Application Integration

1. Introduction

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.

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.

2. Implementation Details

Approach description

2.1. The Resource Monad

Functional wrapper

Resource Monad: Resource<ResourceOccurrence>.

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

2.2. Core Classes

Resource Monad bound objects.

ResourceOccurrence
- representation : Representation
- onOccurrence(ResourceOccurrence occurrence) : ResourceOccurrence context (dispatch to Representation ContentType)
- getOccurrences(S, P, O) (dispatch to Representation ContentType. S, P, O criteria)
- getOccurringContexts(S, P, O) (dispatch to Representation ContentType. S, P, O criteria)
- getAttributes() : String (by means of occurrences / schema)
- getAttribute(String) : String
- setAttribute(String, String)

ResourceOccurrence Class Hierarchy:

ID extends ResourceOccurrence
- primeID : long
- urn : string
- occurrences : Map<Kind, IDOccurrence[]>
- CPPEembedding : long

IDOccurrence extends ID
- occurringId : ID
- occurringContext : ID
- occurringKind : Kind

ID occurrences by Kind (ContentType part).
IDOccurrence occurringKind (ContentType part).

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 (Transforms implemented in XSLT / Custom Functional Logic / Helper Services) extends Model
- kind : Kind
- typeSignature : String ([modelName]/[occurrenceType]/[kindName]/[encoding])
- onOccurrence(ResourceOccurrence) : ResourceOccurrence (transform)
- getOccurrences(S, P, O) : ResourceOccurrence[] (transform. S, P, O criteria)
- getOccurringContexts(S, P, O) : ResourceOccurrence[] (transform. S, P, O criteria)
- fromRepresentation(Representation) : Representation
- toRepresentation(ContentType) : Representation

Representation extends ContentType
- contentType : ContentType
- encodedState : String (Encoding Types)

2.2.1. 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

2.2.2. FCA

2.2.3. Semantic Addressing

type:id:context. DIDs.

2.2.4. Statements

SPOs / Kinds
Data
Kinds
Schema

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

2.2.5. Models

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)

2.3. FCA CPPE / RCVs inference

FCA SPO Contexts.
Prime IDs.
Concept Lattices (Types)
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 .

Materializing Relational Instances

Once a schema is inferred, the system can find and explicitly link all data instances that conform to it.

- **Algorithm: Materializing Instances**:
    1. **Iterate Schemas**: The Alignment Service iterates through the newly inferred :RelationalSchema classes.
    2. **Construct Query**: For each schema (e.g., :DeveloperWorksOnProject), it constructs a SPARQL query to find all raw statements that fit the schema's domain and range.
    SELECT ?statement WHERE {

```
    ?statement a rdf:Statement ;
            rdf:subject ?s ;
            rdf:object ?o .
    ?s a :Developer .
    ?o a :Project .
}
```

3. Link Instance to Schema: For each matching ?statement, it materializes a new triple, explicitly typing the statement as an instance of the inferred schema.
   <:stmt_123> rdf:type :DeveloperWorksOnProject .
   This makes future queries for "all instances of developers working on projects" much faster, as it becomes a simple rdf:type lookup.

Inferring and Materializing Order Relations

Order is inferred from type/state hierarchies and then materialized as explicit links.

- **Schema Inference**:
  - **Method**: The Alignment Service analyzes state transition data. It observes that instances of :Order consistently transition from a state of :Placed to :Paid to :Shipped.
  - **Materialization**: It creates a directed graph of the state schema using a custom property:
    :Paid :isPrecededBy :Placed .
    :Shipped :isPrecededBy :Paid .

- **Instance Materialization**:
  - **Method**: The service listens for events. When it sees an OrderPaid event for order_789 at T1 and later an OrderShipped event for the same order at T2, it can materialize the instance of the order relation.
  - Materialization: It creates a direct link between the reified event statements:
    <:event_ship_789> :isPrecededBy <:event_paid_789> .

Inferring Domain-Specific & Transitive Relations

This is where the system learns complex, multi-hop rules from the data.

- **Attribute Closure (e.g., knowsLanguage)**:
  - **Schema Inference Algorithm**:
    1. Run a SPARQL query to find recurring paths of length two: SELECT ?r1 ?r2 WHERE { ?a ?r1 ?b . ?b ?r2 ?c . }.

2. Aggregate the results. A frequent co-occurrence of (:Developer)-[:worksOn]->(:Project) and (:Project)-[:usesLanguage]->(:Language) suggests a new potential relation.
3. Materialize the new relational schema: :knowsLanguage owl:propertyChainAxiom (:worksOn :usesLanguage) . This uses OWL 2's powerful property chain axiom to formally define the rule.

- **Instance Materialization**: Once the rule is defined, the Jena reasoner can automatically infer the (:Developer)-[:knowsLanguage]->(:Language) triples without needing to materialize them explicitly. Queries for :knowsLanguage will return results as if the triples existed.

