodel class hierarchy (see 3.3.1). Domain or discovery specific routes / transforms could be added.

Metamodel service instances will be declared by a metamodel blueprint XML namespace (and custom tags).

Messages: Activation / Protocol (aggregate messages). Reactive publisher / consumer. Metamodels message exchange will be handled by metamodel Camel contexts endpoints (example: to / from metamodel:fact - metamodel:kind).

4.1.2) Metamodel OSGi blueprint namespace ('metamodel' tag)

Custom OSGi blueprint namespace provides a way to declare Metamodel service instances (persistence, alignment, reasoning, inference, endpoints, etc.). A Metamodel instance is backed by an Apache Jena RDF ontology instance. An upper (common) ontology should exist for alignment of domains.

Aggregation is performed at Metamodel level as means to provide basic type inference and (dataflow) reactive activation mechanisms from / to message exchanges.

Individual out-of-the-box Metamodels provides (message driven) services: persistence, alignment, reasoning, inference, endpoints, etc. A metamodel ‘Connector / driver’ is the ultimate backend of such functionalities. Maven archetypes should exist that ease development of metamodels.

OSGi Service interface exported by Metamodels.
Instantiates / exports Camel routes / bindings: Connector, layer, MetaModel. Broadcast / pattern URIs.
Metamodel routes (layers: connector / sources (bus?) <-> layers <-> Metamodel Protocol Message Dialog).
Aggregation. APIs (extension points) declarations.

4.1.3) Connector Bundle Project (Driver) : Connection

IO Underlying service / datasource.
Provider (service, source) of wrapper Metamodel service instance Data layer.
Routes layers: connector / source (bus?) <-> layer <-> MetaModel.
APIs (extension points) between routes implementation (class, interface, templates).
Protocol: Source generated events or 'polling' (ie. query for rows, query for svc. args.).
Protocol: Layers routing 'dialog' through activation flows. pub/sub routes flows through layers when lower/higher layer activates lower/higher layer. API customization.

4.1.4) Metamodel Service Instances

(declarative metamodel namespace, Connectors 'Connection').
Source (Connector 'Connection' producer / consumer) declaration. IO (RDBMS, Service, etc).
Aggregation. APIs (extension points) between routes implementation (class, interface, template).

4.1.5) Domain Metamodels Instances

Source: Connector service / source 'Connection' exposes / consumes REST to Connector Client. Connects with other Metamodels (Message IO)
Aggregation enables merge / integration.
Declaratively aggregated / inferred domains (Purpose ontology alignment). Ranks, Topic, Topic Instance.
Templates / API (extension points) for aggregation / merge.
Domain data / behavior schema / instance data encoded in Metamodel layers CRUD / Message IO.

4.1.6) Application Client Project

JCA / JAF/ AOM / DCI HATEOAS (HAL, JSON-LD) Domain Metamodel Browser.

Client interfaces vía Metamodel protocol endpoint implementations. Local DOM / ORM. JAF / JCA (Java).

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

Protocol bindings (Node IO) Services:

REST HATEOAS (HAL / JSONLD). LDP.
SOAP.
O Data.
SPARQL.

Platform bindings (Clients) Services:

Java EE (JCA / JAF / dynamic languages).
JavaScript (browser).
JavaScript (Node JS).
PHP.
.Net (LINQ).

Platforms bindings are meant to provide a ‘native’ language / platform representation (classes and instances) of knowledge (data, schema and behavior) stored into Node(s) via their services.

Interaction with knowledge aware nodes is provided by platform bindings wrapping calls into an specific protocol binding. Then, entities (classes / instances / behavior) of each specific platform / language / runtime are ‘generated’ from parsed data and metadata.

This way business applications of different platforms in different languages will leverage the benefits of having a real time integration and interoperability backend.

4.1.7) Application Project Bundle (Maven archetype)

Metamodel blueprint instances configuration (declaratively: Connectors, APIs, etc).
APIs (extension points): class, interface, template between Metamodels. Routes query / transform (filter events / msgs.).
At least one 'domain' Metamodel to be public.

