1 Introduction

1.1 Declarative application metamodels.

The main idea is to be able to merge diverse datasources (from existing applications databases for example) and from they and their metadata expose 'declarative' application models which can be used for domain driven front ends or services.

This is to be accomplished through the use of a hierarchical 'reference model’ for all types of resources which, by the means of Big Data and learning capabilities, is enabled to provide inference and further 'Features’ over these representations.

The 'representations’ mentioned (discussed in section 2.2) are nothing but just the maximum possible de-aggregation (or reification) of resources and statements from RDFized inputs, being they become enabled to have an uniform 'Resource reference model’ and allowing for composition into layers, Models and Containers. This pattern intention is to allow for composition of layers and, ultimately, to be able to compose bindings for Adapters and Ports which are the Runtime’s final input and output bindings.

2 Architecture

2.1 Resources: Dataflow architecture

Each ‘container’ has (at least) three models: Data, Grammars and Services models. Each model is implemented with the same structure (classes) and has the same functional invocation model (Rules aggregation to Events). Models has three layers, for Data models those are: Facts, Topics and Purpose.

Layers Resource flow: Resources (Statements) ‘flows’ through model’s layers because each model statements (and thus resources) patterns (class quads) are defined in terms of previous ones.

Model Profile and Messages: A message matching some model’s profile (via Grammars) is ‘consumed’ by the receiving model applying the message statements to itself and returning whatever statements results from querying itself again with the message contents.

Model Profile: (Model, DataModelFacade, GrammarModelFacade, ServicesModelFacade) : Facade’s Statements (Rules / Events).

Wiring / Discovery (Containers). Adapters / Ports Dataflow.

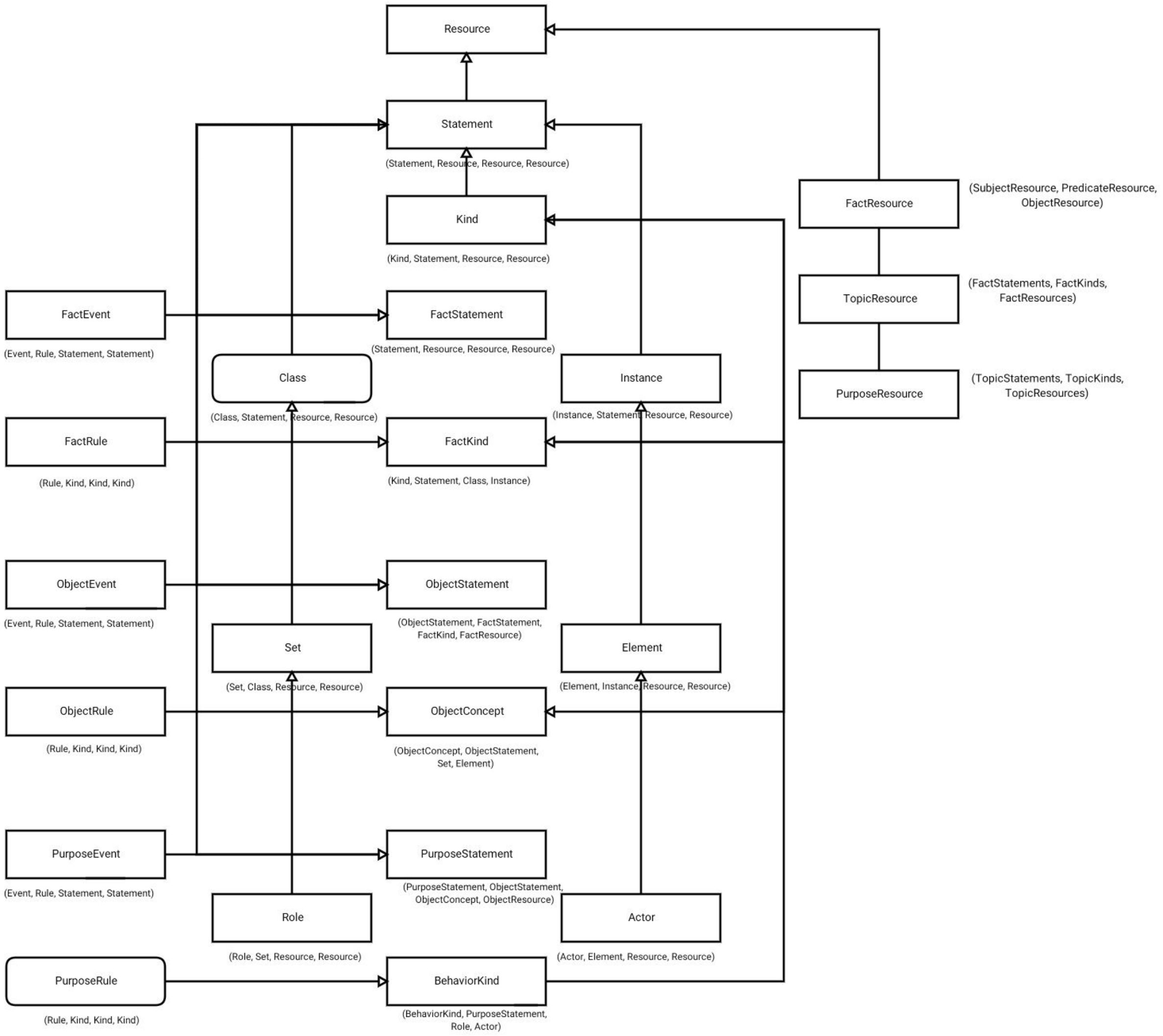
State (Actor) / Interface (Role) / Context declarations (Facts, Topics, Behavior statements).

