

Application Integration and Multi Domain Generic Client Rendering Services

© 2025 Sebastián Samaruga

Enterprise Application Integration (EAI) / Business Intelligence (BI) stack leveraged by Semantic Web and GenAI / ML. Implemented in a Functional / Reactive stream-oriented fashion. Allow the integration of diverse applications by parsing application backends source data (tabular, XML, JSON, graph), inferring layered domain schemas, states, and data models, and exposing an activation-model API for cross-application interactions while synchronizing source backends. Infer existing applications behaviors or tasks and recreate them into an augmented interoperable model.

Overview

Semantic Web / GenAI enabled EAI (Enterprise Application Integration) Framework Proposal

This document covers the inception phase documentation links related to a novel approach of doing EAI through the use of Functional / Reactive Programming leveraging GenAI and Semantic Web (graphs inference) and also the implementation of a novel approach of doing embeddings, not only for similarity calculation but also for relationships inference, query and traversal in an algebraic fashion.

The goal is to allow to integrate diverse existing / legacy applications or API services by parsing theirs backend's source data (in tabular, XML / JSON, graph, etc. forms) and, by means of aggregated inference using semantic models over sources schema and data, obtain a layered representation of the domains and data of source applications to be integrated until reaching enough knowledge as for being able to represent application's behaviors into an inferred use-cases Activation model.

Expose the Activation model inferred use-cases types (Contexts) and transactions use-cases instances (Interactions) through a Producer generic use-case browser client / API. Allow to browse and execute use-cases Contexts and Interactions in and between integrated applications, possibly enabling use cases involving more than one source integrated application. Example: Inventory integrated application and Orders integrated application interaction. When Inventory application level of one product falls below some threshold an Order needs to be fulfilled to replenish the Inventory with the products needed for operational levels.

The concept is to manage raw Datasources data and schema (inferred) into layers of Aggregation, Alignment and Activation services. Then the Producer component is able to parse and render Activation model into an application (API / generic frontend) Contexts and Interactions browser. An Augmentation service provides for orchestration between the three main layers of the service architecture and provides for interaction between Datasources and Producer services.

Vision

Brief definition of intelligence:

The ability to convert entities Data (subjects key / value properties: product price) into Information (subjects key / value relationships, properties in a given context: product price across the last couple of months) and the ability to convert such Information into Actionable Knowledge (actionable tools / inferences into a given context / analogy: product price increase / decrease rate, determine if it is convenient to buy).

The goal is to facilitate the integration of diverse existing / legacy applications or API services by

parsing their backend's source data in tabular, XML, JSON, graph, etc. forms and, by means of

aggregated inference using semantic models over sources data, obtain a layered representation of the applications domains inferred schema, states and data of source applications to be integrated until reaching enough knowledge as for being able to represent application's behaviors into an inferred use-case oriented activation model API, rendering usable interactions in and between integrated applications inferred scenarios and keeping in sync integrated applications backends source data with the results of this interactions.

In today's competitive landscape, organizations are often hampered by a portfolio of disconnected legacy and modern applications. This creates information silos, manual process inefficiencies, and significant barriers to innovation. This Application Integration Framework project is a strategic initiative designed to address these challenges head-on.

The project's core goal is to "integrate diverse existing / legacy applications or API services" by creating an intelligent middleware layer. This framework will automatically analyze data from various systems, understand the underlying business processes, and expose the combined functionality / use cases through a single, modern, and unified interface keeping in sync this interactions with the underlying integrated applications backends.

Mission

Implement a Semantic (graphs inference) / AI / GenAI enabled Business Intelligence / Enterprise Application Integration (EAI) platform with a reactive microservices backend leveraging functional programming techniques.

Implement a novel custom way to encode embeddings algebraically, enabling GenAI / MCP custom interactions, not just similarity but also mathematical relationships inference and reasoning. This by means of FCA (Formal Concept Analysis) contexts and lattices.

Expose an unified API façade / frontend (Generic Client / Hypermedia Application Language: HAL Implementation) of integrated applications use cases (Contexts) and use cases instances (Interactions) by means of Domain Driven Development and DCI (Data, Contexts and Interactions) design patterns and render inter-integrated applications use cases that could arise between integrated applications.

Exposing this Activation model inferred use-cases types (Contexts) and transactions use-cases instances (Interactions) through a Producer generic use-case browser client / API. Allow to browse and execute use-cases Contexts and Interactions in and between integrated applications, possibly enabling use cases involving more than one source integrated application. Example: Inventory integrated application and Orders integrated application interaction. When Inventory application level of one product falls below some threshold an Order needs to be fulfilled to replenish the Inventory with the products needed for operational levels.

Values

Determinism: Favor explainable, reproducible behavior in inference and execution.

Explainability: Provide transparent semantics for schemas, roles, and transformations.

Interoperability: Align models and APIs to enable cross-application scenarios. Upper Ontologies.

Modularity: Compose functionality via reusable monads, kinds, and nodes. Functional Reactive Streams.

Scalability: Support large event streams, FCA lattices for numerical inference, and graph traversals.

Implementation

The idea is to build a layered set of semantic models, with their own levels of abstraction, backing a set of microservices from data ingestion from integrated business / legacy applications from their datasources, files and APIs feed to an Aggregation layer which performs type inference / matching, then to an Alignment layer which performs Upper Ontologies Matching and then to an Activation layer which exposes a unified interface to the integrated applications use cases, keeping in sync integrated applications backends with this Activation layer's interactions.

The proposal is not only to "integrate" but to "replicate" the functionalities of integrated or "legacy" applications based solely on the knowledge of their data sources (inputs and outputs) and, through heuristics (FCA: Formal Concept Analysis) and semantic inference, provide a unified API / frontend for each application's use cases (replicated) and for any

use cases that may arise "between" integrated applications (workflows, wizards), all while keeping the original data sources synchronized.

Generic Client

Incorporating or directly creating a new application or service (perhaps to be integrated with the previous ones) would simply be a matter of defining a source model schema and a set of initial reference data. And this today could certainly benefit greatly from GenAI / LLMs and MCP in both client and server modes.

The idea is that by doing an "ETL" of all the tables / schemas / APIs / documents of your domains and their applications, translating the sources into triples (nodes, arcs: knowledge graph) the framework can infer your entity types, relationships and the contexts ("use cases") possible in and between your integrated applications providing means for a generic overlay (Producer API Service, generic front end) in which to integrate in a unified, conversational and "discoverable" interface (API, web assistant, "wizards") the integrated contexts interactions in and between the source integrated applications.

To unify and integrate diverse data sources, transform all the information from each source into triples (Entity, Attribute, Value) into a graph in the "Datasources" component. The other components / services deal with type / state inference (Aggregation), relationships and equivalences / matching / ordering (dimensional) inference (Alignment) and use case descriptions / executions (Activation) then exposing the description of the possible contexts and their interactions in and between the integrated applications. The user interface component could be a generic front end or an API endpoint to interact according to the metadata of each context (use case) augmentation allowing to make possible Contexts executable and their executions (Interactions) browseable.

Simple example (use cases): I have fruits and vegetables, I can open a greengrocer's. I want to open a greengrocer's, I need fruits and vegetables. Actors: supplier, greengrocer, customer. Contexts / Interactions: supply, sale, etc.

Another example: I have these indicators that I inferred from the ETL, what reports can I put together? I want a report about these aspects of this topic, what indicators (roles) do I need to add.

Ultimately, it is about creating a "generator" of unified interfaces for the integration of current or legacy applications or data sources (DBs, APIs, documents, etc.) in order to expose diverse sources in an unified way, such as a web frontend (generic use case wizards), chatbots, API endpoints, etc. integrating the functionality of integrated applications use cases relating each other in an unified forms flow layout (wizards).

Core Architecture

Reactive Message Driven Services Core Streaming Layout:

- Single Topic Architecture
- Blackboard design pattern

Data Model

The idea is to enable model representations being equivalent (containing the same data) in various layers to be switched back and forth between each layer representation to be used in the most appropriate task for a given representation.

The nodes and arcs of the graph triples are URIs and should have a "retrievable" internal representation with metadata that each service / layer populates through the "helper" services: Registry, Naming (NLP) and Index service shared by each layer. Describe core model classes serialization in JSON.

Reference Model

Underlying Model for main persistence in the RDF store, reifying other models knowledge and enabling conversion back and forth other models representations handled by the Model Service.

FCA Model (Reference Model View)

FCA Prime IDs (Embeddings): (link Sowa)

Each ID is assigned a unique prime number ID at creation time. FCA Context / Lattices built upon, for example for a given Data / Schema predicate / arc occurrence role, having the context objects being the statement occurrence subjects and the context attributes the statement occurrence objects, Predicate FCA Context: (Subjects x Objects). For a subject statement occurrence the context is: Subject FCA Context: (Predicates x Objects and for an object statement occurrence role the context is: Object FCA Context (Subjectx x Predicates).

Embeddings: For an ID, its prime ID number plus all ID's occurrences embeddings. For an IDOccurrence, its ID class embeddings, its occurring ID embeddings and its context embeddings.

Embeddings similarity: IDs, IDOccurrences sharing the same primes for their embeddings in a given context. FCA Concept Lattice Clustering. (TODO).

Statements:

(Context, Attribute, Value)

TODO:

FCA / Multidimensional features (OLAP like):

Dimensions: Time, Product, Region

Units: Month / Year, Category / Item, State / City

Context : (Context, Attribute, Value)

Examples:

(soldDate, aProduct, aDate)

((soldDate, aProduct, aDate), Product, aProduct)

((soldDate, aProduct, aDate), Product, aProduct), Region, aRegion)

URIs are identifiers (Strings) and have assigned an unique prime number ID at their creation time. FCA (Formal Concept Analysis) techniques could be employed to build a concept lattice for each URI in a given context where the product of the primes of the URI context occurrence concept lattice attributes and values URIs are employed to identify the concept the URI belongs to and to subsume other possible attributes.

Classes:

Context
Relation
Object
Attribute

Statements:

(Context, Relation, Object, Attribute).
CPPE / RCV inference schema / data Statements.

Sets Model (Reference Model view)

Classes:

Context extends IDOccurrence
Subject extends IDOccurrence
Predicate extends IDOccurrence
Object extends IDOccurrence

Interface Kind<OccurrenceType, AttributeType, ValueType>

- superKind : Kind
- attributeValues : Tuple<AttributeType, ValueType>[]
- occurrences : OccurrenceType[]

Reification: Kind implementations extends / plays Subject, Predicate and Object roles in statement.

SubjectKind extends Subject, implements Kind<Subject, Predicate, Object>

PredicateKind extends Predicate, implements Kind<Predicate, Subject, Object>

ObjectKind extends Object, implements Kind<Object, Predicate, Subject>

The underlying model Statements can be represented as sets being Subjects, Predicates and Objects three sets where the intersection of Predicates and Objects sets conforms the “Subject Kinds” set, the intersection of the Subjects and Objects sets conforms the “Predicate Kinds” set, the intersection of the Subjects and Predicates sets conforms the “Object Kinds” set and the intersection of the three sets

conforms the “Statements” set. The set that encloses Subject, Predicate and Object sets is the Context set.

Sets based inference and functional algorithms should leverage this form of representation of the model graph.