5) Connectors: Declaratively configured Metamodel Connections:

. RDBMSs.
. LDAP / JNDI / JCR.
. Camel (JMS, legacy).
. Service (REST / SOAP).
. Alignment: Identity (classification: class, metaclass / instance, class / occurrence / instance

nesting).
. Alignment: Attributes / Links (regression: class, metaclass / instance, class / occurrence / instance nesting).
. Alignment: Contextual sort (clustering: class, metaclass / instance, class / occurrence / instance nesting).
. Streams (Big Data: Spark, MapReduce, etc.).
. BI: OLAP / Mining.
. Lucene.
. Google Tensor Flow (ML for Alignment Metamodels: ID, Attrs, Ctx and Big Data. ‘Distance’ calculation models between two aligned resources in a given ‘axis’ or parent class).
. Drools / Flow CEP / JBPM.
. Purpose Metamodel: Task accomplishment services / QA.
. Dimensional alignment: x is y of z in w. Travel example.
. Others.

Alignment outputs: augmented Metamodel facts (update history / provenance data). New statements / roles / contexts.

QA Alignment: Task accomplishment. Goals. Determine arguments / steps for expected results in context (hierarchically).

QA Alignment: Task accomplishment. Determine possible result arrangements (Purpose Goals) of given arguments / elements in context (workflows, planner).

Template Metamodel: RESTFul HATEOAS engine. Templates (XSL) for endpoints representations.

6) Metamodel API

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.

Metamodel: Jena impl, align upper metamodel (alignment metamodel). Layers (backend, domain, purpose) upper (data, information, knowledge) templates.

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

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

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

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 a given occurrence.

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.

For whatever occurrences a player instance may have there will be a corresponding set of ‘attributes’ and ‘values’ pairs determining, 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, the aggregated pairs of those occurrences, in common with other instances, the ‘subject kind’ of the resource, thus performing basic type inference.

A Resource is a (functional) monadic type which wraps a reference of its parent resource (resource in which it occurs) and a (dynamic) list of its occurrences (parent / child).

A top level Resource implements / extends custom Camel Message implementation for normalized endpoint IO (messages, persistence, aggregation). A top level Resource monadically wraps a Source ‘connection’ with metamodel kind specific behavior.

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 Application / Binding Statements: (Application, Monad, Function, Monad); Bound functions.

Class hierarchy. Normalize Metamodel for facts, concepts, roles / contexts layers (data, information and knowledge).

Resource is a monad of <T extends Source> (driver Connector connection provided backend). A subclass instance set represent a subset relationship with those of its superclass.

Metamodel (source : Metamodel, data : Resource, information : Kind, knowledge : Statement);

Resource (player : Resource, occurrence : Resource, attribute : Resource, value : Resource);

Statement (Statement, subject : Resource, predicate : Resource, object : Resource);

Kind (Kind, occurrence : Statement, spo : Resource, class : Class);

Class : Kind (Class, occurrence : Kind, attribute : Resource, value : Resource);

Rule : Statement (Rule, subject : Kind, predicate : Kind, object : Kind);

Flow : Resource (Flow, rule : Rule, lhs : Class, rhs : Class);

Data: (Resource, Flow); Information: (Kind, Class); Knowledge: (Statement, Rule);
Functional apply:
occurrence(occurring) : occurrences.
Kind(Class) : Statements.

Functional query:

Functional transform:

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: Task accomplishment services / QA. Over Domain Metamodels. Over Backend Metamodels.

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

Layer scopes for Purpose Metamodels:

Data: Facts / Dialog (Resource / Flow), Information: Concepts / Session (Statement / Rule), Knowledge: Behavior / Templates (Class / Kind).

Purpose Metamodel: 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 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.

4D: What (state in time), Where, When (ISO 15926. Temporal parts).

Upper ontology (ISO 15926): Templates.

<u>Metamodel (updated)</u>:

Monad (Player, Occurrence)

Function (Attribute, Value)

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

Resource : Application (Resource, Resource, Resource, Resource)

Statement : Resource (Statement, Resource, Resource, Resource)

Rule : Statement (Rule, Klass, Klass, Klass)

Metamodel : Resource (Metamodel, Statement, Klass, Resource)

Behavior : Metamodel (Behavior, Rule, Flow, Statement)

Klass (Klass, Statement / Metamodel?, Resource, Resource)

Flow : Klass (Flow, Rule / Behavior?, Klass, Klass)

Statement: Context aggregates Subjects / Topics?

Resource<T, M extends Resource<?, ?>> extends Functor<T, M>

M flatMap(Function<T,M> f); T::funName : M. (T t) -> t.funName(args) : M. Function.apply(val).

Functions: Subject, Predicate, Object, Aggregate, Compose.

filter(Predicate<T>)

TODO:
Factories. Parameterized instances (Resource, Statement, Metamodel, Klass) constructor. Repository (get instances triple store) metamodel connection. Constructor (parameterized get instance). Unit. Apply : Application.apply(Application) performs aggregation (occur attrs, vals) in function of recursive match. Match : pattern selector. Activation. Aggregation.

