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

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 datasources 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.

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

1.1) Objectives

Current landscape: Document Web vs. Data Web applications.

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 evenually 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.

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.

1.2) Description

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.

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

2.1) Leverage existing solutions

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.

2.2) Architecture

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

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.

2.3.1) Connectors

Connectors are the way an existing or new deployed application communicates to the ESB and performs knowledge aware operations.

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.

2.3.2) Domain use case example

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 (JavaEE JCA / REST HATEOAS HAL / JSONLD 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.

2.3.3) Visualization

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.

2.3.4) Metamodels and Drivers

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).

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) Implementation

3.1) OSGi container: Peers

The integration framework proposed here is to be implemented as a set of Apache ServiceMix / JBoss Fuse API Maven bundle archetypes.

An OGGi blueprint metamodel namespace is to be provided for the instantiation of the different metamodel services implementations. Driver and declarative metadata regarding metamodel impl.

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 previous / next 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 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.

3.2) Apache Camel custom components / Metamodel blueprint

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 namespace (and tags).

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

3.3) Metamodel service

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 metamodels provides (message driven) services: persistence, alignment, reasoning, inference, endpoints, etc. A metamodel 'driver' is the ultimate backend of such functionalities. Maven archetypes should exist that ease development of metamodels.

3.3.1) Metamodel API

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.

Layers hierarchy:

Resource is a monad of Source (driver provided backend). Classes are enumerated from the top most to the lower most in the hierarchy. A sub class instance set represent a sub set relationship with those of its super class.

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

Resource(T extends Source): (Resource, Resource, Resource, Resource);

Statement(T extends Resource): (Statement, Resource, Resource, Resource);

Model(T extends Statement): (Model, Resource, Resource, Resource);

Model example:
Model: (Database / Service, Table / Operation, Row / Arguments, Cell / Value);

Statement example:
Statement / Table: (Resource Table, Resource PK, Resource Column, Resource Value);

Fact (dimensionally aggregated from SPO Statement): (Subject / Fact, Kind / Table, Attribute / Column, Value);

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

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

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

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

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

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

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

Dimensional statements:
Encode relations: x is y of z in w.

Resource statements (Source SPO): (aTravelStmt, travel, distance, 60km); (aTravelStmt, travel, origin, placeA); (aTravelStmt, travel, destination, placeB);

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

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

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

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

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

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

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

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

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

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

Alignment: abstract upper ontology. Primitives. Inference and alignment by comparison / relationship (roles) in contexts. Concept lattice / FCA.

Resource encoding: encode type (instance), order and attribute alignment metadata aggregated as kinds / classes.

Alignment: Semiotic contexts (sign, concept, object). Metamodel. Layers.

3.3.2) Functional Metamodel API

Resource API: Activation. Reactive. Dataflow.

Resource.getParent() : T;
Resource.getOccurrences(Resource pattern) : T super;
Resource.getFactory() : Context;
Resource.retrieve(Resource pattern);
Resource.from(pattern parent, pattern occurs);
Resource.match(parent, pattern)
Resource.apply(occurs ctx, pattern occur);
Resource.history(Resource pattern);

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

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

API: OSGi Blueprint namespace for metamodel services. Camel contexts bound to a specific service instance.

API: Metamodel service. Interfaces. Hierarchy. Aggregation. Message IO, Resource functional / activation API (message flows).

API: Metamodel service implementations. Classes, interfaces, annotations. XML / XSL Templates. Paths / Queries (transforms). Driver API (low level backend / connectors IO to RDF Message). Peer archetype.

Flow messages: entity / behavior encoded (bidirectionally) in Messages. Functional API. Expose hier + svcs.

Metamodel API.

Dimensional statements.

Aggregation. Layers.

3.3.3) Metamodel implementations

Backend. Provides component service.

Service. Provides component service.

Alignment. Provides component service.

3.4) Messages

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

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

3.5) Message flows

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

Dialog example (fact / kind):

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

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

3.6) Connectors

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

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

Protocol bindings (Node IO) Services:

REST HATEOAS (HAL / JSONLD). LDP.
SOAP.
OData.
SPARQL.

Platform bindings (Clients) Services:

JavaEE (JPA / JVM dynamic language).
JavaScript (browser).
JavaScript (NodeJS).
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) Protocol

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

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

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

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

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

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

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

C: Path.
S: Assert.
P: Query.
O: Reply.

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

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

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

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

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

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

CSPO Paths: Patterns. C: Path, S: Assertion, P: Query, O: Answer 'patterns'. Discovery by comparison alignment of obtained resource 'paths' rel with goal 'pattern' (referrer).

Example encoding:

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

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

De referenceable resource: aggregated Message (IO).

Domain translation: fulfill (dynamic) template.

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

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

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

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

Reified Metamodels (upper ontology).

OntoClean. Primitives (Number). Upper ontology. Bit map coding (digit, position). Octal comparison results mask (order rel expressed in three bits).

Number (Base, Digit*). Base : Number. Digit : Number, Position. Position : Number.

(Instance: Base, S: Number, P: Position, O: Digit);

(Occurrence, Resource : Number, Attribute : Base, Value : Digit);

(Fact1, Peter, wife, María);

(Fact2, Man, relationshipWith, Woman);

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

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

Resource Number: aggregated attribute/value pairs (define resource by extension).

Comparison encoding: Render (recursively) compared occurrences SPOs. Convert compared resources SPOs into functional context SPOs. Obtain comparison result operating converted resources (more specialized hierarchy match). Return functional contexts compared resource.

URI encoding (SPO instance, occurrence):

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

Example:

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

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

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

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

Dashboard: actionable domain translation of problem spaces flows.

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

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

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

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

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

(RO, RO, RO, RO);

Aggregated / Activation from RTL.

Peter, fact1, worksAt, IBM.

John, fact2, traineeOf, Peter.

Peter, fact3, promotionTo, seniorDeveloper.

John, fact4, replaces, Peter.

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

(fact1, fact2, fact3, fact4);

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

Resource Occurrences: Fact, Statement, Kind, Class.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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