# 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):
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 (Subject 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.

Statements:
(Context, Attribute, Value)
**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<Attribute Type, ValueType>[]
- occurrences: Occurrence Type[]

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: Resource Occurrence 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[]>
    - CPPEembedding: 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

*(Image description: A Venn diagram showing the intersection of Subject, Predicate, and Object sets to form SubjectKind, PredicateKind, ObjectKind, and StatementKind.)*

### 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.

### 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.
**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. CPPE(Google, worksFor) = primeID(Alice) * primeID(Bob) * ...
- **Similarity Calculation & Inference:**
  - **Similarity:** The similarity between two entities in the same context is the Greatest Common Divisor (GCD) of their CPPEs. GCD(CPPE(Google), 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 GCD(CPPE_brotherOf(A), CPPE_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 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:** RCV(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 = calculateCPPE(statement.subject, 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 = calculateCPPE(statement.predicate, 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 = calculateCPPE(statement.object, 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 rdf:type schemaURI.
        2. For each instance s_i, retrieve its cached RCV_i = (S_i, P_i, O_i).
        3. Calculate the schema RCV components:
            - S_schema = LCM(S_1, S_2, ..., S_n)
            - P_schema = LCM(P_1, P_2, ..., P_n)
            - O_schema = LCM(O_1, O_2, ..., O_n)
        4. The result (S_schema, P_schema, O_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
  compose(RCV1, RCV2)
  - **Inferred Subject (S_inferred):** S_inferred = RCV1.s * RCV2.o
  - **Inferred Object (O_inferred):** O_inferred = RCV1.s * RCV2.o
  - **Inferred Predicate (P_inferred):** P_inferred = RCV1.p * RCV2.p
- **Step 2: Calculate Schema Archetypes**
  - The Alignment Service calculates archetypal RCVs for source schemas:
    - 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):
    - 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.
- Step 3: The Inference Algorithm at Query Time
  A user asks: "Does dev:Alice know lang:Java?"
  1. **Retrieve Instance RCVs:** Retrieve RCV1 for (dev:Alice, :worksOn, proj:Orion) and RCV2 for (proj:Orion, :usesLanguage, lang:Java).
  2. **Calculate Hypothetical Instance RCV:** RCV_hypothetical = compose(RCV1, RCV2).
  3. **Retrieve Schema Archetype:** Retrieve RCV_knowsLang_schema.
  4. **Perform Numerical Check:** boolean knows = isInstanceOf(RCV_hypothetical, RCV_knowsLang_schema).
  5. **Result:** If knows is true, the inference is validated.

## Querying and Traversal by Numerical Properties:

- **Find by Relational Role:** "Find all entities that have acted as a :Developer".
  - The query becomes: "Find all statements whose instanceRCV.s component divides RCV_dev_schema.s."
- **Traversal by Numerical Similarity:**
  - Start at stmt_A with RCV_A.
  - The next step: "Find stmt_B whose RCV_B has the highest GCD with RCV_A."
  - Allows traversal based on numerically similar relational contexts.

# Appendix E: Numerical Representation and Inference

## of Relational Schemas

*(This section re-iterates the details from "Algebraic Embeddings" through "Querying and Traversal" in a formal appendix format.)*

### E.1. The Relational Context Vector (RCV)

...

### E.2. Numerical Representation of Schema vs. Instance

...

### E.3. Inference via Mathematical Operators

...

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

...

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

...

### E.4. Querying and Traversal by Numerical Properties

...

# Functional Resources Approach

## Resource Monad API

- **The Resource Monad:** Functional wrapper, Resource<ResourceOccurrence>.
- 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) Activation

- ResourceOccurrence::onOccurrence(...)
- ResourceOccurrence::getOccurrences(...)
- ResourceOccurrence::getOccurringContexts(...)

#### Functional Output Model Building (Representation folding):

- ID::getOccurrences(...) -> IDOccurrence
- ...
- Representation::getOccurrences(...) -> 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.

### 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 and publish augmented Model Events.

## Functional Retrieval / Traversal Operations

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

## Event Streams

### Aggregation

- Produce SPO and kinds aggregated statements with DIDs, prime IDs, and FCA clustering.
- **Inference:** type / state / order. FCA Model.
- **Hierarchies:** Type / State hierarchies.
- Consumes (ID, ID, ID) Statements; produces SPO / Kinds Aggregated Statements.

### Alignment

- Infer equivalences, link predictions, and kind hierarchies for models and instances.
- **Model:** DOM (OGM) Model.
- Consumes SPO / Kinds Aggregated Statements; produces Graph / Models Statements.
- **Functionality:** Ontology matching, Link Prediction, Clustering, Classification.

