API4KP Metamodel: a Meta-API for Heterogeneous Knowledge Platforms

Tara Athan[1] Roy Bell[2] Elisa Kendall[3] Adrian Paschke[4] Davide Sottara[5]

[1] Athan Services (athant.com), West Lafayette, Indiana, USA
taraathan@gmail.com
[2] Raytheon, Fort Wayne, Indiana, USA
Roy_M_Bell@raytheon.com
[3] Thematix Partners LLC, New York, New York, USA ekendall@thematix.com
[4] AG Corporate Semantic Web, Freie Universitaet Berlin, Germany
paschke@inf.fu-berlin.de
[5] Department of Biomedical Informatics, Arizona State University, USA
davide.sottara@asu.edu

Abstract. API4KP (API for Knowledge Platforms) is a standard development effort that targets the basic administration services as well as the retrieval, modification and processing of expressions in machine-readable languages, including but not limited to knowledge representation and reasoning (KRR) languages, within heterogeneous (multi-language, multi-nature) knowledge platforms. KRR languages of concern in this paper include but are not limited to RDF(S), OWL, RuleML and Common Logic, and the knowledge platforms may support one or several of these. Additional languages are integrated using mappings into KRR languages. A general notion of structure for knowledge sources is developed using monads. The presented API4KP metamodel, in the form of an OWL ontology, provides the foundation of an abstract syntax for communications about knowledge sources and environments, including a classification of knowledge source by mutability, structure, and an abstraction hierarchy as well as the use of performatives (inform, query, ...), languages, logics, dialects, formats and lineage. Finally, the metamodel provides a classification of operations on knowledge sources and environments which may be used for requests (message-passing).

1 Introduction

The inherent complexity of many application domains - including but not limited to finance, healthcare, law, telecom and enviromental protection - paired with the fast pace of innovation, requires increasingly robust, scalable and maintainable software solutions. Design patterns have shifted from monolithic applications towards distribution and service-orientation. Standards have been published to improve interoperability. Model driven architectures (MDA) have been adopted to support declarative, platform-independent specifications of an application's business logic [7]. A special type of MDA, Knowledge Driven Architectures (KDA) [13], rely on models such as ontologies that are not only standard,

but also have a formal grounding in KRR. KDA, while not yet ubiquitous, have a variety of applications. We consider as a running example a scenario from the healthcare domain.

A connected patient system gathers input from biomedical devices, part of a publish-subscribe architecture, which post observations including physical quantities, spatio-temporal coordinates and other context information. The data can be represented in a device-specific format (e.g. using XMPP[6]) or as streams of RDF graphs over time. The vocabularies referenced in the streams include units of measure, time, geospatial and biomedical ontologies, expressed in RDF(S), OWL or Common Logic (CL). Healthcare providers will submit SPARQL queries and receive incremental streams as new data becomes available. A Clinical Decision Support System (CDS), implemented using event-condition-action (ECA) rules, will also react to events simple (e.g. a vital parameter exceeding a threshold) and complex (e.g. a decreasing trend in the average daily physical activity) and intervene with alerts and reminders. If an alert is not addressed in a timely fashion, it will escalate to another designated recipient. Some patients will qualify for clinical pathways and the system will maintain a stateful representation of their cases, allowing clinicians to check for compliance with the planned orders (e.g. drug administrations, tests, procedures, ...). This representation will include an ontology-mediated abstraction of the patient's electronic medical record, extracted from the hospital's database. As medical guidelines evolve, the logic of the pathway may need revision: queries to the patient's history should be contextualized to whatever logic was valid at the time orders were placed.

From a systems-oriented perspective communicating entities in distributed systems are processes (or simple nodes in primitive environments without further abstractions) and from a programming perspective they are objects, components or services/agents. They may be single-sorted or many-sorted, with sorts being characterized by the kind of communications that may be initiated, forwarded or received, and by the kind of entity that may be received or forwarded from or sent to. Communication channels may in general be many-to-many and uni- or bidirectional. Each communication has a unique source; multi-source communications are not modelled directly, but are emulated by knowledge sources that publish streams that may be merged to give the appearance of multiple sources. We will allow for failure, either in communication or in execution, but do not specify any particular failure recovery strategy. Various types of communication paradigms are supported from strongly-coupled communication via low-level inter-process communication with ad-hoc network programming, loosely coupled remote invocation in a two-way exchange via interfaces (RPC/RMI/Component/Agent) between communicating entities, to decoupled indirect communication, where sender and receiver are time and space uncoupled via an intermediary such as a publish-subscribe and event processing middleware. The communication entities fulfill different roles and responsibilities (client, server, peer, agent) in typical architectural styles such as client-server, peer-to-peer and multi-agent systems. Their placement (mapping) on the phys-