Statements:

Data: (Context, Subject, Predicate, Object)

Schema: (Context, SubjectKind, PredicateKind, ObjectKind)

Dimensional Model (Reference Model view)

Classes:

ContextStatement extends Statement(C, S, P, O)

Dimension

Attribute / Axis

Value / Measure

Statements:

(ContextStatement: recursive, Dimension, Attribute / Axis, Value / Measure)

Examples:

(Time, soldDate, aProduct, aDate)

((Time, soldDate, aProduct, aDate), Item, Product, aProduct)

((Time, soldDate, aProduct, aDate), Item, Product, aProduct), Region, Country, aCountry)

Encode inference of order relationships. Implement Order inference as a feature of the Dimensional Model: type (schema) and instances (data) hierarchies inferred in FCA Contexts.

DOM Model (Reference Model view)

Classes:

Instance extends IDOccurrence

- id : ID
- label : string
- class : Class
- attributes : Map<string, Instance>

Class extends Instance

- id : ID
- label : string
- fields : Map<string, Class>

Statements:
(Class, Instance, Field, Instance);

Activation (DCI, Actor / Role) Model (Reference Model view)

Classes:

Context
- roles : Role[]

Role extends Class
- previous : Map<Context, Dataflow>
- current : Map<Context, Dataflow>
- next : Map<Context, Dataflow>

Dataflow extends Context

- role : Role
- rule : Rule
Interaction
- actors : Actor[]

Rule: Dataflow specification.

Actor extends Instance
- previous : Map<Context, Transform>
- current : Map<Context, Transform>
- next : Map<Context, Transform>

Transform
- actor : Actor
- production : Production

Production: Transform execution.

Statements:
Data: (Context, Interaction, Actor, Transform)
Schema: (Context, Context / Dataflow, Role, Dataflow)

Class Model: ResourceOccurrence Hierarchy

Resource Monad bound objects.

ResourceOccurrence
- representation : Representation
- onOccurrence(ResourceOccurrence occurrence)

- getOccurrences(S, P, O)
- getOccurringContexts(S, P, O)
- getAttributes() : String[]
- getAttribute(String) : String
- setAttribute(String, String)

ID extends ResourceOccurrence

- primeID : long
- urn : string
- occurrences : Map<Kind, IDOccurrence[]>
- CPPEmbedding : long

IDOccurrence extends ID

- occurringId : ID
- occurringContext : ID
- occurringKind : Kind

Subject extends IDOccurrence

- occurringId : ID
- occurrenceContext : Statement
- occurringKind : SubjectKind

Predicate extends IDOccurrence

- occurringId : ID
- occurrenceContext : Statement
- occurringKind : PredicateKind

Object extends IDOccurrence

- occurringId : ID
- occurrenceContext : Statement
- occurringKind : ObjectKind

Statement extends IDOccurrence

- subject : Subject
- predicate : Predicate
- object : Object

Parameterized interface Kind<Player, Attribute, Value>

- getSuperKind() : Kind
- getKindStatements() : KindStatement
- getPlayers() : Player[]
- getAttributes() : Attribute[]
- getValues(Attribute) : Value[]

Parameterized class KindStatement<Player extends Kind, Attribute, Value> extends Statement

SubjectKind extends Subject implements Kind<Subject, Predicate, Object>

- statements : SubjectKindStatement[]

SubjectKindStatement extends KindStatement<SubjectKind, Predicate, Object>.

PredicateKind extends Predicate implements Kind<Predicate, Subject, Object>

- statements : PredicateKindStatement[]

PredicateKindStatement extends KindStatement<PredicateKind, Subject, Object>.

ObjectKind extends Object implements Kind<Object, Predicate, Subject>

- statements : ObjectKindStatement[]

ObjectKindStatement extends KindStatement<ObjectKind, Predicate, Subject>

Kinds Aggregation:

Kinds: Statements Predicate FCA Contexts (concepts hierarchies)

States: Statements Subject FCA Contexts (concept hierarchies)

Roles: Statements Object FCA Contexts (concept hierarchies)

Kinds Schema Aggregation:

Aggregation over KindStatement(s) SPOs.

Graph (Statements Occurrences given their SPOs / Kinds contexts) implements Kind<Subject, Predicate, Object>

- context : Kind

- statements : Statement[]

Model (Graph Occurrences) extends Graph

- graphs : Graph[]

- merge(m : Model) : Model

ContentType

extends Model

- kind : Kind

- typeSignature : String

Representation extends ContentType

- contentType : ContentType

- encodedState : String (Encoding Types)

ResourceOccurrence hierarchy Resource Monad bound API

Dispatches to ResourceOccurrence Representation ContentType.

ResourceOccurrence Events:

ResourceOccurrence::onOccurrence(ResourceOccurrence occurrence) : ResourceOccurrence context.

ID::onOccurrence(IDOccurrence) : URN
IDOccurrence::onOccurrence(SPO / Kinds) : ID
SPO / Kinds::onOccurrence(Statement) : IDOccurrence
Statement::onOccurrence(Graph) : SPO / Kinds
Graph::onOccurrence(Model) : Statement
Model::onOccurrence(ContentType) : Graph (merge)
ContentType::onOccurrence(Representation) : Model
Representation::onOccurrence(ResourceOccurrence) : ContentType

ResourceOccurrence Occurrences:

ResourceOccurrence::getOccurrences(S, P, O) : ResourceOccurrence. S, P, O filter /criteria / matching.

Leverages CPPE / RCV / FCA / Kinds / Alignment schema / instances inference / filter / query / traversal.

Representation::getOccurrences(S, P, O) : ResourceOccurrence
ContentType::getOccurrences(S, P, O) : Representation
Model::getOccurrences(S, P, O) : ContentType
Graph::getOccurrences(S, P, O) : Models
Statement::getOccurrences(S, P, O) : Graphs
SPO / Kinds::getOccurrences(S, P, O) : Statements
IDOccurrence::getOccurrences(S, P, O) : SPO / Kinds
ID::getOccurrences(S, P, O) : IDOccurrence

ResourceOccurrence Occurring Contexts:

ResourceOccurrence::getOccurringContext(S, P, O) : ResourceOccurrence. S, P, O filter /criteria / matching.

Leverages CPPE / RCV / FCA / Kinds / Alignment schema / instances inference / filter / query / traversal.

ResourceOccurrence::getOccurringContexts(S, P, O) : Representation
Representation::getOccurringContexts(S, P, O) : ContentType

```
ContentType::getOccurringContexts(S, P, O) : Model  
Model::getOccurringContexts(S, P, O) : Graphs  
Graph::getOccurringContexts(S, P, O) : Statements  
Statement::getOccurringContexts(S, P, O) : SPO / Kinds  
SPO / Kinds::getOccurringContexts(S, P, O) : IDOccurrence  
IDOccurrence::getOccurringContexts(S, P, O) : ID  
ID::getOccurringContexts(S, P, O) : URN
```

Statements, SPOs and Kinds

Elevate statements into higher-order kinds for schema and instance traversal.

Identifiers

Represent IDs and IDOccurrences with URNs, prime IDs, and kind-specific occurrences.

SPOs / Kinds

Data

Kinds

Schema

Statements Dataflow: ContentType Representation Model Graphs (Kind augmented Statements).

Statements

Model Subjects, Predicates, and Objects IDs with typed statement IDoccurrence(s).

SPOs

Model Subjects, Predicates, and Objects IDs with typed statement IDOccurrence(s).

Kinds

Entities & Implementation:

Kind: A set of IDOccurrences that share common structural properties. A SubjectKind like :Customer is formed by grouping all Subjects that interact with a similar set of (Predicate, Object) pairs.

Aggregate kinds for Statement's Subjects, Predicates and Objects into concept hierarchies. Kinds aggregate IDs IDOccurrence(s) (SPOs) Attributes and Values, thus performing a very simple Resource Type (attributes) and State (values) inference.

In the case of a SubjectKind, its attributes are its occurrences Predicates and its values are its occurrences Objects.

In the case of a PredicateKind, its attributes are its occurrences Subjects and its values are its occurrences Objects.

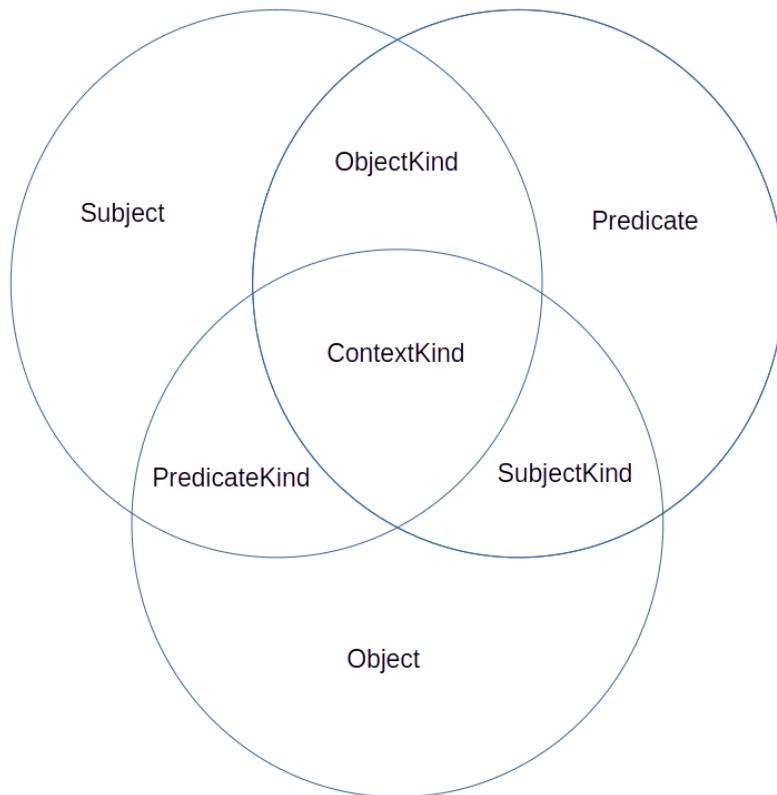
In the case of an ObjectKind, its attributes are its occurrences Predicates and its values are its occurrences Subjects.

Kind hierarchies occur in the case that a Kind attributes / values are in a superset / subset relationship.

Instance Aggregation

Aggregate kind occurrences into Graph(s) and Model(s) instances for traversal and matching.

SPO / Kinds Sets Layout



Model Abstractions

Define IDs, occurrences, SPO structures, and kinds for schema aggregation.

Content Types

Models:

Schema (Model): Upper Alignment. (SubjectKind, PredicateKind, ObjectKind) Statements Graphs.

Instances (Model): Kind aggregated (Subject, Predicate, Object) Statements Graphs.

Composite Model Graphs Statements. Example: (Employee : Kind, :salary : Predicate, 10K) : Criteria. (Employee : Kind, :salary : Predicate, GreaterThan : ComparisonKind).

Built in Schema Kinds (Alignment): Relationship / Role / Player / Transform (Relationship) / Data / Information / Knowledge / Comparison.

Streams Dataflow: Models Merge.

TODO

Inferred Kinds Schema Alignment

Organize relationship, role, and player kinds and align upper ontologies. Statements composed by Kinds (SPOs).

Inference & Traversal (Functional Interfaces):

Capability: "Given the :Customer type, what types of actions can they perform?"

Schema (Model): Upper Alignment. (SubjectKind, PredicateKind, ObjectKind) Statements Graphs.

Instances (Model): Kind aggregated (Subject, Predicate, Object) Statements Graphs.

Relationships and Events upper Schema (Kinds) Alignment

Relationships (Events / Roles / Players):

Schema:

(Relationship, Role, Player);

Relationship, Role, Player : Kinds.

Examples:

(Promotion, Promoted, Employee);

(Marriage, Married, Person);

Relationship, Role, Player attributes: from Kinds definitions. Example: Married.marryDate : Date.

