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.

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.

Features.

Connectors.

1.2) Description

Current needs / problems. Considerations (BI).

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

This shall be accomplished by means of an event-driven architecture in a semantics aware layer which leverages diverse backend integration (sync) and schema merge / interoperability. Also, a

declarative layer is provided for aggregation and composition of related event flows of various objectives to met a given purpose.

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

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

1.3) Proposal

Business domain translation (dynamic templates).
Features (BI).

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

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

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

Feature: Data backends / services virtualization (federation / syndication / synchronization)

Any datasources (applications / services) entities and schema regarded as being meaningful for a business domain translation and integration use case, regardless of their source format or protocol, should be provided of an adapter mechanism (Node + Service) which makes it available as ‘reactive endpoint’ which provides CRUD services for other Node / Service to consume / synchronise.

Feature: Schema (ontology) merge / match / alignment

Diverse domain application data with disparate 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) attributes or links. Relationship type promotion.

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

Un enfoque sería el de “enriquecer” aplicaciones o servicios actuales con conocimiento relacionado al dominio de los mismos en el contexto de una interacción.

El otro podría ser “relevar” servicios y orígenes existentes para integrarlos en el contexto de un despliegue que convenga en exponer toda su funcionalidad actual aumentada y mejorada con servicios de bases de conocimiento y la integración declarativa con otros dominios o procesos.

2) Solution

2.1) Leverage existing solutions

Un ejemplo del segundo punto anterior sería que las instancias de determinados casos de uso en el contexto del dominio de determinada aplicación “disparen” instancias de flujos de casos de uso en aplicaciones de diversos dominios cuya realización está relacionada en algún modo con la realización del primero.

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

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), alignments, rules, workflows, inference, learning and endpoints.

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.

2.2) Architecture

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

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

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

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

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

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

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

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

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

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

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

Metamodel example: TBD.

2.3) Enhanced deployments

TBD.

2.3.1) Connectors

TBD.

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.

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.

2.3.4) Metamodels and Drivers

TBD.

3) Implementation

3.1) OSGi container: Peers

ServiceMix / Fuse API Bundle archetypes.

API: Metamodel Bundle archetypes.

Architecture: Node. Camel custom component endpoint bindings and Metamodel service binding.

Architecture: Services. Metamodel specific driver’s custom declarations.

Architecture: Message implementations. Encoding. Routing. Aggregation.

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

3.2) Apache Camel custom components

API: Apache Camel custom metamodel component. Metamodel endpoint URIs / routes. Bindings. Processing. Transforms.

Camel contexts bound to specific Metamodel service instances.

Messages: Activation / Protocol (aggregate messages). Reactive publisher / consumer.

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

3.3) Metamodel service

OSGi blueprint namespace provided.

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

3.3.1) Metamodel API

Dimensional statements:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Classes are modelled as a quad hierarchy of OOP classes:

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

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

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

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

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

Hierarchy: Model : Statement : Resource.

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

Statement(T extends Resource): Quad.

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

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

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

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

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

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

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

Model / Resource / Statement monads wraps specific node roles implementations of IO (specific models, Database, has specific resource and specific statement instantiated). Provides sources

CSPO IO / CRUD. Then aggregates Fact, Kind, Class, Event, Rule, Flow (Resource monads wrappers of their players).

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

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

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

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

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

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

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

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

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

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

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

Resource.occurrences() : Statement.

Statement.occurrences() : Model.

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

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

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

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

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

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

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).

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

Metamodel API.

Dimensional statements.

Aggregation. Layers.

3.3.2) Functional Metamodel API

Resource API: Activation. Reactive. Dataflow.

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

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

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

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

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

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

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

3.3.3) Metamodel implementations

Backend. Provides component service.

Service. Provides component service.

Alignment. Provides component service.

3.4) Messages

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

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

3.5) Message / Event 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.