[6] http://xmpp.org/rfcs/rfc3920.html

ical distributed infrastructure allows many variations (partitioning, replication, caching and proxing, mobile) such as deployment on multiple servers and caches to increase performance and resilience, use of low cost computer resources with limited hardware resources or adding/removing mobile computers/devices.

Given this variety of architectural requirements, an abstraction is required to facilitate the interchange, deployment, revision and ultimately consumption of formal, declarative pieces of knowledge within a knowledge-driven application. In 2010 the Object Management Group (OMG) published the first formalized set of KDA requirements in an RFP titled "the API for Knowledge Bases (API4KB)". In 2014 the OMG published a second RFP titled "Ontology, Model, and Specification Integration and Interoperability (OntoIOp)" [9]. This second RFP contains the requirements for a substantial part of the API4KB, and a submission, called DOL [1] is near completion. To address the remaining aspects of the RFP, a working group is creating a standard meta-API, called API4KP, for interaction with the Knowledge Platforms at the core of KDAs.

To provide a semantic foundation for the API4KP operations and their arguments, we have created a metamodel of knowledge sources and expressed it as an OWL ontology[7]. The primary concepts of the API4KP metamodel are described in Sec. 2, with details for structured knowledge resources and their relationship to nested functor structures in Sec. 3. In Sec. 4 we provide an application of the metamodel to the healthcare scenario. Related work is discussed in Sec. 5, with conclusions and future work described in Sec. 6.

2 Upper-level Concepts and Basic Knowledge Resources

The current API4KP metamodel focuses on the notion of knowledge resources, the environment where the resources are to be deployed and their related concepts. The metamodel is hierarchical, with a few under-specified concepts at the upper levels, and more precisely defined concepts as subclasses. These upper-level concepts indicate, at a coarse level, the kinds of things that are in the scope of API4KP. The main upper-level concepts in the API4KP metamodel are

Knowledge Source: source of machine-readable information with semantics. Examples: a stream of RDF graphs providing data from biomedical devices, a stateful representation of a patient's history with OWL snapshots, or a database with a mapping to an ontology.

Environment: mathematical structure of mappings and members, where the domain and codomains of the mappings are members of the environment. Example: a KRR language environment containing semantics-preserving translations from RDF and OWL into CL, assisting in the integrated interpretation of a stream of RDF graphs and OWL ontologies.

Knowledge Operation: function (possibly with side-effects. i.e. effects beyond the output value returned) having a knowledge source, environment or operation type in its signature. Examples: publishing or subscribing to a stream

[7] `https://github.com/API4KBs/api4kbs`

of RDF graphs; submitting a SPARQL query; initiating an ECA Rulebase; checking for compliance with plans; revising an ontology of guidelines.

Knowledge Event: successful evaluation or execution of a knowledge operation by a particular application at a particular time[8] Examples: when a nurse activates a biomedical device, a stream of RDF graphs is "published" describing a patient's vital signs; a specialist, like a cardiologist, taps the heartrate symbol on a touchscreen that results in the submission of a SPARQL query about a semantically-defined subset of a patient's vital signs.

These definitions are intentionally vague so as to be adaptable to a variety of implementation paradigms. We have developed a hierarchy of *knowledge source level* of abstraction that is a generalization of the FRBR [3] Work-Expression-Manifestation-Item (WEMI) hierarchy of abstraction tailored for machine-readable KRR languages. The fundamental building blocks of knowledge sources are *basic knowledge resources*, which are immutable knowledge sources without structure. Subclasses of basic knowledge resources are defined according to their knowledge source level.

Basic Knowledge Expression: well-formed formula in the abstract syntax of a machine-readable language.[9] Example KE1: the instance of the OWL 2 DL abstract syntax for the latest version of a biomedical ontology from an ontology series KA1 defining observable entities, such as the 2015 international version of the SNOMED-CT knowledge base[10] (see also the definition of Basic Knowledge Asset below). This ontology differs from other versions of the series only in the natural language definitions.