Events: Relationship Transforms (Roles)

Schema:

(SourceRole, Transform, DestRole);
SourceRole, Transform, DestRole : Kinds.

Examples:

(Developer, Promotion, Manager);
(Single, Marriage, Married);

SourceRole, Transform, DestRole attributes: from Kinds definitions. Example:
Manager.projects : Project[].

Infer Relationship / Roles / Players / Events / Transforms schema Kinds (upper Alignment Kinds). Order Alignment.

Infer / Align Relationship / Roles / Players / Events / Transforms Instances (from aligned Kinds schema attributes occurrences). Attributes resolution from context: Ontology matching / Link prediction.

Streams Dataflow:

ResourceOccurrence onOccurrence chain plus getters and helper services: schema, instances, resolution inference.

Example:

TransformKind::onOccurrence(SourceKind) : DestKind;
RelationshipKind::onOccurrence(RoleKind) : PlayerKind;

Traverse Kinds / Instances (ResourceOccurrence functional chain).

Roles Promotion: From Resource Monad bound Transforms.

TODO

Models (Kinds Alignment): Definitions, Aligned schemas (attributes) and Model Instances.

Kinds: Upper alignment concepts. Aligned Kinds.

Statements: Upper schemas, aligned Kinds and Instance occurrences.

Resource Monad API Semantics: i.e.: Roles Promotion.

ContentType / Representation (Model Graphs Statements).

FCA:

(Context, Object, Attribute);

(expand: positions. Attributes: align / match).

Relationships / Events:

(Relationship, Role, Player);

(Role, EventTransform, Role);

(expand: positions. Attributes: align / match)

Dimensional (base upper ontology?):

Data Statements.

Information Statements.

Knowledge Statements.

DOM:

Type / Instance Statements.

DCI:

Context / Interaction Statements.

Actor / Role Statements.

(XSalaryEmployee, SalayRaise, YSalaryEmployee); RaiseAmount Relationship with pattern matching (rule execution). Roles are SubjectKinds with their corresponding Kinds in the Statement context.

Upper Ontologies (Models Alignment):

Reified models types (Kinds): :Statement, :Subject, :SubjectKind, etc.

Pattern Statements: (MatchingKind : PatternKind, MatchingKind : PatternKind, MatchingKind : PatternKind); Recursive: PatternKind as MatchingKind.

PatternTransform: Relationship: (PatternTransform, Role, Player). Objects Attributes (types) / Values (state) Matching.

Event: (MatchingKind, PatternTransform, PatternKind); PatternTransform: Kind => Kind.

Pattern Statements: (PatternTransform, PatternTransform, PatternTransform);

Relationships Roles / Players Reification: (Role, Role, Player); (Player, Role, Player);

(Marriage, Married, Person);

(Married, Spouse, Person);

(Person, Marriage, Spouse) : Event / Transform.

Functional helpers:

SPO / Kinds::onOccurrence(Statement) : IDOccurrence;

SPO / Kinds::getOccurrences(S, P, O) : Statements;

Statement::getOccurringContexts(S, P, O) : SPO / Kinds;

SPO / Kinds::getOccurringContexts(S, P, O) : IDOccurrence;

FCA:

(Context, Object, Attribute) : SPO / Kinds

Patterns:

(:Predicate : Context, :Subject : Object, :Object : Attribute);

(:Subject : Context, :Predicate : Object, :Object : Attribute);

(:Object : Context, :Predicate : Object, :Subject : Attribute);

(Concept, Objects, Attributes);

(:Kind : Concept, :Kind : Objects, :Kind : Attributes);

Relationship: (:Employment : PredicateKind, :Employee : SubjectKind, :Person : ObjectKind);

DOM

Dimensional / Comparisons

Relationships

Events / Transforms

DCI (Actor / Role)

Relationships and Alignment

Define composed relations (e.g., knowsLanguage) and leverage reasoners for closure.

Dimensional Alignment

Align data, information, and knowledge layers to support comparisons and events.

Dimensional Upper Model Kinds. Relationships / Events inference.

Data:

Measures. Players.

Information:

Dimensions: Measures in Context. Roles.

Knowledge:

Measures in Context inferred Relationships / Events (Transforms, from State Comparisons / Order).

Relationships / Events order / closures.

Algebraic Embeddings

CPPE Embeddings

FCA-based Embeddings: A Deterministic Approach

We will replace LLM-based embeddings with deterministic, structural embeddings derived from FCA contexts and prime number products. This provides explainable similarity based on shared roles and relationships.

- **Contextual Prime Product Embedding (CPPE):** For any IDOccurrence (i.e., a resource in a specific statement), we can calculate an embedding based on its relational context.
 1. **Define FCA Contexts:** For a given relation (predicate), we can form an FCA context. Example: For the predicate :worksFor:
 - **Objects (G):** The set of all subjects of :worksFor statements (e.g., {id:Alice, id:Bob}).
 - **Attributes (M):** The set of all objects of :worksFor statements (e.g., {id:Google, id:StartupX}).
 2. Calculate Prime Product: The CPPE for id:Google within the :worksFor context is the product of the primeIDs of all employees who work there.
$$\text{CPPE}(\text{Google, worksFor}) = \text{primeID(Alice)} * \text{primeID(Bob)} * \dots$$
- **Similarity Calculation & Inference:**
 - **Similarity:** The similarity between two entities in the same context is the **Greatest Common Divisor (GCD)** of their CPPEs. $\text{GCD}(\text{CPPE(Google)}, \text{CPPE(StartupX)})$ reveals the primeID product of their shared employees, giving a measure of personnel overlap.
 - **Relational Inference:** We can infer complex relationships. Consider the goal of finding an "uncle".
 1. Calculate the CPPE for "Person A" in the :brotherOf context (the product of

- their siblings' primes).
2. Calculate the CPPE for "Person B" in the :fatherOf context (the product of their children's primes).
 3. If $\text{GCD}(\text{CPPE}_{\text{brotherOf}}(A), \text{CPPE}_{\text{fatherOf}}(B)) > 1$, it means A is the brother of B's father. The system can then materialize a new triple: (A, :uncleOf, ChildOfB). This inference is stored and queryable.

FCA-based Relational Schema Inference

The system can infer relational schemas (rules or "upper concepts") from the structure of the data itself using FCA.

- **FCA Contexts for Relational Analysis:** We use three types of FCA contexts to analyze relationships from different perspectives:
 1. **Predicate-as-Context:** (G: Subjects, M: Objects, I: relation). This context reveals which types of subjects relate to which types of objects for a given predicate.
 2. **Subject-as-Context:** (G: Predicates, M: Objects, I: relation). This reveals all the relationships and objects associated with a given subject, defining its role.
 3. **Object-as-Context:** (G: Subjects, M: Predicates, I: relation). This reveals all the subjects and actions that affect a given object.
- **Algorithm: Inferring Relational Schema:**
 1. **Select Context:** For a given predicate P (e.g., :worksOn), the Alignment Service constructs the Predicate-as-Context.
 2. **Build Lattice:** It uses an FCA library (e.g., fcalib) to compute the concept lattice from this context.
 3. **Identify Formal Concepts:** Each node in the lattice is a *formal concept* (A, B), where A is a set of subjects (the "extent") and B is the set of objects they all share (the "intent").
 4. **Materialize Schema:** Each formal concept represents an inferred relational schema or "upper concept". The system creates a new RDF class for this concept. For a concept where the extent is {dev1, dev2} (both :Developers) and the intent is {projA, projB} (both :Projects), the system can materialize a schema:
`:DeveloperWorksOnProject a rdfs:Class, :RelationalSchema ;
:hasDomain :Developer ;
:hasRange :Project .`

Relational Context Vectors

The Relational Context Vector (RCV):

The core of this approach is the **Relational Context Vector (RCV)**. For any given statement (a reified triple), we compute a vector of three BigInteger values, (S, P, O). Each component is a CPPE calculated from one of the three FCA context perspectives, providing a holistic numerical signature of the statement's role in the graph.

- **RCV Definition:** $\text{RCV}(\text{statement}) = (S, P, O)$

 **S (Subject Context Embedding)**: The CPPE of the statement's **subject** from the **Subject-as-Context** perspective. This number encodes *everything the subject does*.

- $S = \text{calculateCPPE}(\text{statement.subject}, \text{SubjectAsContext})$

 **P (Predicate Context Embedding)**: The CPPE of the statement's **predicate** from the **Predicate-as-Context** perspective. This number encodes *every subject-object pair the predicate connects*.

- $P = \text{calculateCPPE}(\text{statement.predicate}, \text{PredicateAsContext})$

 **O (Object Context Embedding)**: The CPPE of the statement's **object** from the **Object-as-Context** perspective. This number encodes *everything that happens to the object*.

- $O = \text{calculateCPPE}(\text{statement.object}, \text{ObjectAsContext})$

- **Implementation**: A Java record RCV(BigInteger s, BigInteger p, BigInteger o). The **Index Service** is responsible for calculating and caching the RCV for every reified statement in the graph.

Schema Archetypes

This dual representation is key to performing inference.

- **Instance RCV**: The RCV calculated for a specific, concrete statement (e.g., stmt_123: (dev:Alice, :worksOn, proj:Orion)) is its unique numerical signature. It represents a single data point.
- **Schema RCV (Archetype)**: The RCV for a *relational schema* (e.g., the :DeveloperWorksOnProject schema) is an "archetype" vector. It is calculated by finding the **Least Common Multiple (LCM)** of the corresponding components of all instance RCVs that belong to that schema.

 **Algorithm: calculateSchemaRCV(schemaURI)**

1. Find all instance statements s_i where $s_i \text{ rdf:type schemaURI}$.
2. For each instance s_i , retrieve its cached $\text{RCV}_i = (S_i, P_i, O_i)$.
3. Calculate the schema RCV components:
 - $S_{\text{schema}} = \text{LCM}(S_1, S_2, \dots, S_n)$
 - $P_{\text{schema}} = \text{LCM}(P_1, P_2, \dots, P_n)$
 - $O_{\text{schema}} = \text{LCM}(O_1, O_2, \dots, O_n)$
4. The result $(S_{\text{schema}}, P_{\text{schema}}, O_{\text{schema}})$ is the numerical archetype for the schema. The LCM ensures that the schema's numerical signature is "divisible" by all of its instances.

Subsumption

Subsumption / Instance Checking (rdf:type):

- **Concept**: An instance belongs to a schema if the instance's RCV "divides into" the schema's RCV.
- **Algorithm: isInstanceOf(instanceRCV, schemaRCV)**

1. Perform a component-wise modulo operation.
 2. boolean isS = schemaRCV.s.mod(instanceRCV.s).equals(BigInteger.ZERO);
 3. boolean isP = schemaRCV.p.mod(instanceRCV.p).equals(BigInteger.ZERO);
 4. boolean isO = schemaRCV.o.mod(instanceRCV.o).equals(BigInteger.ZERO);
 5. Return isS && isP && isO.
- **Use Case:** This is a high-speed, purely numerical method for checking type constraints, which can be performed in memory without a complex graph query.

Property Chains

Define composed relations (e.g., `knowsLanguage`) and leverage reasoners for closure.

This section details the specific numerical algorithm for the `(:Developer)-[:worksOn]->(:Project)` and `(:Project)-[:usesLanguage]->(:Language)` ==> `(:Developer)-[:knowsLanguage]->(:Language)` inference.

- Step 1: Define the Composition Operator

We need a mathematical operator `compose(RCV1, RCV2)` that takes the numerical signatures of the two source relationships and produces the signature of the inferred one. The key is how the contexts are combined. The linking element is the object of the first statement (Project) and the subject of the second.