Resource<T extends Resource>.. Ts: super(T this). T delegates getters and setters to super Resource (cast). Resource, Statement, Rule: parameterize hierarchy, cast getters and setters.

Type constructor (super call with this). Unit.

Match.

Apply. Performs aggregation (occur, attr, val match, generic at Resource level?).

Resource<T, M extends Resource<?, ?>> extends Functor<T, M>

M flatMap(Function<T,M> f);

Function: Aggregate (i.e. between Statement, Rule).

Function: Compose (i.e. between Statement, Metamodel).

Aggregate / Compose on events. Dataflow.

Resource.player = this; (URI).

Resource.resourceFactory

Resource.statement (JenaResource)

Updates

Monads. Reactive. Bookmarks. Dev. Guides. Design into docs.

Observable. Streams (Data events ordered, contexts?, distance?). Publisher. Subscriber. (Context roles, co-monads). Subscription (Context). Interaction: onNext, onError, onComplete. Subjects (Observable, Subscriber). Operations. Filter (where). Selector (new stream). Merge. Resource monad of Observable T extends URI (URI : REST API events). Events: aggregation, composition, inter model IO (data, domain, knowledge, domain, data IO).

[ ] class FOptional<T> implements Functor<T,FOptional<?>>. of ctor.

[ ] Collections of values T. Map apply to each one. Collect chained members. java.util.stream.Stream<T>. list.stream().map(f).collect(toList()).

[ ] Promise<Customer> customer = //...
[ ] Promise<byte[]> bytes = customer.map(Customer::getAddress).map(Address::street).map(String::getBytes);

[ ] When upstream promise completes, downstream promise applies a function passed to map() and passes the result downstream. Suddenly our functor allows us to pipeline asynchronous computations in a non-blocking manner. Flows (Consumer, aggregation, composition).

[ ] Resource monad T (final player) delegates getters / setters into player T (extends Statement) or Functons which extracts SPOs Resources. Subclasses define SPOs. Constructor(T).

[ ] Resource<T, M extends Resource<?, ?>> extends Functor<T, M>

[ ] M flatMap(Function<T,M> f); T::funName : M. (T t) -> t.funName(args) : M. Function.apply(val).

[ ] Functions: Subject, Predicate, Object, Aggregate, Compose.

[ ] filter(Predicate<T>)

[ ] JenaResource. Statement / Factory. Get instance receives factory (package visibility).

[ ] Monad<Iterable<T>> sequence(Iterable<Monad<T>> monads) : occurrences (Rx Observable instead of Iterable)

[ ] ListFunctor<Monad<T>>::sequence

[ ] (Monad<Iterable<T>>) resources.map((ListFunctor<Statement> s) -> ...);

Streams. Declarative DCI. Cactoos.org

Selector / match: Resource Iterable T extends URI : occurrence / attribute / value class / instance patterns (recursive). Class URIs (contexts) / Instance URIs (occurrences). Filter. Predicate.

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

Resource : Monad<FListIterable<T extends URI>> : occurrences (Rx Observable instead of Iterable)

Reactive flows between layers and metamodels (fmap transform functions) also on MM IO (RxCamel, OSGi services consumers).

Resource : Monad<Iterable<T extends URI>> : occurrences (Observable)

[ ] Statement : URI (Resource, Resource, Resource, Resource) CSPO member Resources.

[ ] Topic : Statement (Statement, Resource, Resource, Resource)

[ ] Rule : Topic (Rule, Klass, Klass, Klass)

[ ] Metamodel : Statement (Metamodel, Topic, Klass, Resource)

[ ] Behavior : Metamodel (Behavior, Rule, Flow, Statement)

[ ] Klass : Statement (Klass, Metamodel, Resource, Resource)

[ ] Flow : Klass (Flow, Behavior, Klass, Klass)

6.1) P2P Metamodel integration

Peers (OSGi containers). DHTs (hash discovery of models and resources). Scopes. Advertisements. Camel network bindings (routes). Distributed dataflow activation graphs: inputs: Resource Observable T, models, outputs: Resource Observable T. Models : Function input, output: aggregate, compose, alignment, etc. 'discovered', resolved and applied from context (input) on each layer metamodel. Declarative models / trained models 'portable' (shared, distributed, discoverable) across peers.

7) Functional Metamodel API

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.

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.

Protocol semantics: Discovery, subscriptions. Commands (JAF). Message: C: Path, S: Assert, P: Query, O: Reply.