Basic Knowledge Manifestation: character-based embodiment of a basic knowledge expression in a concrete dialect. Example KM1: the OWL/RDF XML Document Object Model (DOM) document instance of example KE1.

Basic Knowledge Item: single exemplar of a basic knowledge manifestation in a particular location. Example KI1: a file on a network server embodying example KM1.

Basic Knowledge Asset: equivalence class of basic expressions determined by the equivalence relation of an asset environment (see Sec. 2.2.) Example KA1: an OWL2 DL series for a biomedical ontology, viewed as an equivalence class of basic knowledge expressions, including example KE1, according to a semantics-preserving environment for the OWL2 DL language where the mapping to the focus language strips the natural language definitions from the axioms.

[8] Some Knowledge Operations can be used as transition functions for a mutable knowledge source, where their evaluation describes an event in the sense of [14], as a state transition of a dynamic entity; we generalize this concept of events because not all API4KP Knowledge Events correspond to state transitions.

[9] The use of "basic" in API4KP differs from its usage in DOL - a DOL basic OMS (ontologies, models and specifications) is a set, and corresponds to a Set-structured knowledge asset in API4KP.

[10] http://browser.ihtsdotools.org/

API4KP lifting/lowering operations (see 2.4) provide transformations from one level to another complying with the following relations:

exemplify: to instantiate (a knowledge manifestation) in particular format(s) and at particular location(s) (address in some virtual address space). Example: KI1 exemplifies KM1, KM1 prototypes KI1. Inverse: *prototype*

embody: to represent (a knowledge expression) in concrete syntax(es) (dialects) of particular KRR language(s). Example: KM1 embodies KE1, KE1 parses KM1. Inverse: *parse*

express: to represent (a knowledge asset) in abstract syntax(es) of particular KRR language(s). Example: KE1 expresses KA1, KA1 conceptualizes KE1. Inverse: *conceptualize*

2.1 Mutability

Following RDF concepts[11], knowledge sources are characterized as mutable or immutable. Immutable knowledge sources are called *knowledge resources*. In this context, immutable does not necessarily mean static; a stream of knowledge, e.g. a feed from a biomedical device, may be considered an *observable* knowledge resource that is revealed over time, as described further in Sec. 3. A *mutable knowledge source* is a container that has, at any point in time, an explicit state that is fully represented by a knowledge resource, e.g. the snapshot of a patient's current condition (with timestamp). The language, structure and content of a mutable knowledge source may change over time, but the abstraction level is unchanging. We distinguish between the *implicit* state that a mutable knowledge source holds indirectly when operators such as actions, complex event patterns or aggregations are computed, and the explicit state that evolves with time and that can be managed explicitly by an additional state transformer component responsible for *explicit* state management, concurrency control, reasoning (specifically, inference of state deltas), and state updates. There are various ways to manage explicit state, e.g. embedded inside the processors of the knowledge source in global variables or state-accumulating variables or tuples that are available either locally to an individual operator or across the operators as a shared storage, or with explicit state and concurrency control which lies outside of knowledge resource processors, e.g. by threading the variables holding state through a functional state transformer and by using State monads (see 3), which exist within the context of another computation or transformation, thus allowing to attach state information to any kind of functional expression.

2.2 Environments

In DOL, a concept of heterogeneous logical environment is defined as "environment for the expression of homogeneous and heterogeneous OMS, comprising a logic graph, an OMS language graph and a supports relation". In API4KP, we generalize this concept of environment as follows.

[11] http://www.w3.org/TR/rdf11-concepts/

Categorical Environment: environment with an associative composition operation for mappings, that is closed under composition and contains an identity mapping for every member

Language Environment: environment whose members are languages

Focused Environment: nonempty environment which has a member F (called the focus or focus member) such that for every other member A, there is a mapping in the environment from A to F

Preserving Environment: environment where every mapping preserves a specified property

Asset Environment: focused, categorical, preserving language environment where the focus is a KRR language

The special case where all languages in an asset environment are KRR languages supporting model-theoretic semantics without side-effects (logics), and the preserving property is characterized by a logical graph reduces to a heterogeneous logical environment as defined in DOL.