 **Inferred Subject (S_inferred):** The new subject is the original subject (Developer). Its context is expanded by the context of the final object (Language).

This represents that the developer's role is now influenced by the languages of the projects they work on.

- $S_{inferred} = RCV1.s * RCV2.o$

 **Inferred Object (O_inferred):** The new object is the original object (Language).

Its context is expanded by the context of the original subject (Developer). This represents that the language's role is now influenced by the developers who use it.

- $O_{inferred} = RCV1.s * RCV2.o$

 **Inferred Predicate (P_inferred):** The new predicate (`knowsLanguage`) is a direct composition of the original two (`worksOn` and `usesLanguage`). Its numerical signature is their product.

- $P_{inferred} = RCV1.p * RCV2.p$

- **Step 2: Calculate Schema Archetypes**

 The **Alignment Service** first calculates the archetypal RCVs for the source schemas using the LCM method described in E.2:

- $RCV_worksOn_schema = (S_{wo}, P_{wo}, O_{wo})$
- $RCV_usesLang_schema = (S_{ul}, P_{ul}, O_{ul})$

 It then calculates the archetypal RCV for the *inferred schema* (`knowsLanguage`) using the composition operator:

- $S_{kl} = S_{wo} * O_{ul}$
- $P_{kl} = P_{wo} * P_{ul}$
- $O_{kl} = S_{wo} * O_{ul}$

- This resulting RCV_knowsLang_schema = (S_kl, P_kl, O_kl) is stored as the numerical definition of the knowsLanguage rule.
- Step 3: The Inference Algorithm at Query Time
A user asks: "Does dev:Alice know lang:Java?"
 1. **Retrieve Instance RCVs:** The system retrieves the cached RCVs for the two prerequisite statements from the **Index Service**:
 - RCV1 for (dev:Alice, :worksOn, proj:Orion)
 - RCV2 for (proj:Orion, :usesLanguage, lang:Java)
 2. **Calculate Hypothetical Instance RCV:** The system calculates the numerical signature of the *potential* inferred relationship by applying the composition operator to the instance RCVs:
 - RCV_hypothetical = compose(RCV1, RCV2)
 3. **Retrieve Schema Archetype:** The system retrieves the pre-calculated archetypal RCV_knowsLang_schema.
 4. **Perform Numerical Check:** It uses the isInstanceOf algorithm to check if the hypothetical instance conforms to the general rule:
 - boolean knows = isInstanceOf(RCV_hypothetical, RCV_knowsLang_schema)
 5. **Result:** If knows is true, the inference is validated. The system has proven that Alice knows Java by showing that the specific numerical context of her work on the project aligns with the general numerical rule of how skills are acquired, all without performing a costly multi-hop graph traversal at query time.

Relationships and Alignment

Define composed relations (e.g., knowsLanguage) and leverage reasoners for closure.

Querying and Traversal by Numerical Properties:

This numerical representation unlocks new ways to query the graph.

- **Find by Relational Role:** "Find all entities that have acted as a :Developer".
 - Instead of ?x a :Developer, we can query numerically. First, calculate the archetypal RCV for the :DeveloperWorksOnProject schema. Let this be RCV_dev_schema.
 - The query becomes: "Find all statements whose instanceRCV.s component divides RCV_dev_schema.s." This finds all statements where the subject is playing a role consistent with being a developer.
- **Traversal by Numerical Similarity:**
 - Start at a given statement stmt_A. Calculate its RCV_A.
 - The next step in the traversal could be: "Find the statement stmt_B in the graph whose RCV_B has the highest GCD with RCV_A."
 - This allows for a novel form of graph traversal where the path is not defined by explicit links, but by moving from one numerically similar relational context to the next. This could be used to discover analogous processes or events across different domains within the integrated enterprise data.

Appendix E: Numerical Representation and Inference of Relational Schemas

This appendix details a novel method for representing both relational schemas (rules) and instances (data) as numerical vectors, enabling inference, querying, and traversal through direct mathematical operations. This elevates the CPPE concept to a new level of abstraction.

E.1. The Relational Context Vector (RCV)

The core of this approach is the **Relational Context Vector (RCV)**. For any given statement (a reified triple), we compute a vector of three BigInteger values, (S, P, O). Each component is a CPPE calculated from one of the three FCA context perspectives, providing a holistic numerical signature of the statement's role in the graph.

- **RCV Definition:** $\text{RCV}(\text{statement}) = (\text{S}, \text{P}, \text{O})$
 - **S (Subject Context Embedding):** The CPPE of the statement's **subject** from the **Subject-as-Context** perspective. This number encodes *everything the subject does*.
 - $\text{S} = \text{calculateCPPE}(\text{statement.subject}, \text{SubjectAsContext})$
 - **P (Predicate Context Embedding):** The CPPE of the statement's **predicate** from the **Predicate-as-Context** perspective. This number encodes *every subject-object pair the predicate connects*.
 - $\text{P} = \text{calculateCPPE}(\text{statement.predicate}, \text{PredicateAsContext})$
 - **O (Object Context Embedding):** The CPPE of the statement's **object** from the **Object-as-Context** perspective. This number encodes *everything that happens to the object*.
 - $\text{O} = \text{calculateCPPE}(\text{statement.object}, \text{ObjectAsContext})$
- **Implementation:** A Java record $\text{RCV}(\text{BigInteger s, BigInteger p, BigInteger o})$. The **Index Service** is responsible for calculating and caching the RCV for every reified statement in the graph.

E.2. Numerical Representation of Schema vs. Instance

This dual representation is key to performing inference.

- **Instance RCV:** The RCV calculated for a specific, concrete statement (e.g., stmt_123: (dev:Alice, :worksOn, proj:Orion)) is its unique numerical signature. It represents a single data point.
- **Schema RCV (Archetype):** The RCV for a *relational schema* (e.g., the :DeveloperWorksOnProject schema) is an "archetype" vector. It is calculated by finding the **Least Common Multiple (LCM)** of the corresponding components of all instance RCVs that belong to that schema.

● **Algorithm: calculateSchemaRCV(schemaURI)**

- Find all instance statements s_i where $s_i \text{ rdf:type schemaURI}$.
- For each instance s_i , retrieve its cached $\text{RCV}_i = (\text{S}_i, \text{P}_i, \text{O}_i)$.
- Calculate the schema RCV components:
 - $\text{S}_{\text{schema}} = \text{LCM}(\text{S}_1, \text{S}_2, \dots, \text{S}_n)$

- $P_{\text{schema}} = \text{LCM}(P_1, P_2, \dots, P_n)$
- $O_{\text{schema}} = \text{LCM}(O_1, O_2, \dots, O_n)$
- The result $(S_{\text{schema}}, P_{\text{schema}}, O_{\text{schema}})$ is the numerical archetype for the schema. The LCM ensures that the schema's numerical signature is "divisible" by all of its instances.

E.3. Inference via Mathematical Operators

With schemas and instances represented numerically, inference becomes a set of direct mathematical tests.

E.3.1. Subsumption / Instance Checking (rdf:type)

- **Concept:** An instance belongs to a schema if the instance's RCV "divides into" the schema's RCV.
- **Algorithm: `isInstanceOf(instanceRCV, schemaRCV)`**
 1. Perform a component-wise modulo operation.
 2. boolean `isS` = `schemaRCV.s.mod(instanceRCV.s).equals(BigInteger.ZERO);`
 3. boolean `isP` = `schemaRCV.p.mod(instanceRCV.p).equals(BigInteger.ZERO);`
 4. boolean `isO` = `schemaRCV.o.mod(instanceRCV.o).equals(BigInteger.ZERO);`
 5. Return `isS && isP && isO`.
- **Use Case:** This is a high-speed, purely numerical method for checking type constraints, which can be performed in memory without a complex graph query.

E.3.2. Numerical Inference of Attribute Closure (knowsLanguage)

This section details the specific numerical algorithm for the `(:Developer)-[:worksOn]->(:Project)` and `(:Project)-[:usesLanguage]->(:Language) ==> (:Developer)-[:knowsLanguage]->(:Language)` inference.

- **Step 1: Define the Composition Operator**
We need a mathematical operator `compose(RCV1, RCV2)` that takes the numerical signatures of the two source relationships and produces the signature of the inferred one. The key is how the contexts are combined. The linking element is the object of the first statement (Project) and the subject of the second.
 - **Inferred Subject (S_{inferred})**: The new subject is the original subject (Developer). Its context is expanded by the context of the final object (Language). This represents that the developer's role is now influenced by the languages of the projects they work on.
 - $S_{\text{inferred}} = RCV1.s * RCV2.o$
 - **Inferred Object (O_{inferred})**: The new object is the original object (Language). Its context is expanded by the context of the original subject (Developer). This represents that the language's role is now influenced by the developers who use it.
 - $O_{\text{inferred}} = RCV1.s * RCV2.o$
 - **Inferred Predicate (P_{inferred})**: The new predicate (`knowsLanguage`) is a direct composition of the original two (`worksOn` and `usesLanguage`). Its numerical signature is their product.
 - $P_{\text{inferred}} = RCV1.p * RCV2.p$
- **Step 2: Calculate Schema Archetypes**

- The **Alignment Service** first calculates the archetypal RCVs for the source schemas using the LCM method described in E.2:
 - $\text{RCV_worksOn_schema} = (\text{S_wo}, \text{P_wo}, \text{O_wo})$
 - $\text{RCV_usesLang_schema} = (\text{S_ul}, \text{P_ul}, \text{O_ul})$
- It then calculates the archetypal RCV for the *inferred schema* (`knowsLanguage`) using the composition operator:
 - $\text{S_kl} = \text{S_wo} * \text{O_ul}$
 - $\text{P_kl} = \text{P_wo} * \text{P_ul}$
 - $\text{O_kl} = \text{S_wo} * \text{O_ul}$
- This resulting $\text{RCV_knowsLang_schema} = (\text{S_kl}, \text{P_kl}, \text{O_kl})$ is stored as the numerical definition of the `knowsLanguage` rule.
- Step 3: The Inference Algorithm at Query Time
 A user asks: "Does dev:Alice know lang:Java?"
 1. **Retrieve Instance RCVs:** The system retrieves the cached RCVs for the two prerequisite statements from the **Index Service**:
 - RCV1 for (dev:Alice, :worksOn, proj:Orion)
 - RCV2 for (proj:Orion, :usesLanguage, lang:Java)
 2. **Calculate Hypothetical Instance RCV:** The system calculates the numerical signature of the *potential* inferred relationship by applying the composition operator to the instance RCVs:
 - $\text{RCV_hypothetical} = \text{compose}(\text{RCV1}, \text{RCV2})$
 3. **Retrieve Schema Archetype:** The system retrieves the pre-calculated archetypal $\text{RCV_knowsLang_schema}$.
 4. **Perform Numerical Check:** It uses the `isInstanceOf` algorithm to check if the hypothetical instance conforms to the general rule:
 - $\text{boolean knows} = \text{isInstanceOf}(\text{RCV_hypothetical}, \text{RCV_knowsLang_schema})$
 5. **Result:** If `knows` is true, the inference is validated. The system has proven that Alice knows Java by showing that the specific numerical context of her work on the project aligns with the general numerical rule of how skills are acquired, all without performing a costly multi-hop graph traversal at query time.

E.4. Querying and Traversal by Numerical Properties

This numerical representation unlocks new ways to query the graph.