Interfaces: Rules from each model (Data, Grammars, Services) layers aggregate knowledge from which obtain Events related to a ‘functional invocation’. The same are of the form:

ModelLayer(ctx : Kind, res : Resource[, rel : Resource])

So, each model layer has a rules aggregate from which to obtain events, being rules the kind of an occurrence of something (‘kind’ of statement) and the events holds information regarding rules instances.

2.2 Core Models and Classes


Resource
Statement
(Statement, Resource, Resource, Resource)
FactResource
(SubjectResource, PredicateResource, ObjectResource)
Kind
(Kind, Statement, Resource, Resource)
TopicResource
(FactStatements, FactKinds, FactResources)
FactEvent
(Event, Rule, Statement, Statement)
FactStatement
(Statement, Resource, Resource, Resource)
PurposeResource
(TopicStatements, TopicKinds, TopicResources)
Class
(Class, Statement, Resource, Resource)
Instance
(Instance, Statement, Resource, Resource)
FactRule
(Rule, Kind, Kind, Kind)
FactKind
(Kind, Statement, Class, Instance)
ObjectEvent
(Event, Rule, Statement, Statement)
ObjectStatement
(ObjectStatement, FactStatement, FactKind, FactResource)
Set
(Set, Class, Resource, Resource)
Element
(Element, Instance, Resource, Resource)
ObjectRule
(Rule, Kind, Kind, Kind)
ObjectConcept
(ObjectConcept, ObjectStatement, Set, Element)
PurposeEvent
(Event, Rule, Statement, Statement)
PurposeStatement
(PurposeStatement, ObjectStatement, ObjectConcept, ObjectResource)
Role
(Role, Set, Resource, Resource)
Actor
(Actor, Element, Resource, Resource)
PurposeRule
(Rule, Kind, Kind, Kind)
BehaviorKind
(BehaviorKind, PurposeStatement, Role, Actor)

The diagram above shows classes (named boxes) of Resource's hierarchy, some of their relationships (arrows), some of them representing 'specialization' in 'pseudo' UML and finally below each box is a statement declaration which represent aggregations made for this 'class' instance's RDF quads (patterns).

Core classes syntax: (a ':' means 'extends', names in parentheses express -quad- members of the class)

Statements

A model loads/aggregates resources from statements or from other models.
Statements are the means for obtaining resources (SPOs). In the context of an statement a resource has a 'kind' (type or class). For example, a subject (S) has a 'subject kind' (PO of an

statement). Given any resource occurring in a statement its kind will be the remaining SPO parts of the statement, being one part the ‘attribute’ (class) and the other one the ‘value’ (metaclass).

A model’s layers statement is built from aggregating ‘source’ statements or statements, kinds and resources from previous layers as SPO having each layer its own type of kinds and statements.

Resource addressing scheme (layers / models) IRI scheme.

Rule (example): (Marriage, Person, Single, Married).

Event (example): (aMarriageEvent, Marriage, singlePersonStatements, marriedPersonStatements).

Models

Runtime

Peer

Container (Models: data, grammars, services)

Adapter (src : Model, dest : Model)
Src: JenaModel, RDBMSModel, etc.
Dest: InMemoryModel, etc.