### Activation

- Expose contexts, interactions, actors, and roles via the activation API.
- **Model:** DCI (Actor / Role) Model.
- Consumes Graph / Models Statements; produces ContentType Representation Statements.
- Provides Contexts, Interactions, Actors, Roles State API.

## Nodes Functional Reactive Behavior

- Consume Augmented Model (Representation?)
- Functional Input Model Traversal (Representation unfolding)
- Functional Output Model Building (Representation folding)
- Publish Augmented Model (Representation?)

## Events Stream and Augmentation Services

- **Messages:** Models (Representations)
- **Events:** Model (Representation?) Messages.
- **Topic Event loop / Registry:** Blackboard design pattern.
- **Datasource node:** Produces and listens for Model Events for syncing backends.
- **Producer node:** Consumes Model Events, publishes Activation API, produces API interaction Events.

## Core Services

### Model Service (Content Types)

Main RDF store persistence handler. Handles persistence of Reference Model Statements and conversion between model views.

### Application Service

Encloses Datasource, Augmentation, Producer Service interactions.

### Augmentation Service

Encloses Aggregation, Alignment, Activation Service Interactions.

*All services should have an administration / management interface for each step of the workflow.*

### Datasource Service

Produce and consume raw integrated datasource triples. Keep source backends consistent.

### Aggregation Service

Infer domain schemas, states, and data. Perform FCA / Kinds Augmentation, CPPE / RCVs handling.

### Alignment Service

Infer Relationships, Events and Transforms. Equivalent entities ontology matching. Missing links prediction.

### Activation Service

Expose use-case contexts and interactions. Instantiates inferred use cases (Contexts) executions (Interactions).

### Producer Service

Exposes a Context / Interaction aware API for browsing / initiating application transactions. Handles dynamic transactions flows (wizard-like interface).

## Helper Services

### MCP Service

MCP Server and Client features. Enable Application Service as an MCP Client.

### Index Service

Functionality related to inference, query, retrieval and traversal of models. Cache RCVs.

### Naming Service

Functionality related to name resolution and inference, such as labeling Kinds and Relationships. LLM MCP bridge.

### Registry Service

Main ResourceOccurrence(s) repository. IDs generation. Core graph model repository.

## Semantic Hypermedia Addressing

Imagine the possibility of not only annotating resources (Text, Images, Audio, Video, Tabular, Hierarchical, Graph) with metadata and links but having those annotations and links being generated by inference and activation. RESTful principles could apply rendering annotations and links as resources also, making them discoverable and browsable / query-able.

This "Semantic Hypermedia Addressing" knowledge layer, rendered in RDF, could be consumed further by LLMs Agents. User-generated resources and business application interactions would leverage this semantic addressing, becoming part of a resource-oriented linked knowledge network.

## Implementation Details

- **Technologies:** RDF / FCA (Formal Concept Analysis) for inference, an FCA-based embeddings model, and DDD (Domain Driven Development) / DOM (Dynamic Object Model) / DCI (Data, Context and Interaction) and Actor / Role Pattern.
- **References:**
  - [FCA]
  - [DDD]
  - [DOM]
  - [DCI]
  - [Actor / Role Pattern]

## Use Case Example: Federated Supply Chain

Demonstrate end-to-end integration across independent participants.

### Participants

- **Manufacturer:** SportProducts Manufacturing Inc. (SPM)
- **Consumer:** Sport and Fitness Stores (SFS)
- **Provider:** Sports Goods Raw Materials LLC (SGRM)

### Order Flow

From low inventory trigger at the retailer (SFS) to automated order placement via MCP to the manufacturer (SPM).

### Procurement

Manufacturer (SPM) checks for raw materials and places orders with the provider (SGRM) via MCP.

### Federated BI / Analytics

Participants share anonymized measures to evaluate end-to-end efficiency with dimensional models, building a federated view of the supply chain's health.

## Miscellaneous Features

### FRONTEND

- **COST / HAL WebUI:** Implement a reactive functional dynamic forms COST / HAL frontend in reactive Angular.

### ADMINISTRATION / CONFIGURATION

- Reified Components / Services configuration data as Application Models Instances.
- Editable via COST (Producer WebUI client).

### TECHNICAL CONCEPTS

- 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.

### Functional Reactive Stream Pipeline Components

- **Naming:** URN Crafting / Matching.
- **Registry:** Resource Repository.
- **Index:** Resource Contents URNs Resolution.
- **Main Event Loop:** Topic for resource publishing/subscriptions.

### Custom Resources (IO Monad)

- **Datasources Resource Instances:** Configured declaratively.
- **Producer Resource Instance:** Produces APIs / UI.
- Wrap LLM / MCP into a Resource.