- **Find by Relational Role:** "Find all entities that have acted as a :Developer".
 - Instead of `?x a :Developer`, we can query numerically. First, calculate the archetypal RCV for the `:DeveloperWorksOnProject` schema. Let this be RCV_dev_schema .
 - The query becomes: "Find all statements whose `instanceRCV.s` component divides RCV_dev_schema.s ." This finds all statements where the subject is playing a role consistent with being a developer.
- **Traversal by Numerical Similarity:**
 - Start at a given statement `stmt_A`. Calculate its RCV_A .
 - The next step in the traversal could be: "Find the statement `stmt_B` in the graph whose RCV_B has the highest GCD with RCV_A ."
 - This allows for a novel form of graph traversal where the path is not defined by explicit links, but by moving from one numerically similar relational context to the

next. This could be used to discover analogous processes or events across different domains within the integrated enterprise data.

Functional Resources Approach

Resource Monad API

The Resource Monad:
Functional wrapper

Resource Monad: `Resource<ResourceOccurrence>`.

Wraps successive `ResourceOccurrence` class hierarchy occurrence events, getter and context methods.

Provide functional wrappers (`Resource<ResourceOccurrence>`) for occurrence events and accessors. Wraps successive `ResourceOccurrence` class hierarchy occurrence events, getter and context methods.

ContentType (Representations Transforms)

- Transforms (XSLT / Custom Logic) for each `ContentType` type instance
- Model Types:
 1. FCA
 2. DOM (OGM)
 3. Activation (Actor / Role)
- Resource Occurrence Types:
 1. `ResourceOccurrence` Classes
- Encoding Types:
 1. Reference (Topic Maps TMRM)
 2. RDF / RDFS
 3. JSON-LD

Representation : `ContentType` instance

- `ContentType`
- Encoded State (XML / Custom Classes)

`ResourceOccurrence`

- Representation
- Methods (Dispatch to Representation `ContentType` Transforms):
 -  `onOccurrence(ResourceOccurrence occurrence) : ResourceOccurrence context (event)`
 -  `getOccurrences(S, P, O)`
 -  `getOccurringContexts(S, P, O)`

-  `getAttributes()` : Attributes (by means of occurrences / schema)
 - `getAttribute(Attribute)`
 - `setAttribute(Attribute, Value)`
- Hierarchies (TODO) : ContentType hierarchies?

ResourceOccurrence(s) Classes.

Embed in URNs / ContentTypes ResourceOccurrence instance type and context instance ID.
Example: urn:graph:subjectKind1. Dynamic Kinds ContentTypes.

ResourceOccurrence(s) Activation (ContentType handled, Resource Monad bound):

ResourceOccurrence Events:

`ResourceOccurrence::onOccurrence(ResourceOccurrence occurrence)` : ResourceOccurrence context.

ResourceOccurrence Occurrences:

`ResourceOccurrence::getOccurrences(S, P, O)` : ResourceOccurrence. S, P, O filter /criteria / matching.

Leverages CPPE / RCV / FCA / Kinds / Alignment schema / instances inference / filter / query / traversal.

ResourceOccurrence Occurring Contexts:

`ResourceOccurrence::getOccurringContext(S, P, O)` : ResourceOccurrence. S, P, O filter /criteria / matching.

Leverages CPPE / RCV / FCA / Kinds / Alignment schema / instances inference / filter / query / traversal.

Reactive Runtime

Core Streaming Layout

Single Topic Architecture:

All Services publish and consume messages from a single topic. This allows for further message augmentation.

Blackboard design pattern:

Services decide which and how to handle specific messages according to its criteria (ContentType for example).

Nodes Functional Reactive Behavior

- Consume Augmented Model (Representation?)
- Functional Input Model Traversal (Representation unfolding):
- `Representation::getOccurrences(S, P, O)` : ResourceOccurrence
- `Representation::onOccurrence(ResourceOccurrence)`

- ContentType::onOccurrence(Representation) : Model
- Model::onOccurrence(ContentType) : Graph
- Graph::onOccurrence(Model) : Statement
- Statement::onOccurrence(Graph) : SPO / Kinds
- SPO / Kinds::onOccurrence(Statement) : IDOccurrence
- IDOccurrence::onOccurrence(SPO / Kinds) : ID
- ID::onOccurrence(IDOccurrence) : URN
- Functional Output Model Building (Representation folding):
- ID::getOccurrences(S, P, O) : IDOccurrence
- IDOccurrence::getOccurrences(S, P, O) : SPO / Kinds
- SPO / Kinds::getOccurrences(S, P, O) : Statements
- Statement::getOccurrences(S, P, O) : Graphs
- Graph::getOccurrences(S, P, O) : Models
- Model::getOccurrences(S, P, O) : ContentType
- ContentType::getOccurrences(S, P, O) : Representation
- Representation::getOccurrences(S, P, O) : ResourceOccurrence
- Publish Augmented Model (Representation?)

MESSAGES:

Messages (Services and Models Statements exchange): Services / Components interactions and Registry Models storage is in the form of Reference Model Statements. Use Reference Model Statements as an implementation of an underlying model representation for Statement Messages exchange between Services and Models and persistence in the Model Service. Each service leverages the model type (view) most appropriate for its task.

Runtime:

Events: Model Messages.

Main Event Loop:

Aggregation, Alignment, Activation stream nodes Model Events Topic consumers / producers. Matches for Models ContentType(s).

Topic streaming:

Stream nodes consume Model (Representation?) Events and publish augmented Model (Representation?) Events Context back to the stream for further augmentation. Augmentation nodes update Model (Representation?) ContentType(s).

Resource Activation: each stream node unfolds consumed Model (Representation?) Event and invokes occurrences events, traversing occurrences / occurring contexts getters. Node augmentation logic in Resources Representations ContentType(s) events transforms.

Datasource node: Produces Models (Representation?) Events published to the topic and listens for Model (Representation?) Events for syncing back backends state.

Producer node: consumes Model (Representation?) Events, publishes Activation API from Models and produces API interactions Model Events.

- Augmented Model in Events Context
- Aggregation Node: type / state / order inference. FCA Model.
- Augmentation Node unfolded Model Context ResourceOccurrence events stream (ContentType transforms):
- Functional Input Model Traversal (Representation unfolding):
 1. Representation::getOccurrences(S, P, O) : ResourceOccurrence
 2. Representation::onOccurrence(ResourceOccurrence)
 3. ContentType::onOccurrence(Representation) : Model
 4. Model::onOccurrence(ContentType) : Graph
 5. Graph::onOccurrence(Model) : Statement
 6. Statement::onOccurrence(Graph) : SPO / Kinds
 7. SPO / Kinds::onOccurrence(Statement) : IDOccurrence
 8. IDOccurrence::onOccurrence(SPO / Kinds) : ID
 9. ID::onOccurrence(IDOccurrence) : URN
- Functional Output Model Building (Representation folding):
 1. ID::getOccurrences(S, P, O) : IDOccurrence
 2. IDOccurrence::getOccurrences(S, P, O) : SPO / Kinds
 3. SPO / Kinds::getOccurrences(S, P, O) : Statements
 4. Statement::getOccurrences(S, P, O) : Graphs
 5. Graph::getOccurrences(S, P, O) : Models
 6. Model::getOccurrences(S, P, O) : ContentType
 7. ContentType::getOccurrences(S, P, O) : Representation
 8. Representation::getOccurrences(S, P, O) : ResourceOccurrence
- Publish Augmented Model (Representation?)
- Alignment Node: equivalence / upper ontology alignment, link prediction. DOM (OGM) Model.
- Augmentation Node unfolded Model Context ResourceOccurrence events stream (ContentType transforms):
- Functional Input Model Traversal (Representation unfolding):
 1. Representation::getOccurrences(S, P, O) : ResourceOccurrence
 2. Representation::onOccurrence(ResourceOccurrence)
 3. ContentType::onOccurrence(Representation) : Model
 4. Model::onOccurrence(ContentType) : Graph
 5. Graph::onOccurrence(Model) : Statement
 6. Statement::onOccurrence(Graph) : SPO / Kinds
 7. SPO / Kinds::onOccurrence(Statement) : IDOccurrence
 8. IDOccurrence::onOccurrence(SPO / Kinds) : ID
 9. ID::onOccurrence(IDOccurrence) : URN
- Functional Output Model Building (Representation unfolding):
 1. ID::getOccurrences(S, P, O) : IDOccurrence
 2. IDOccurrence::getOccurrences(S, P, O) : SPO / Kinds
 3. SPO / Kinds::getOccurrences(S, P, O) : Statements
 4. Statement::getOccurrences(S, P, O) : Graphs
 5. Graph::getOccurrences(S, P, O) : Models
 6. Model::getOccurrences(S, P, O) : ContentType

- 7. ContentType::getOccurrences(S, P, O) : Representation
 8. Representation::getOccurrences(S, P, O) : ResourceOccurrence
- Publish Augmented Model (Representation?)
- Activation Node: possible verbs / state changes / transforms. DCI (Actor / Role) Model.
- Augmentation Node unfolded Model Context ResourceOccurrence events stream (ContentType transforms):
- Functional Input Model Traversal (Representation unfolding):
 1. Representation::getOccurrences(S, P, O) : ResourceOccurrence
 2. Representation::onOccurrence(ResourceOccurrence)
 3. ContentType::onOccurrence(Representation) : Model
 4. Model::onOccurrence(ContentType) : Graph
 5. Graph::onOccurrence(Model) : Statement
 6. Statement::onOccurrence(Graph) : SPO / Kinds
 7. SPO / Kinds::onOccurrence(Statement) : IDOccurrence
 8. IDOccurrence::onOccurrence(SPO / Kinds) : ID
 9. ID::onOccurrence(IDOccurrence) : URN
- Functional Output Model Building (Representation folding):
 1. ID::getOccurrences(S, P, O) : IDOccurrence
 2. IDOccurrence::getOccurrences(S, P, O) : SPO / Kinds
 3. SPO / Kinds::getOccurrences(S, P, O) : Statements
 4. Statement::getOccurrences(S, P, O) : Graphs
 5. Graph::getOccurrences(S, P, O) : Models
 6. Model::getOccurrences(S, P, O) : ContentType
 7. ContentType::getOccurrences(S, P, O) : Representation
 8. Representation::getOccurrences(S, P, O) : ResourceOccurrence
- Publish Augmented Model (Representation?)
- Naming Helper
- Registry Helper
- Index Helper

Functional Dataflow Operations

Route onOccurrence handling to representation content types for SPO and kinds.

ID::onOccurrence(IDOccurrence) : URN
 IDOccurrence::onOccurrence(SPO / Kinds) : ID
 SPO / Kinds::onOccurrence(Statement) : IDOccurrence
 Statement::onOccurrence(Graph) : SPO / Kinds
 Graph::onOccurrence(Model) : Statement
 Model::onOccurrence(ContentType) : Graph (merge)
 ContentType::onOccurrence(Representation) : Model
 Representation::onOccurrence(ResourceOccurrence) : ContentType

Functional Retrieval / Traversal Operations

Support getOccurrences and getOccurringContexts across IDs, statements, graphs, models, and representations.