Port (Adapter, Adapter)
Dest Adapter Model: SOAPModel, RESTModel, ODataModel, etc.

Model : (Model, Facts, Objects, Purposes) Aggregates layers statements (Facade).

Container Models

DataModel (Facts, Topics, Purposes)

GrammarModel (FactRules, TopicRules, PurposeRules).
Populated from statements with SPOs from DataModel layers Kinds (S - SK, etc).

ServicesModel (Index/mappings/state, Naming/order/contexts, Registry/behavior/interaction) .
Populated from statements which reify DataModel classes.

Functional Model

Rule : Data(F, T, P) (Context, Resource [, Resource]) : Event[]

Rule : Grammar(FR, TR, PR) (Context, Resource [, Resource]) : Event[]

Rule : Services(I, N, M) (Context, Resource [, Resource]) : Event[]

Adapter / Port Message IO.

Profiles (Actor/Role) : Models (grammar)

Message : Model (Profile, State, Context, Interaction) - Facts, Topics, Purpose Statements.

Templates : Model. Transitions, Flows, Transforms. Data / Grammar / Service layers functional invocation in statements (reactive Model). TemplateResource (variables / wildcards).

Graph Views (of statements, rules, events. Example: relationships browse). Functional reference model.

Aggregated Models

Resource (IRI : Resource, ctx/occur/parent : Resource, attr : Resource, value : Resource).

ModelResource (ModelResource, Model, Kind, Resource).

ModelStatement : Statement (ModelStatement, ModelResource, ModelResource, ModelResource).

Model : ModelStatement (Model, ModelStatement, ModelStatement, ModelStatement).

Model : Statement (Model, FactStatement, TopicStatement, PurposeStatement). Data Model.

Profile : Model (Profile, FactRule, TopicRule, PurposeRule). Grammars Model.

Template : Profile (Template, FactEvent, TopicEvent, PurposeEvent). Services Model.

Services accounts for actual statements (Events) Rule(Data) : Events. And for possible Rules in a given state: Event(Data) : Rules. FactEvents, TopicEvents, PurposeEvents : index, naming, registry services respectively.

Message : Template (Message, Template, Template, Template).

Container : Model (Container, Model, Profile, Template).

Adapter : Model (Adapter, Model, Profile, Model). Profile : criteria.

Port : Model (Port, Adapter, Template, Adapter). Template : transform.

Services: Models Rules / Events.

Populate Facades (Models) aggregating bottom up. Resolve models (messages): Container -> Template -> Profile -> Model.

IOC / Dataflow

Resource : Quad<C, S, P, O>(ctx : Quad, data : List<SQuad<PQuad<OQuad>>>)

QuadClass / AggregatorClass: For each Resource (Quad) factory / registry / events class. IOC setup. Instances (Quads). SPO AggregatorClass(es) (injected instances). Parent Aggregator (injected instance). SPO / parent Agreggator injected by context (Aggregator instances).

Quad / Aggregator factory / CRUD methods (over data). Quad, Aggregator, instances (data by ctx Quad). Events:

Quad.onDataEnter(Aggregator, ctx Quad).
Quad.dataEntered(Aggregator, Quad).
Aggregator.apply(ctx : Quad, data : Quad). Notify SPO Aggregators.
Aggregator.notify(ctx : Quad, data : Quad). Notify parent Aggregator.
Quad.onDataLeave(Aggregator, ctx Quad).
Quad.dataLeaved(Aggregator, Quad).

Aggregator.onResource: Event callback matches / visits Aggregator by Resource / Quad CSPO types.

Aggregator.addResource(Quad data): Adds / updates (apply / notify). Notify all child/parent Aggregators. Child: create if new. Parent / update: resolve same Resource (for same Quad). Emit same Quad ctx to match from src Resource (ie.: add / update returns Statement ID of child / parent. Registry / Map temp values).

2.3 CSPOS, Kinds and Statements

A special type of Resource are 'Kind’s. Each SPO in a Statement has its corresponding Kind which has an ‘attribute’ and a 'value’ in respect to its position in the triple. For example, the triple:

(Peter) (worksAt) (IBM)

Has a 'SubjectKind’ of (worksAt, IBM) for its Subject (Peter). Subject’s attribute and value are (worksAt) and (IBM) respectively. This metadata could be used for basic type inference by aggregating Employees (worksAt domain) and specific employees (which work at IBM).

