Application Integration

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.

Strategic Plan Overview

This document contains strategic goals and objectives without quantitative performance data.

Vision

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.

Mission

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.

Goal 1:

Integration Architecture

Establish layered representations and activation to connect heterogeneous sources.

Objective 1.0:

Service Oriented Layout

Core Services:

- Datasources
- Aggregation
- Alignment
- Activation
- Producer

Helper Services:

- Index Service
- Naming Service
- Registry Service

Core Streaming Layout:

- Single Topic Architecture

- Blackboard design pattern

Objective 1.1:

Unified Datasources Service

Produce and consume raw integrated datasource triples. Keep source backends consistent with interaction outcomes across integrated applications.

Objective 1.2:

Aggregation Service

Infer domain schemas, states, and data from tabular, XML, JSON, and graph input triples from Datasources Service. Perform FCA / Kinds Augmentation.

Objective 1.3:

Alignment Service

Infer Relationships, Events and Transforms and align aggregated information into this (inferred upper) ontologies. Equivalent entities ontology matching. Missing links prediction.

Objective 1.4:

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).

Objective 1.5:

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 Convertsational State Transfer: COST implemented).

Objective 1.6:

Helper Services

Services that are orthogonal to other stream based services.

Index Service:
Functionality related to inference, query, retrieval and traversal of models, as for example the FCA model (see Goal 5).

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. Leveraged by Index and Naming Services.

Objective 1.7:

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 its criteria (ContentType for example).

Goal 2:

Functional Reactive Approach

Wrap resource occurrences to unify events, getters, and contexts across content types.

Objective 2.1:

Resource Monad API

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

Objective 2.2:

ResourceOccurrence Hierarchy

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)

Objective 2.3:

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

Objective 2.3:

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

Goal 3:

SPO Kinds

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

Objective 3.1:

Identifiers

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

Objective 3.2:

Statements

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

Objective 3.3:

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.

Objective 3.4:

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.

Objective 3.5:

Instance Aggregation

Aggregate kind occurrences into graphs and models for traversal and matching.

Objective 3.6:

Dimensional Alignment

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

Dimensional Upper Model Kinds. Relationships / Events inference.

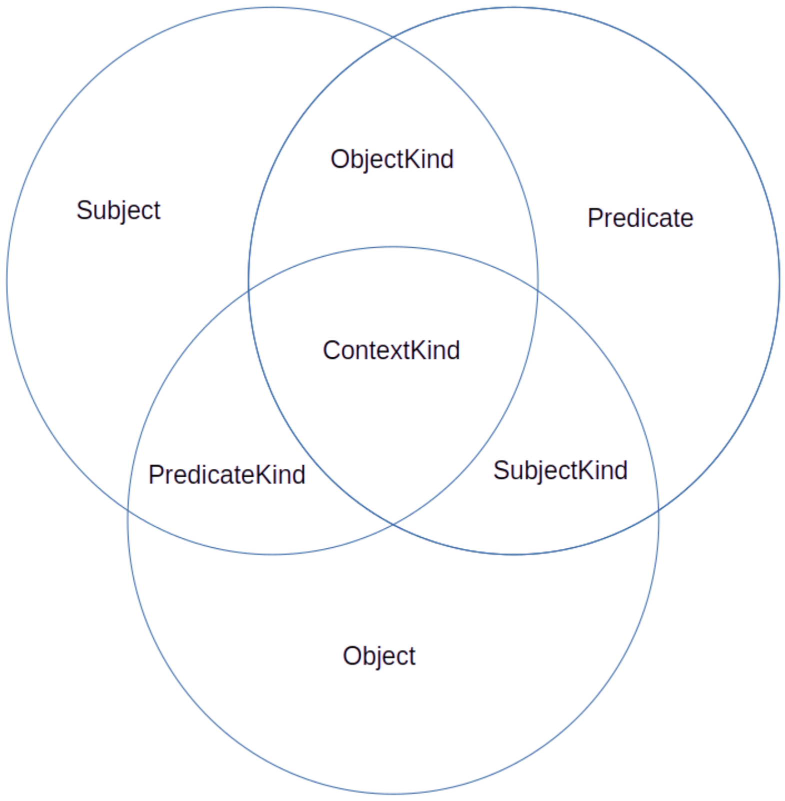
Data:
Measures. Players.

Information:
Dimensions: Measures in Context. Roles.

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

Relationships / Events order / closures.

Sets Layout



Goal 4:

Model Abstractions

Define Models (schemas) declaratively enhancing ontology alignments and contextual inference.

Objective 4.1:

Relationships and Events upper Schema (Kinds) Alignment Model:

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

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

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

Kinds: Upper alignment concepts. Kinds Alignment.

Statements: Upper schemas, aligned Kinds and Instance occurrences.

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.

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

ContentType / Representation (Model Graphs Statements).

Objective 4.6:
Relationships and Alignment
(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;

Objective 4.2:

FCA Model

(Context, Object, Attribute);
(expand: positions. Attributes: align / match).

Relationships / Events:
(Relationship, Role, Player);
(Role, EventTransform, Role);
(expand: positions. Attributes: align / match)

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);

Objective 4.3:

Dimensional Alignment Model:

Dimensional (base upper ontology?):
Data Statements.
Information Statements.
Knowledge Statements.

Objective 4.4:

DOM (Dynamic Object Model) Alignment Model:

DOM:
Type / Instance Statements.

Objective 4.5:

DCI (Data, Context and Interaction) Alignment Model:

DCI:
Context / Interaction Statements.
Actor / Role Statements.

Goal 5:

Numerical Inference

Apply FCA-based reasoning, CPPE embeddings, and relational context vectors for schema and instance derivation.

Objective 5.1:

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 .
```

Objective 5.2:

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**: 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.

Objective 5.3:

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.

Objective 5.4:

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.

Objective 5.5:

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.

Objective 5.6:

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.

Goal 6:

Event Streams

Operate model events and augmentation services via a topic-based loop.

Objective 6.1:

Aggregation

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

type / state / order inference. FCA Model.

Consumes (ID, ID, ID) Statements; (and hierarchy).
Produces SPO / Kinds Aggregated Statements.

DIDs, Prime IDs, FCA Clustering.

Objective 6.2:

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: preparated 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.

Objective 6.3:

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(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?)

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.

Goal 7:

Helper Services

Support naming, registry, and indexing across dynamic
kinds and content types.

Objective 7.1:

Naming

Embed instance type and context in URNs and content type signatures.

Objective 7.2:

Registry

Track models, transforms, and dynamic kind content types.

Objective 7.3:

Indexing

Cache RCVs and related signatures for fast inference and traversal.

Goal 8:

Federated Supply Chain

Demonstrate end-to-end integration across independent participants.

Objective 8.1:

Participants

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

Objective 8.2:

Order Flow

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

Objective 8.3:

Procurement

Initiate material acquisition through internal interactions and supplier MCP endpoints.

Objective 8.4:

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.

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.