Representation::getOccurrences(S, P, O) : ResourceOccurrence

ContentType::getOccurrences(S, P, O) : Representation

Model::getOccurrences(S, P, O) : ContentType

Graph::getOccurrences(S, P, O) : Models

Statement::getOccurrences(S, P, O) : Graphs

SPO / Kinds::getOccurrences(S, P, O) : Statements

IDOccurrence::getOccurrences(S, P, O) : SPO / Kinds

ID::getOccurrences(S, P, O) : IDOccurrence

ResourceOccurrence::getOccurringContexts(S, P, O) : Representation

Representation::getOccurringContexts(S, P, O) : ContentType

ContentType::getOccurringContexts(S, P, O) : Model

Model::getOccurringContexts(S, P, O) : Graphs

Graph::getOccurringContexts(S, P, O) : Statements

Statement::getOccurringContexts(S, P, O) : SPO / Kinds

SPO / Kinds::getOccurringContexts(S, P, O) : IDOccurrence

IDOccurrence::getOccurringContexts(S, P, O) : ID

ID::getOccurringContexts(S, P, O) : URN

Event Streams

Aggregation

Produce SPO and kinds aggregated statements with DIDs, prime IDs, and FCA clustering.

type / state / order inference. FCA Model.

Type / State hierarchies.

Entities with the same attributes are considered as of the same type, superset / subset of attributes: type hierarchy. Attributes with the same values, same states. Superset / subset of values / states: state hierarchy.

Types are ordered in respect to their common attributes. Most specific types (more common attributes) are considered to inherit from types with less common attributes. A more specific type is considered to be “after” a more generic type (Person → Employee). Regarding state values, hierarchies are to be considered regarding attribute values, being resources with common state grouped into hierarchies (Marital status attribute: Single →

Married → Divorced).

Consumes (ID, ID, ID) Statements; (and hierarchy).

Produces SPO / Kinds Aggregated Statements.

DIDs, Prime IDs, FCA Clustering.

Alignment

Infer equivalences, link predictions, and kind hierarchies for models and instances.

equivalence / upper ontology alignment, link prediction. DOM (OGM) Model.

Consumes SPO / Kinds Aggregated Statements (and hierarchy).

Produces Graph / Models Statements.

Models schema Statements: prepared for alignment (equivalent attributes / values).

Predicates: worksFor, hires, same SKs / OKs (Employee, Employer). Inverse relationship (infer).

Kinds hierarchy: order (set / superset of attributes).

Alignment: Find equivalent attributes for Kinds in SPO contexts (matching). Clustering.

Matching: Find equivalent values for equivalent attributes for Kinds in SPO contexts (links). Classification.

Link Prediction: superset of attributes of aligned (extending) Kinds. Infer object values. Regression.

Similar Structures Occurrences:

Graphs for each SPO Statement Kinds.

Definition Graphs: Model Roles in Statements Position.

Instance Graphs: Merge equivalent Roles (Align). Match Model Occurrences.

Materialize Same As: same / equiv attrs. (Kinds), same / equiv values (Instances). Merge Kinds / instances.

Activation

Expose contexts, interactions, actors, and roles via the activation API.

possible verbs / state changes / transforms. DCI (Actor / Role) Model.

Consumes Graph / Models Statements (and hierarchy).

Produces ContentType Representation Statements.

Contexts Interactions Actors Roles State API.

Nodes Functional Reactive Behavior

Consume Augmented Model (Representation?)

Functional Input Model Traversal (Representation unfolding):

Representation::getOccurrences(S, P, O) : ResourceOccurrence

Representation::onOccurrence(ResourceOccurrence)

ContentType::onOccurrence(Representation) : Model

Model::onOccurrence(DataType) : Graph

Graph::onOccurrence(Model) : Statement

Statement::onOccurrence(Graph) : SPO / Kinds

SPO / Kinds::onOccurrence(Statement) : IDOccurrence

IDOccurrence::onOccurrence(SPO / Kinds) : ID

ID::onOccurrence(IDOccurrence) : URN

Functional Output Model Building (Representation folding):

ID::getOccurrences(S, P, O) : IDOccurrence

IDOccurrence::getOccurrences(S, P, O) : SPO / Kinds

SPO / Kinds::getOccurrences(S, P, O) : Statements

Statement::getOccurrences(S, P, O) : Graphs

Graph::getOccurrences(S, P, O) : Models

Model::getOccurrences(S, P, O) : ContentType

ContentType::getOccurrences(S, P, O) : Representation

Representation::getOccurrences(S, P, O) : ResourceOccurrence

Publish Augmented Model (Representation?)

Events Stream and Augmentation Services

Messages : Models (Representations)

Events: Model (Representation?) Messages.

Topic Event loop / Registry: Blackboard design pattern.

Statements Dataflow: ContentType Representation Model Graphs (Kind augmented Statements).

Main Event Loop:

Aggregation, Alignment, Activation stream nodes Model Events Topic consumers / producers. Matches for Models ContentType(s).

Topic streaming:

Stream nodes consume Model (Representation?) Events and publish augmented Model (Representation?) Events Context back to the stream for further augmentation.

Augmentation nodes update Model (Representation?) ContentType(s).

Resource Activation: each stream node unfolds consumed Model (Representation?) Event and invokes occurrences events, traversing occurrences / occurring contexts getters.

Node augmentation logic in Resources Representations ContentType(s) events transforms.

Datasource node: Produces Models (Representation?) Events published to the topic and listens for Model (Representation?) Events for syncing back backends state.

Producer node: consumes Model (Representation?) Events, publishes Activation API from Models and produces API interactions Model Events.

Core Services

Model Service (Content Types)

Main RDF store persistence handler across services. Persistence of underlying Reference Model Statements and conversion back and forth between other model views handling.

Keep core Reference Model state in sync with views models interactions and handle persistence within them.

Application Service

Encloses Datasource, Augmentation, Producer Service interactions. Listen for Statements from Datasource Service / Producer Services. Dispatches to / consumes from Augmentation Service. Publish Statements to Datasource Service for backend sync / to Producer Service for Interactions state.

Augmentation Service

Encloses Aggregation, Alignment, Activation Service Interactions. Listen for Statements from Application Service. Dispatches Statements to / consumes from Aggregation, Alignment, Activation reactive functional pipeline. Publish Statements to Application Service.

All services should have an administration / management interface for each step of the workflow. Example: Add datasources, view inferred types and their instances, view aligned

upper ontologies (endpoint), view current contexts / interactions, browse available API endpoints definitions.

Services configuration is in the form of other aggregated / integrated data.

The communication between services is in the form of serialized core model statements messages and events which each service process and augments in a functional reactive manner a core model graph in the helper Registry Service, performs upper ontologies alignment and matching in the helper Naming Service and provides for a repository of aligned resources to be activated (created, retrieved and updated) in the helper Index Service.

Datasource Service

Produce and consume raw integrated datasource triples. Keep source backends consistent with interaction outcomes across integrated applications. Datasource Service: consumes raw backends data, publish Statements (for Application Service). Listen for Statements (from Application Service), synchronizes backends data.

ETL / Synchronization with the backend datasources of the integrated applications providing and consuming graph models streams handling provenance and comsumption / updating of the integrated applications backend datasources.

One basic translation from tabular data could be to represent a row in a source application database as a triple in the form: (S: Row PK value, P: Row Column Name, O: Row Column Value) for a given PK value).

Inputs / Outputs.

Aggregation Service

Infer domain schemas, states, and data from tabular, XML, JSON, and graph input triples from Datasources Service. Perform FCA / Kinds Augmentation. Augmentation Pipeline step. FCA Prime IDs assignation, FCA Contexts creation / updates / sync. W3C DIDs assignation. Type / State / Contexts / Relationship / Order (hierarchies) inference. Consumes / Updates Model Service. CPPE / RCVs handling. Consumes / Updates Model Service. Alignment / sync of augmented input Statements with Model Service.

Consumes (ID, ID, ID) Statements; (and hierarchy).

Produces SPO / Kinds Aggregated Statements.

DIDs, Prime IDs, FCA Clustering.

Consumes: DataSources (CSPO) URI Strings Statements.

Produces: Reference Model (ID, ID, ID, ID) Statements.

Features:

- IDs / Embeddings.
- FCA Contexts (Clustering).
- Basic types / attributes inference (FCA).

Given a set of raw SPO triples from Datasources Service, performs type inference (common attributes aggregation) and state inference (common attribute values aggregation) and performs type / state hierarchies inference.

Type inference: Subjects with the same Attributes belong to the same type.

State inference: Subjects with Attributes (types) with the same Values are in the same state.

Type / State hierarchies:

Entities with the same attributes are considered as of the same type, superset / subset of attributes: type hierarchy. Attributes with the same values, same states. Superset / subset of values / states: state hierarchy.

Types are ordered in respect to their common attributes. Most specific types (more common attributes) are considered to inherit from types with less common attributes included into the more specific types. A more specific type is considered to be "after" a more generic type (Person → Employee). Regarding state values, hierarchies are to be considered regarding attribute values, being resources with common state grouped into hierarchies (Marital status attribute: Single → Married → Divorced).

Order: Inferred via Type / State hierarchies. Types: Married extends from Single, Divorced extends from Married. States: Young extends from Child, Old extends from Young. Cycles in types resolved by state (Unemployed, Employed, Unemployed). Used in (2.4) Alignment Service Ordering upper ontology.

`Map<Subject, Set<Predicate>>`
`Map<Set<Predicate>, Type>`

`Map<Type, Set<Map<Predicate,Value>>>`
`Map<Set<Map<Predicate,Value>>, State>`

Inputs / Outputs: Core Model Classes Statements (see below). Leverages ML Classification.

Alignment Service

Infer Relationships, Events and Transforms and align aggregated information into this (inferred upper) ontologies. Equivalent entities ontology matching. Missing links prediction. Alignment Service: Augmentation Pipeline step. Ontology matching. Upper ontology alignment. Link / Attribute prediction. Order inference (via Dimensional Model view). Consumes / Updates Model Service. Alignment / sync of augmented input Statements with Model Service.

equivalence / upper ontology alignment, link prediction. DOM (OGM) Model.

Consumes SPO / Kinds Aggregated Statements (and hierarchy).
Produces Graph / Models Statements.

Models schema Statements: prepared for alignment (equivalent attributes / values).

Predicates: worksFor, hires, same SKs / OKs (Employee, Employer). Inverse relationship (infer).

Kinds hierarchy: order (set / superset of attributes).

Alignment: Find equivalent attributes for Kinds in SPO contexts (matching). Clustering.

Matching: Find equivalent values for equivalent attributes for Kinds in SPO contexts (links). Classification.

Link Prediction: superset of attributes of aligned (extending) Kinds. Infer object values. Regression.

Similar Structures Occurrences:

Graphs for each SPO Statement Kinds.

Definition Graphs: Model Roles in Statements Position.

Instance Graphs: Merge equivalent Roles (Align). Match Model Occurrences.

Materialize Same As: same / equiv attrs. (Kinds), same / equiv values (Instances). Merge Kinds / instances.

Consumes: Reference Model (ID, ID, ID, ID) Statements.

Produces: Graph Model (CSPO) Statements.

Features:

- Upper (inferred) ontology alignment.
- Links completion, Ontology Matching.
- Types / Kinds Inference. Order (hierarchies / dimensional).

Aligns (links / attributes, ontology matching, upper ontologies alignment) Aggregation Statements. Augments overall model.

Upper ontologies:

a) Domains: Aligned integrated application domains inferred common concepts and relationships. Infer equivalent concepts and relationships between source applications domains and populate Domains upper ontology. Materialize integrated domains concepts and relationships mappings to inferred upper concepts and relationships. Abstract common meaning (semantics) of source applications concepts and relationships to enable inter domain contexts interactions.

b) Order: Dimensional arrangement of entities attributes and values. Align measures (attribute values) into dimensional units. According to Aggregation Service types and states hierarchies establish order relationships (before, greater than, contains, etc.)

between measures. Materialize measures relationships and map dimensional units measures occurrences into the materialized order relationships. See: [5. Dimensional Features].