Domain translation: (dynamic) templates.

Representations: Browseable hypermedia (HATEOAS HAL / JSON-LD).

Resource API: Activation. Reactive. Dataflow.

Resource.getParent() : Resource;
Resource.getOcurrences(Resource pattern) : Set<Resource>;
Resource.retrieve(Resource pattern) : References selector;
Resource.from(Resource pattern) : Referenced selector;
Resource.apply(Resource pattern);
Resource.history(Resource pattern);

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

8) Dialog Protocol

Components:

Connector

Connection

Metamodels

Component Metamodels:

Backend Metamodel. Connector driven. Shared (scope context attribute).

Domain Metamodel. Connection driven. Shared (scope context attribute).

Purpose (dialog / session) Metamodel. Shared (scope context attribute).

Component has its corresponding associated Metamodel(s) given its Connection / Connector and other configurations (domains, purpose, etc.).

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

Dialog flow inside metamodel: Data, Information, Knowledge, Information, Data layers.

Dialog flow between metamodels: Data, Data layers.

Metamodel population (Data): Connector -> Connection -> Metamodel (aggregates Data into Information and Knowledge layers).

Align / Aggregate: Metamodel aggregates input Data as Information and Knowledge.

Upper ontology Metamodels: ontology backends.

Activation: Dependencies. Selector. Events.

Activation: Dependants. Selector. Events.

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

Data: Query / Facts (Resource / Flow).

Information: Response / Dialog (Kind / Class).

Knowledge: Lookup / Knowledge (Statement / Rule).

8.1) Messages

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):

(backendMetamodel:backendMessage, domainMetamodel:domainMessage, purposeMetamodel:purposeMessage);

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).

Connector <- (?, ?) -> Connection <- (Data, Data) -> Metamodels <- (Data, Data) -> Metamodels

8.2) Routes

Custom Camel component routes between Components Connectors / Connections and Metamodels (layers).

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

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

8.2.1) Connector / Connection routes

Connector to Metamodel routes. Connector to Connector routes. Connector endpoint.

8.2.2) Connection / Metamodel routes

Connector to Metamodel routes. Connector to Connector routes. Connector endpoint.

8.2.3) Metamodel / Metamodel routes

Connector to Metamodel routes. Connector to Connector routes. Connector endpoint.

8.3) Routes APIs

Classes, interfaces and templates API for configuring route behaviors.

8.3.1) Connector / Connection routes API

Classes, interfaces and templates

8.3.2) Connection / Metamodel routes API

Classes, interfaces and templates

8.3.3) Metamodel / Metamodel routes API

Classes, interfaces and templates

9) Appendix: Monads

Template 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).

10) Links. References

Semantic Web:

https://www.w3.org/2000/10/swap/Primer

http://logicerror.com/semanticWeb-long

http://infomesh.net/2001/swintro/

OSGi (ServiceMix / Fuse container):

http://www.theserverside.com/news/1363825/OSGi-for-Beginners

http://www.vogella.com/tutorials/OSGi/article.html

http://www.vogella.com/tutorials/OSGiServices/article.html

https://www.ibm.com/developerworks/opensource/library/os-osgiblueprint/

ServiceMix / Fuse:

https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.3/html/developing_and_deploying_applications/develop

https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.3/html/deploying_into_apache_karaf/

https://access.redhat.com/documentation/en-us/red_hat_jboss_a-mq/6.3/html/product_introduction/

https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.3/html/apache_camel_development_guide/

https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.3/html/apache_cxf_development_guide/

https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.3/html-single/apache_camel_component_reference/

Apache Camel (for custom component development):

http://camel.apache.org/creating-a-new-camel-component.html

https://github.com/FuseByExample

http://camel.apache.org/data-format.html

http://people.apache.org/~dkulp/camel/xslt.html http://camel.apache.org/xpath.html

http://camel.apache.org/xquery.html

OSGi Blueprint custom namespace handlers:

https://docs.spring.io/spring/docs/2.5.x/javadoc-api/org/springframework/beans/factory/xml/NamespaceHandler.html

https://github.com/WASdev/sample.osgi.blueprint-cm

http://grepcode.com/file/repo1.maven.org/maven2/org.apache.aries.blueprint/blueprint-parser/1.0.0/org/apache/aries/blueprint/NamespaceHandler.java

http://aries.apache.org/

DCI (Data, Context, Interaction):

https://en.wikipedia.org/wiki/Data,_context_and_interaction

Recomended:

XSL, XPath, XQuery: http://www.w3schools.com