The same holds for classifying Predicates and Objects into their types and their corresponding meta-types. Different Model layers (discussed later) have its own (Predicate based) SPO Sets definitions and, thus, their own Statement and Kind structures.

Kinds reification may be performed given, for example, this example 'Employee’ SubjectKind becoming a 'Employee’ Subject. This way attributes and links may be stated for the 'Employees’ set in general (Grammars model).

Statement Kinds: Rules / Events.

2.4 Application core Model Layers

The first Model (Facts) is built from raw input triples (SPOs) and its Kinds are aggregated according its Statements and Resource occurrences.

Then, TopicModel’s Statement and SPO instances are Resource Sets built from Predicates which take previous Model Sets (Facts) Statements, Kinds and SPOs for building Model’s SPO respectively. The same pattern is applied for building BehaviorModel layer using TopicModel’s Topics, Kinds and SPOs.

FactModel
SPO: (Subject, Predicate, Object).
Kinds: SubjectKind (PO), PredicateKind (SO), ObjectKind (SP).
Statements: (FactCtxPred, Subject, Predicate, Object).

TopicModel
SPO: (Object: FactModel’s Statements, Concept: FactModel’s Kinds, Sign:FactModel’s SPOs).
Kinds: ObjectKind, ConceptKind, SignKind.
Statements: (TopicCtxPred, Object, Concept, Sign).

BehaviorModel
SPO: (Topic: TopicModel’s Statements, Scenario: TopicModel’s Kinds, Player: TopicModel’s SPOs).
Kinds: TopicKind, ScenarioKind, PlayerKind.
Statements: (BehaviorCtxPred, Topic, Scenario, Player).

Grammars (Facts example)
SPO: (SubjectKind, PredicateKind, ObjectKind) non-terminals.
Kinds: Subject, Predicate, Object (terminals).

Statements: Kinds / SPOs (rules, productions).

2.5 Services: Reactive Models and Containers

Services are a special type of Model held by Containers in their scope (along with Grammars). They have the same structure of other models but they instantiate a reified version of model's classes and instance data.

Given this model 'materialization' layers of this model provides with Rules and Events which take into account 'state' (indexing of rels) of resources, 'context' (order in namespace) of resources and 'hierarchy' (location in interaction) of resources. So, with functional invocation of these, 'use cases' may be declaratively stated and executed.

2.6 Resources reference model. Index, Naming and Registry Services

Reference model and functional invocation model are means to access models and resources declaratively and functionally.

Services are Models and thus reactive to its Container Model's events.

Index Service:
Mappings: types, indexing. Materialization. Functional.

Naming Service:
Ordering example. Encode order roles/rels in ctx. Materialization.
Naming encoding. Functional.

Registry Service:
Registry. Hierarchies. Contexts. Interactions. Registry. Materialization. Functional.

2.7 Functional API

Functional API is the core low-level client API exposed by a Container models and layers hierarchy for interaction with its held Resources.

2.7.1 Dialog / Dataflow Protocol. Message Statements

Statements (Message) are produced / consumed by Models. Reactive dataflow is performed 'applying' resources (materializing results from this operations). For establishment of this type of 'connection' an Adapter and a Port must be stated on each Runtime (Peer / Client / Server) side.

A Model itself is a Facade given a Profile narrowing Model's domain or subjects. A Profile is a Model which defines criteria for such narrowing, also for defining messages dispatch into peers.

Messages IO performs a dialog like protocol in which wildcard and variable placeholders are allowed. Message submission may resolve into a response which has placeholders to be filled which produces client (original sender) to receive the message for being responsable to resolve missing data from its own models.

2.7.2 Containers, Models and Messages discovery and binding (Profiles)

One should be allowed to interact with a Model (Facade) Container querying for its allowed message / events (Statements / Profiles)

2.7.3 Client bindings: DOM / DCI patterns over declarative Action Model layer metadata

From declarative layers metadata build DOM (Dynamic Object Model) with classes, instances, contexts, roles and interactions (DCI design pattern) objects for specific Runtime language bindings (Java, JavaScript, C, etc.).

JAF enabled content types and commands (JavaBeans Activation Framework) design pattern for REST enabled operations.

Monads. Materialize messages / dialogs. Addressable contexts / interactions. Functional design patterns.