Ontology Matching: Find and map equivalent entities and relationships domains occurrences (Core Model Classes), align core model resources into Domains upper ontology.

Links / Attributes inference: Given an aligned model (mapped to Domains upper ontology) infer possible links / relationships between resources and possible attributes and their values.

Ordering: Order dimensional upper ontology alignment. Type / State hierarchies

Inputs / Outputs: Core Model Classes Statements (see below). Leverage ML Clustering.

Activation Service

Expose use-case contexts and interactions via a producer client and API for cross-application execution. Infer Contexts (use cases), their player roles, their Interactions (executions), their role playing Actors and their Interaction outcomes (Actors Transformations). Activation Service: Augmentation Pipeline step. Instantiates inferred use cases (DCI Activation model Contexts) executions (DCI Activation model Interactions). Enables use cases execution by means of Messages Statements exchange (state available verbs for actors transforms productions). Reference Model (corresponding views) alignment / sync of inferred use cases and their current executions. Implement COST / Server implementation classes and a sample dynamic forms Angular client application. Show an example.

possible verbs / state changes / transforms. DCI (Actor / Role) Model.

Consumes Graph / Models Statements (and hierarchy).
Produces ContentType Representation Statements.

Contexts Interactions Actors Roles State API.

Consumes: Graph Model (CSPO) Statements.
Produces: Activation Model DCI (Context, Interaction, Role, Actor) Statements.
Features: Use case inference / execution. Browse / Run transactions across integrated applications.

Activates Resources discovering from their types, states and order relationships which Use Cases (Contexts) are available in and between Resource types, states and order and which Roles are played by which types in state and order and allows to instantiate Transactions (Use Case Contexts Interactions) assigning Actors Resources to play specific Context Use Case Roles. The business logic of each Transaction (data flow) between Actors of different integrated domains applications playing Roles in a Context Interaction is to be inferred from the Alignment Service upper ontologies (Domains and Order).

Contexts, Roles, Interactions and Actors are inferred and aligned to an Activation upper ontology leveraging Alignment Service Domains and Order upper ontologies. Activation upper ontology should enable Consumer API Service to expose available Contexts, Contexts state (Interactions instances), instantiate Contexts into new Interactions and fulfill Interactions Context Roles with the playing actors for this transaction and performs any steps involved in the creation of the current transaction (steps, forms flow, wizard like interface).

Activation upper ontology should be able to be queried by the Consumer API Service to build an Context Interaction scenario given a desired Context Interaction transaction outcome, letting the Activation Service populate possible Context Roles Actors for the desired outcome and showing possible scenarios to the user. Activation upper ontology follows the guidelines of the DCI: Data, Context and Interactions design pattern, letting the part of the transactions ordered steps / invocations data flow to be inferred from the current aligned Context Interactions transactions instances materialized in a declarative fashion into the model.

Data flow encoding:

(Contexts / Roles, Interactions, Actors) : Kinds(Type, State).

(Buy, Product, Good
(Good, Price, Amount)
(aBuy, contextType, Buy) : has ContextType → Interaction
(aBuy, Product, aProduct
(aProduct, Price, anAmount)

(anAmount, buyer → seller); (aProduct, seller → buyer);

Infer / Materialize / Perform operations. Encode functional mappings: assign / transform roles attributes.

Inputs / Outputs: Core Model Classes Statements (see below). Leverages ML Regression.

Contexts, Roles / Interactions, Actors

Contexts Actions flows and actions behavior declaratively stated from inference into dynamically stated logic / dataflow (XSLT Transforms generated from inference) into flows of reactive streams. SPARQL Backend CRUD, MCP Tools / Server.

Producer Service

Exposes a Context / Interaction aware API for browsing / initiating application transactions (use case executions) and handles dynamic transactions flows forms layouts (wizard like interface: REST / HAL Conversational State Transfer: COST implemented). Producer Service: Listen for Statements (from Application Service). Publish Statements (to Application Service). Handles COST Conversations States.

Communication with the Producer API Service, unified REST APIs exposure, is between the Consumer API Service and the Augmentation Service. Augmentation Service dispatch messages and events between the Consumer API Service and the orchestrated services via Helper Services messages and events, retrieving the needed information and providing Consumer API Service with updated information (dialog conversational state).

Find relationships and equivalences between the data of the applications to be unified and their possible interactions. Use cases in and between applications.

Expose through an API the possible interactions to be invoked, their contexts roles and transactions interactions actors, and synchronize transaction data with the original applications. Provide a generic API Service front end (REST / Web). Provide a generic forms front end for rendering Contexts Interactions instances.

One should be able to ask for Contexts Interactions with a desired outcome, via inference performed determining which Actors should play which Roles in which Interactions (state, order) to achieve which Context Interactions results are desired.

Example: Launch new product to the market Context. Manufacturing, Inventory, Orders Delivery and Public Advertising integrated applications interacts as Actors with their respective roles in each step of the Launch new product to the market use case (Context) instance (Interaction).

One should be able to navigate previous Interactions (Contexts executions) or to create new ones (Contexts invocation).

Generic REST API Frontend: Exposes Activation Service Contexts Use Cases and allows to create, browse, update or continue existing Contexts Interactions transactions.

Possible Scenarios: Given a desired outcome, browse possible actors in context roles that would fulfill the desired result.

Gestures (Functions. Content Type available verbs). Domain Driven Design.

Forms / Flows: Roles Placeholders, Actor Values given Context. HATEOAS / HAL.

Inputs / Outputs: Core Model Classes Statements (see below). Leverages ML Regression.

COST (*Conversational State Transfer*)

REST API is in initial state for a given context. The client retrieves the 'current' role context dataflow representation instance (Interaction, Actor, Transform), process it (DSL, 'Activates' and invokes API for the given representation Transform) and posts back the activated representation. The service then is able to determine the next Dataflow Role representation instance in a given use case (Context). TODO: Populate (infer) Dataflow Roles rules (state flows), Populate (infer / execute) Transform Actors productions using data encoded in the proposed models.

The goal is to integrate the domains and functionality of various applications into a unified and integrated API or interface (unified front end). Given all the application / services to integrate: Extract all data sources from the applications to be integrated and represent

them in a unified way. Perform Augmentation (Aggregation, Alignment and Activation) over the source raw data and schema to achieve an unified interface exposed through an unified API Consumer Service which exposes the Contexts (Use Cases) and Interactions (Use Case executions) inferred and possible in and between integrated applications (REST API).

Helper Services

MCP Service

MCP Server and MCP Client features in a separate service module. Enable connect Application Service as an MCP Client. MCP Server Tools: Activation Contexts, client retrieves use cases. MCP Server Prompt templates: Naming Service. Textual description (with placeholders) for Activation Interaction dataflow state transform step. Client retrieves interaction state possible prompts within that state. MCP Server Resources: Registry Service, client retrieve models representations from models service for Content Types. MCP Client features: Sampling, file system roots. Leverage Naming Service APIs.

Index Service

Functionality related to inference, query, retrieval and traversal of models, as for example the FCA model (see Goal 5). Cache RCVs and related signatures for fast inference and traversal. FCA Contexts, Prime IDs, CPPE, RCVs Calculation. Similarity / Relationships Inference query / traversal API. Reference Model Inferences query / traversal API. Other models (views) Inference / query / traversal APIs.

Repository of aligned resources to be activated (created, retrieved and updated) in the Activation service via similarity resolution. Dialog state based interface (Conversational State Transfer).

Resolve possible / actual contexts / interactions given resource representations. Resolve interaction possible / populated context templates (actors in roles placeholders).

Given a Resource representation in a given context and a given verb (Content Type method), retrieve the next Resource representation in the Activation flow (form with Content Type placeholders). Consumer fills in forms placeholders and the index is asked to retrieve again the next Resource representation for a given verb in the Activation flow.

Streams / events based interfaces.

Naming Service

Functionality related to name resolution and inference, such as labeling Kinds and Relationships. LLM MCP bridge. Embed instance type and context in URNs and content

type signatures. Leverage LLMs / GenAI features (with Spring AI) for ontology matching, upper ontology alignment, links / attributes inference and order inference. Kinds, Relationships, Contexts, Roles name assignment / retrieval. MCP Server / Client features. Use CPPE / RCVs for embeddings.

TODO

Upper ontologies (Domains, Dimensional Order and Activation) matching and alignment.

Sets (See: [4 Sets Representation]) internal inference model representation. Sets API and functional set processing operations for matching and alignment tasks.

FCA (Formal Concept Analysis) contexts representation. Concepts Lattices for concepts alignment and attribute inference.

Links / relationships resolution in contexts. Attribute values, Context interaction roles. Order inference materialization.

SPARQL Endpoint. URI Based retrieval. Streams / events based interfaces.

Registry Service

Main ResourceOccurrence(s) repository. IDs generation. Leveraged by Index and Naming Services.

Track models, transforms, and dynamic kind content types. Resource Repository View. Produce / Consume Resources in provided / requested Resource Content Types (Representations). Content Type encoders / decoders for Reference Model Resources APIs. Bi directional encoding / decoding APIs (mappings). Activation Interactions Actor transforms (Resource JAF Content Types activated verbs execution).

TODO

Core graph model repository. To store / retrieve / share results of streams functional processing in each Augmentation (Aggregation, Alignment and Activation) orchestrated services. Hierarchical key / value store. TMRM (ISO Topic Maps Reference Model). Provenance repository (applications datasources synchronization). Embeddings.

SPARQL Endpoint. URI Based retrieval. Streams / events based interfaces.

Semantic Hypermedia Addressing

Given Hypermedia Resources Content Types (REST):

. Text

- . Images
- . Audio
- . Video
- . Tabular
- . Hierarchical
- . Graph

Imagine the possibility of not only annotate resources of those types with metadata and links (in the appropriate axes and occurrences context) but having those annotations and links being generated by inference and activation being that metadata and links in turn meaningful annotated with their meaning given its occurrence context in any given axis or relationship role (dimension).

RESTful principles could apply rendering annotations and links as resources also, with their annotations and links, making them discoverable and browsable / query-able. Naming conventions for standard addressable resources could make browsing and returning results (for a query or prompt, for example) a machine-understandable task.

Also, the task of constructing resources hyperlinked or embedding other resources in a content context (a report or dashboard, for example) or the frontend for a given resource driven (REST) resource contexts interactions will be a matter of discovery of the right resources and link resources.

Given the appropriate resources, link resources and addressing, encoding a prompt / query for a link, in a given context (maybe embedded within the prompt / query) would be a matter of resource interaction, being the capabilities of what can be prompted / queried for available to the client for further exploration.

Generated resources, in their corresponding Content Types, should also address and be further addressable in and by other resources, enabling incremental knowledge composition by means of preserving generated assets in a resources interaction contexts history.

User generated resources: documents, images, audio, video and even mails, chats, calendars, meetings and meeting notes, for example, would leverage this semantic addressing and being semantically addressable capabilities. Even the interactions with

(business) applications, such as the placement of an order, an ERP transaction or a CRM issue resolution, as addressing and addressable resources, will take part of this resource oriented linked knowledge network augmenting and being augmented with AI generated resources and addressing.

Wouldn't it be nice if our browsing history and bookmarks were arranged in a meaningful task and purpose oriented manner, organized in a way where it is possible to know where we are and why, how and from where did we get there?

Examples:

"Given this book, make an index with all the occurrences of this character and also provide links to the moments of those occurrences in the book's picture. Tell me which actor represented that character role".