- **Link Inference (e.g., uncleOf)**:
  - **Schema Inference**: The process is identical to attribute closure. The system detects the frequent path (:Person)-[:brotherOf]->(:Person)-[:fatherOf]->(:Person) and defines the uncleOf property chain.
  - **Instance Materialization**: Again, the OWL reasoner handles the inference.
- **Transitivity (e.g., partOf)**:
  - **Property Inference Algorithm**:
    1. The system queries for instances of the pattern (?a :partOf ?b) , (?b :partOf ?c).
    2. For each result, it checks if the triple (?a :partOf ?c) also exists.
    3. If this holds true for a high percentage of cases, the system can infer that the :partOf relation is likely transitive.
  - **Schema Materialization**: It formally declares the property as transitive: :partOf rdf:type owl:TransitiveProperty .
  - **Instance Inference (Transitive Closure)**: The OWL reasoner will automatically handle the transitive closure. A query for all parts of C will correctly return A even if only the (A, B) and (B, C) links are explicit.

Advanced Calculations and Further Inferences

- **Symmetry/Reflexivity**: The same statistical analysis used for transitivity can be used to infer if a property is symmetric (e.g., :spouseOf) or reflexive. These are also declared using OWL (owl:SymmetricProperty, owl:ReflexiveProperty).
- **Analogous Relationships via CPPE**: The GCD of Contextual Prime Product Embeddings can reveal more than just direct similarity. If GCD(CPPE(A, :manages), CPPE(B, :mentors)) is high, it suggests that A and B have similar

teams, implying an analogous relationship between "managing" and "mentoring" in this domain. This can be flagged for a human expert to review and potentially create a new rdfs:subPropertyOf axiom.
- **Query Acceleration**: All materialized relationships (both schema and instance) act as "shortcuts" in the graph. A complex, multi-hop query for "uncles" becomes a single-hop query for rdf:type :UncleOfRelation, which is dramatically faster and less computationally expensive, directly benefiting the Activation and BI layers.

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.

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.

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)**
        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.

Inference via Mathematical Operators

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

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.

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_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.

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.

2.4. SPO / Kinds based inference

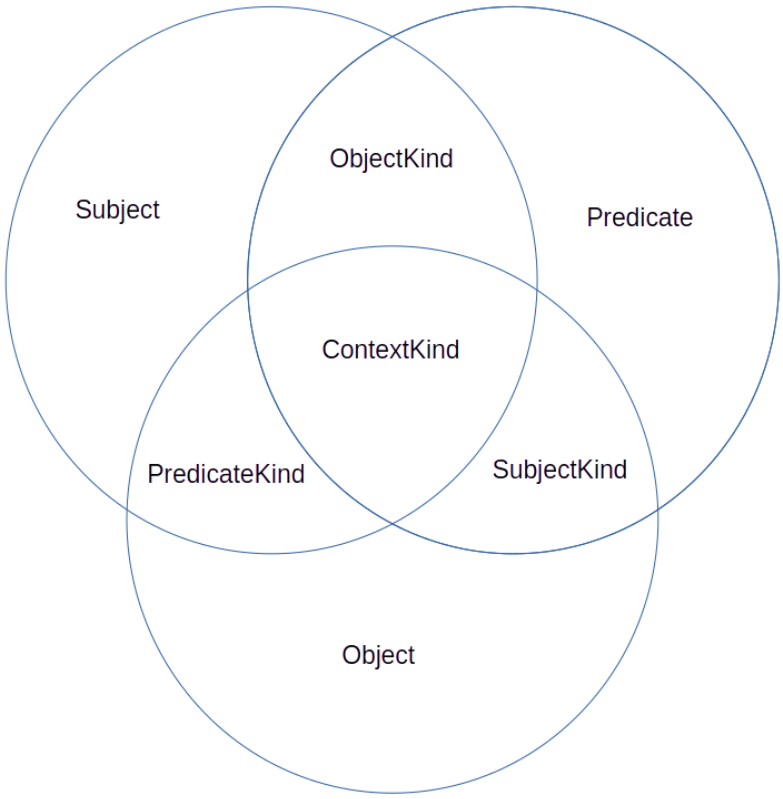
This model elevates the Reference Model by reifying statements into higher-order Kinds.

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.

Implementation: A Kind is a rdfs:Class that is a rdfs:subClassOf its base type (e.g., <:Customer> rdfs:subClassOf <:SubjectKind>).

Inference & Traversal (Functional Interfaces):
Capability: "Given the :Customer type, what types of actions can they perform?"

Implementation (Function<URI, Set<URI>>): A SPARQL query that finds all PredicateKinds connected to the given SubjectKind in schema statements.



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

2.4.1. 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

2.4.2. Dimensional Data / Information / Knowledge Alignment

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.

TODO

3. 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.

3.1. 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?)

3.1.1. Aggregation Node

type / state / order inference. FCA Model.

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

DIDs, Prime IDs, FCA Clustering.

3.1.2. Alignment Node

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.

3.1.3. Activation Node

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.

3.2. Helper Services

3.2.1. Naming Service

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

3.2.2. Registry Service

3.2.3. Index Service

Appendix C: End-to-End Integration Use Case: A Federated Supply Chain

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.