2.8 Data loading. Federation through aggregated Peers

ETL. Datasources - Triples. Reference model features. Align. Statements aggregation. Sync.

Provenance. Temporal contexts, other contexts, metadata (rel. roles).

Example layer's statements.

3 Features

3.1 Align, merge. Identity resolution (by reference model). Algorithms

Services model listen/aggregates to events over Model Resources and resolves its mappings in context (mappings, order, hierarchy).

It also posts events materializing this knowledge into statements. For this it uses aggregation, discovery, learning, classification and regressions over previous knowledge.

Subject same Predicates. Predicate same Objects. Object same Subjects. Entity type recognition. Roles / rels. in context. By IRI or by graph shape / constraints (expression). Inference by Grammars (rules, productions learning).

All possible statements (Grammars). Merge with actual statements. Perform identity align (discard inviable results).

Facade / Templates declarative expressions. Constraints based.

3.2 Attribute and links discovery (by reference model)

Index Service. Facades / Templates declarative expressions.

X rel1 Y (roles, ctx). X rel2 Y.

X rel1 Y. Z rel2 Y. Z rel3 X.

Resource.apply(ctx : Resource, res : Resource);

3.3 Ordering and temporal alignment (by reference model)

Naming Services layer aggregates over Model Resources and resolves its positions in contexts (prev / next). It also posts events materializing this knowledge into statements. For this it uses aggregation, discovery, learning, classification and regressions over previous knowledge.

Facade / Templates declarative expressions. Constraint based.

3.8 Type inference (by reference model)

Index Service. Facade / Templates declarative expressions.

Attribute and links discovery.

3.5 Relationships browse

Registry Services layer aggregates over Model Resources and resolves its hierarchies in contexts (parents / children). It also posts events materializing this knowledge into statements. For this it uses aggregation, dicovery, learning, classification and regressions over previous knowledge.

Example: (Ctx.: rel, Peter, Joe) : where neighbors, then friends, and then partners. Transitions. Truth values (temporal, for ordered 'names').

Relationship browse context example: 'causeOf'. Temporal contexts comparison using octal values. Services result Resources (Events). Selector Resource (pointers this, that, they), vars, wildcards, composite Resource, reference model results Resources.

3.6 Grammars

Grammars (Facts example)
SPO: (SubjectKind, PredicateKind, ObjectKind) non-terminals.
Kinds: Subject, Predicate, Object (terminals).
Statements: Kinds / SPOs (rules, productions).

Primitive terms.
Opposite terms.
Negation terms.
Inverse terms.
Complement.

Materialize terms (context, role, term).

3.7 Datatypes

Primitive (enumerable) types.
Grammars. Rules (class). Production (instance).

3.8 Dimensions

Dimension. Unit. Value.
Materialize: (someValue, someProp, someObj).

Data, information, knowledge: price, price variation, price tendency. (Facts, Topics, Behavior Facade Models). Temporal (naming), hierarchical (registry) and contextual (index) resolution.

3.9 Learning

Service Models Features

Aggregation.

Classification.

Regression.

Dimensional.

Alignement.

Inference.

FCA. Augmentation.

Constraint based.

3.10 Upper ontology alignment. RDFS/OWL Model

Event propagation keeps RDFS/OWL Jena model (upper ontology) aligned and in sync with core models. RDF / SPARQL endpoints.

Bidirectional Adapter setup. Container's dataflow.

4 Backend

Learning and Big Data features.

4.1 Jena backend: Reified statements. RDFS / OWL

All declaration, instances and statements conform to a core ontology and are represented into a Jena model.

4.2 WebDAV / JCR (Registry)

Hierarchical datastore implementation for Registry Service. Contextual: a Resource may have multiple occurrences in different contexts.

JCRModel.

4.3 Lucene / Solr (Index)

Indexing component for Resources (attributes, links, relations). Contextual. Same resource multiple occurrences. Index Service backend.

LuceneModel.

4.5 JNDI (Naming)

Namespace domain resolution for ordered set (flow) of Resource (states) in context. Naming Service backend.

JNDIModel.
JMSModel.

5 Deployment

5.1 Runtime

Base environment configuration for core models execution / interactions.

5.2 Peers (Container)

Main Runtime deployable / bindable unit into Clients / Servers.

Models (Data, Grammar, Service).
Profile / Facade (Model).

5.3 Clients (Ports)

Exposes its underlying Peer as a service (API).