Of course, LLMs could do that and a bunch of other amazing stuff by their "massive brute force" approach that makes them seem "intelligent".

However, what if we ease things for machines a little? Reifying addresses and links as resources on their own, contextually annotable, addressable and linkable, with HTTP / REST means of interaction for their browsing and (link) discovery, having developed a schema on which render the representations of those resources. That's a task in which LLMs could excel. Kind of "meta" AI task, call it "semantic indexing".

Having this "Semantic Hypermedia Addressing" knowledge layer rendered, in RDF resources for example, it could be consumed further by LLMs Agents, given a well defined RAG or MCP tools interface, leveraging the augmented knowledge layer from the previous step. That if you're stuck with AI and LLMs "middleware" (think is better term than "browser" or "client"). Nothing prevents from having this knowledge layer used as a service on its own, with the appropriate APIs.

The rest, use cases and applications, boils down to whatever is possibly imaginable. Each tool bearer ("hammer") will use it to solve every problem ("nail"). Think of "what applications can be done with graph databases". Nearly every tool (programming language) can be used to solve any problem or a part of it (layer)

The question is choosing the right tool for the right layer of the problem. At a networking level, OSI defines seven layers: Application (Protocol), Presentation, Session, Transport, Network, Data Link, and Physical layers. That clean separation allowed us to have browsers, email clients and the internet we know today. MVC pattern and also the Semantic Web itself have a layered pattern layout. Once we know the right layers may we come with the right tools (that's why I said "middleware").

Implementation

RDF / FCA (Formal Concept Analysis) for inference in an Aggregation layer, an FCA-based embeddings model for an Alignment layer and DDD (Domain Driven Development) / DOM (Dynamic Object Model) / DCI (Data, Context and Interaction) and Actor / Role Pattern for the above mentioned Activation layer.

References:

[FCA]:

[DDD]:

[DOM]:

[DCI]:

[Actor / Role Pattern]:

Use Case Example

Federated Supply Chain

Demonstrate end-to-end integration across independent participants.

Participants

Define manufacturer, consumer, and provider systems with app service instances.

Order Flow

Trigger replenishment from low inventory and place orders via MCP interactions.

Procurement

Initiate material acquisition through internal interactions and supplier MCP endpoints.

Federated BI / Analytics

Share anonymized measures to evaluate end-to-end efficiency with dimensional models.

This appendix depicts a complete, multi-organization use case, demonstrating how three independent entities can form a seamless, automated, and intelligent supply chain.

C.1. The Participants & Their Systems

- **Manufacturer:** SportProducts Manufacturing Inc. (SPM)
 - **Systems:** SCM for production, ERP for orders.
 - **AppService Instance:** spm.appservice.com
- **Consumer:** Sport and Fitness Stores (SFS)
 - **Systems:** Legacy ERP for inventory, modern CRM for sales.
 - **AppService Instance:** sfs.appservice.com
- **Provider:** Sports Goods Raw Materials LLC (SGRM)
 - **Systems:** Simple order management database.
 - **AppService Instance:** sgrm.appservice.com

C.2. The Use Case: From Low Stock to Federated BI

Step 1: Low Inventory Trigger at the Retailer (SFS)

- **Services Layout & Roles:**
 - SFS.Datasource: Continuously ingests inventory levels from SFS's ERP via its JCA adapter.
 - SFS.Alignment: Aligns the raw stock number into a canonical Measure. It has previously aligned the "Pro-Lite Running Shoe" from the ERP with SPM's official product definition, creating an owl:sameAs link between did:sfs:product_789 and did:spm:product_ProLite.
 - SFS.Activation: An internal Context named MonitorInventory is constantly running.
- **Messages & Dataflow:**
 1. SFS.Datasource produces a raw statement: ("store_boston", "stock_PLRS", "49").
 2. SFS.Aggregation/Alignment processes this into a Graph Model statement.
 3. The SFS.Activation engine's MonitorInventory Interaction evaluates a rule. The stock level is below the threshold, so it starts a new ReplenishStock Interaction.

Step 2: Automated Order Placement via MCP (SFS -> SPM)

- **Services Layout & Roles:**
 - SFS.Activation: The ReplenishStock Interaction needs to place an order. It knows the canonical product is from SPM. It now acts as an **MCP Client**.

 SPM.Activation: Exposes its capabilities as an **MCP Server**.

- **Messages & Dataflow:**

1. The SFS AppService authenticates to the SPM AppService using **DID-Auth**.
2. SFS queries SPM's MCP endpoint for available Tools related to `did:spm:product_ProLite`.
3. SPM's MCP server responds, offering a `PurchaseOrder` Tool.
4. SFS sends a standardized `ToolCall` message via MCP to SPM's endpoint, containing the required quantity.
5. SPM.Activation receives this, starts a `FulfillOrder` Interaction, and begins a COST/HAL conversation back with the SFS agent to confirm pricing and delivery dates.

Step 3: Manufacturer Checks Raw Materials (SPM)

- **Services Layout & Roles:**

 SPM.Activation: The `FulfillOrder` Interaction's Dataflow includes a Transform to check internal stock for raw materials (`did:sgrm:material_eva_foam`).

 SPM.Datasource: Provides access to SPM's SCM system data.

- **Messages & Dataflow:**

1. It queries its own Graph Model and finds the stock of "EVA Foam" is insufficient.
2. This triggers a new internal Interaction: `ProcureMaterials`. This Interaction identifies the supplier for `did:sgrm:material_eva_foam` as SGRM. SPM's AppService now prepares to act as an MCP Client.

Step 4: Manufacturer Orders Raw Materials via MCP (SPM -> SGRM)

- The process from Step 2 repeats, with SPM acting as the MCP Client and SGRM as the MCP Server.

1 C.3. Business Intelligence Leverage

- **Internal BI:**

 **SFS:** Can analyze its sales data, sliced by store, product, and time, to optimize marketing. They can create a KPI for "Sell-Through Rate".

 **SPM:** Can analyze production data. By correlating `FulfillOrder` Interaction times with the `ProcureMaterials` Interaction times, they can create an indicator for "Production Delay due to Material Shortage."

- **Federated BI:**

 Because all entities use a common (though decentralized) identity system (DIDs) and a common dimensional model, they can agree to share specific, anonymized `ContextMeasure` data.

 They can build a federated view of the entire supply chain's health, analyzing end-to-end efficiency from raw material order to final consumer sale, without exposing sensitive internal operational data.

Miscellaneous Features

FRONTEND:

COST / HAL WebUI

WebUI (COST / HAL Generic Forms) Frontend. Producer WebUI: implement a reactive functional dynamic forms COST / HAL frontend in reactive Angular.

ADMINISTRATION / CONFIGURATION:

Reified Components / Services configuration data as Application Models Instances. Generate an Administration Services configuration schema (bootstrap) in the model, editable via COST (Producer WebUI client) allowing to edit administration schema information from which Services / Components instances retrieve their configurations. Example schema (for Datasource Service: (datasource: (datasourceType, datasourceHost, datasourceUser, datasourcePass))). Implement this configuration mechanism for Services and Components as standard Spring Boot configuration methods / annotations. Service configuration should be treated as another case of backend integration whose configured datasource resides in an internal configuration persistence database and the activation interactions are performed at Service / Component configuration time setting parameter values as the interaction results values. Activation Context: Service configuration schema (roles: keys). Activation Interaction: Service configuration instance (actors: values). Provide a sample configuration dataset from which infer services configuration schema (initial integration Datasource example). Configure multiple types of datasources. Datasource writeBack example for each datasource type.

TOPIC MAPS REFERENCE MODEL (RDF/XML / XSLT)

XSLT DRIVEN ACTIVATION TRANSFORMS

SEMANTIC OBJECT MAPPING

HOMOICONIC APPROACH (DATA AS CODE / CODE FROM DATA)

Data, Information, Knowledge Model levels.

URN as a Resource (ID). Resource events examples: onCreation(Resource created), onOccurrence(Resource context), onAssociation(Resource association). TODO: define pipeline steps stream given components composition functions. Each pipeline Resource event triggers the main event loop pipeline stream from the beginning. Example: Naming (Aggregation) steps, Registry (Alignment) consumes and augments Aggregation Resources with its knowledge which in turn are published / consumed again for further Aggregation (type inference / embeddings / RCV schemas, for example). Idem for Index (Activation).

ID step augmented Resource can shield an Occurrence Resource event (onOccurrence). Occurrence step augmented Resource can shield an Statement Resource (onAssociation). Statement step augmented Resource can shield a Graph Resource

(Flux<Statement>? Serialized Graph Representation?). Stream pipeline steps Functional Streams Composition Functions overloaded for each type of Resource: Resources Activation Content Type dispatch.

Functional Reactive Stream Pipeline Components:

Naming: URN Crafting / Matching. Aggregation. Functional Streams Composition Functions (Function<Resource, Resource> Resources).

Registry: Resource Repository. URNs Resolution / CRUD / Alignment. Functional Streams Composition Functions (Function<Resource, Resource> Resources).

Index: Resource Contents URNs Resolution (inferences, transforms). Activation. Functional Streams Composition Functions (Function<Resource, Resource> Resources).

Main Event Loop: Topic. Resources publishing / subscriptions. Functional Streams Composition Functions. onResource(Resource) : Resource. Naming, Registry, Index stream steps pipeline.

Reactive Resources: ID, IDOccurrence, Statement, Graph, Step, Messages, etc. Content Type Addressing: Graph Statements by patterns / CPPE / RCV schema instances / roles / Kinds / SPARQL. Statement occurrences by position / role (Kind). IDOccurrences by contexts (schema / Kinds). IDs by occurrences (role / context schemas). Content Types: graph/set, graph/activation, graph/reference, statement/activation, occurrence/subject, producer/form (COST state exchange), etc. Super type / sub type functional transforms (Resources Function<Consumes, Produces>): addressing / query / traversal / augmentation steps.

Resources: Data (XML), Transforms (XSLT). Addressable / Registered. Resolvable by Content Type / URNs pattern matching (Resource Messages Endpoint). Content Types: Inferred Schema Resources (RDFS). Upper Alignment. To / from underlying representation Transforms. Activation Verbs inference (Activation) inferred Transform Resources.

Pipelines: Step (Resource), Resource Messages Endpoint (Shared Messages Resource stream topic, Resources publish / subscribe. Obtain Resource representation in Content Type by URNs / pattern matching, retrieve / invoke activated Resource verbs), Shared State / Tools. Reactive Resources: ID, IDOccurrence, Statement, Graph, Step, Messages, etc. Underlying representation, Content Type renderers (Data out) / activation (Transforms out) registered handlers. Augmentation Aggregation, Alignment, Activation pipeline Resources.

Semantic Discoverable URNs (embed context / content). Reactive Resources subscribes / publishes to Messages Endpoint, Content Type / URNs pattern matching dispatches to corresponding event handlers resolving corresponding Transforms Content Types / URNs

with this Resource as payload for each event. For updating FCA metadata / type, Kinds inference for example.

Transforms Resources applied over Content Type / URN matching Resources (Messages) payloads, publishes transformed results back to the Messages Resource.

Functional Composition: Main event loop. Pipeline Datasource Resource inputs, Messages Endpoint dispatch, Producer Resource outputs.

Custom Resources (IO Monad):

Datasources Resource Instances. Configured declaratively in the model. Produces / Consumes data URNs Statements (provenance / sync).

Producer Resource Instance. Produces APIs / UI Representation Content Types. Consumes user interaction forms state.

Wrap LLM / MCP into a Resource. Handle Datasources sync / Producer Activation COST.