Port (src : Adapter, dest : Adapter).

5.4 Servers (Adapters)

Adapter (src : Model, dest : Model).

Configured to be bound as a source for containers (datasource).

6 Applications

6.1 Runtime, Servers, Clients

Dashboard: Peer manager console. Aggregated faceted browser of Behavior, Topic and Fact Facades / Models.

Declarative (model driven) architecture through language bindings of Action layers Facades, DOM (dynamic object models), JAF (JavaBeans Activation Framework) and DCI/MVC design pattern over Protocol/Dialogs.

Runtime

Environment configuration.

Adapters / Ports Container set up (datasources, peer application Models bindings).

Protocol endpoint. REST RDF Service endpoints. Queues, routes, transformations. Dispatch.

Messaging / Events dispatch:

1. Adapter / Port sends / receives Message in payload.
2. Container resolves/dispatchs to its Models (Profiles)
3. Model applies Message
4. Model resolves reply (and reply Profile)
5. Container broadcast reply

For reverse lookup (from Resource) build resolution / routing tables. Dataflow (reactive) traversal may start from leaves (Resource Object, Predicates, Subjects till Contexts) using tables or from outer Resources, each case being Resources visited (holds/apply invoked on them) and using their response for building adequate objects in the hierarchy (Reference model pattern matching: O, P, S, C).

Routing tables lazily populated on demand: when no data is contained for input of a particular Resource into routes tables then the full traversal is performed populating the tables accordingly for later use.

Callbacks: specific events, filter messages.

6.1 Search / Protocol

Statements Dialog example. Facade modification related arguments.

6.2 Integration

Semantic Web / LOD (ISO Jena backend).

Big (Linked) Data: Learning and augmentation.

Governance. MDM. ESB. Rules. Events. Workflows. Microservices (declarative applications)

Business Intelligence: ETL, Analysis, Mining.

Semantic mappings language bindings: Object-Triple Mappings. DOM / JAF augmented functional API. DCI: Map Facades into declarative / discoverable services (ie.: for a RESTFul HATEOAS API, OData endpoint, Solid or SOAP implementation).

DCI Design Pattern:

DOM (Dynamic Object Model). JAF (JavaBeans Activation Framework). Augmented functional API (language bindings).

Lab

Octal order relation encoding.

Encoding and addressing of IDs. Mappings to IRIs.

Patterns: ie.: SK matches Subjects. ID ops. Grammars.

Statements materialization: Triple IDs (x, y) (x, y) (x, y). Classes / Instances IDs. Each with its context, occurrence, attribute and value (CSPO).

Learning

Reference Model inference: Generate reference model services messages / events from materialized statements. Functional mappings inference: Index Service events (ID equivalence resolution). Registry Service events (Hierarchical aggregation). Naming Service events (Alignement and contextual ordering/sort).

ID Resolution / merge: merge all possible statements with KB. Resolve ambiguity using reference model, Templates.

Order resolution: materialize inverse, opposite, complement. Agrupate Events, flows, rules (action, passion, state) and kinds terms by ordering metadata (learn, dict.). Dimensional 'natural' order.

Everything is a Resource (class, instance, statements). Declarative schema, contexts, interactions, roles, data (class/instance in statements).

Command (monads) pattern. Chaining (declarative 'scripts', Resource pipes.

Message routes: outer resources match through inner resources tables propagating into contents.

ServiceMix Camel / JXTA Peers DHT.

Representation, Concept, Sign. This, That, 'Aquello’ ctx var pointers. Observer: roles of elements. Relationships roles. Transforms: precomposed signs or elements sequence. Rendering. Actor / Role.

Filters. Transforms. Pipes. Map / Memory based reference model resolution. IP address / Alpha RGB quad encoding (leverage existing networking and graphics hardware).

Profiles: interface, message patterns. Models: declarative state specs. Templates: state/pattern - pattern/state declarations (transforms). Actor / Role. Class / State (metaclass). Interaction / Context: Actor / Role Template triples (Context with Class / Metaclass Actors).

RDF/OWL Constraints Languages. TM, TMDM, TMRM, TMCL. Constraints for inference / reasoning / alignement. Macros (template based) input (doc forms) and extraction (auto/guided annotations).

Classes / interfaces. Factory, instantiation. Hierarchies. sameAs / domain / range / restrictions for RDFS/OWL representation. Constraints for alignment, reasoning and inference. Macros (document templates IO).