The Knowledge Query and Manipulation Language [2] introduced the concept of *performatives*, which was later extended by FIPA-ACL[12]. The KRR Languages covered by API4KP include ontology languages (e.g. OWL), query languages (e.g. SPARQL), languages that describe the results of queries, events and actions (e.g KR RuleML), and declarative executable languages (e.g. Prolog, ECA RuleML). In the latter case, the languages typically includes syntactic constructs for performatives, e.g. *inform*, *query*, and the description of a knowledge resource may include a list of the performatives that are used within it. Performatives will be modelled as *operations* as defined in Sec. 2.4.

2.3 Descriptions

As stated above, we do not make assumptions regarding the drivers for communications, e.g. an implementation may be message-driven, event-driven, or a combination of both. However, our metamodel takes a message-centric perspective, with the message body typically being a description of a knowledge source or a knowledge operation.

A *knowledge source description* is a knowledge resource whose subject matter is another knowledge source, which may be expressed, e.g., as an OWL ontology of individuals or an RDF graph. The properties and classes in the API4KP namespace that may be employed in knowledge source descriptions are listed in the following tables and formalized in the API4KP OWL ontologies. Further, IRIs in other namespaces may be used to express metadata within a knowledge source description. A description about the description itself may be referenced through an IRI, or included within the description explicitly through the :hasDescription property, OWL annotations, or as an RDF dataset.

[12] http://www.fipa.org/repository/aclspecs.html

Key	Value
Y	exactly 1
Yor	1 or more
Y?	0 or 1
Y*	0 or more
N	exactly 0
I[or?*]	indirect

Table 1. Legend

Prefix	Expansion
:	`http://www.omg.org/spec/API4KP/API4KPTerminology/`
ks:	:KnowledgeSource/
kr:	:KnowledgeResource/
ka:	kr:Asset/
ke:	kr:Expression/
km:	kr:Manifestation/
ki:	kr:Item/
lang:	:Language/
map:	:Mapping/
xsd:	`http://www.w3.org/2001/XMLSchema#`

Table 2. Prefix Mappings

Property	Range	ka:	ke:	km:	ki:
:hasIdentifier	:Identifier	Y?	Y?	Y?	Y?
:level	ks:Level	Y	Y	Y	Y
:usesPerformative	:Operation	I*	Y*	I*	I*
:hasLocator	:Address	Y?	Y?	Y?	Y
:usesLanguage	:Language	I*	Y*	I*	I*
:usesDialect	km:Dialect	N	N	Y*	I*
:usesConfiguration	ki:Configuration	N	N	N	Y*
:accordingTo	lang:Environment	Y	N	N	N
:isBasic	xsd:boolean	Y	Y	Y	Y
:isOutputOf	ev:	Y?	Y?	Y?	Y?
:hasMetaData	:KnowledgeResource	Y*	Y*	Y*	Y*
:hasDescription	:KnowledgeResource	Y*	Y*	Y*	Y*

Table 3. Knowledge Resource Metamodel

2.4 Operations and Events

In the API4KP metamodel, the building blocks for all knowledge operations are *actions* – unary functions, possibly with side-effects and possibly of higher-order. Actions are defined in terms of their possible events. To maintain a separation of concerns, side-effectful actions are assumed to be void, with no significant return value. Particular kinds of actions include:

Lifting Action: side-effect-free action whose output is at a higher knowledge source level than the input

Lowering Action: side-effect-free action whose output is at a lower knowledge source level than the input

Horizontal Action: side-effect-free action whose output is at the same knowledge source level as the input

Idempotent Action: side-effect free action that is equal to its composition with itself (A = A o A)

Higher-Order Action: side-effect-free action whose input or output (or both) is an action

Property	**Domain**	**Range**	**Inverse**
:exemplify (?)	ki:	km:	:prototype (*)
:embody (?)	km:	ke:	:parse (*)
:express (*)	ke:	ka:	:conceptualize (*)

Table 4. Knowledge Resource Elevation Properties

Property	**Range**	**ke:Language**	**km:Dialect**	**ki:Configuration**
:hasIdentifier	:Identifer	Y	Y	Y
:hasLocator	:Address	N	N	Y?
:supports	:Logic	Y	I	I
:usesLanguage	ke:Language	N	Y	I
:usesDialect	km:Dialect	N	N	Y
:usesFormat	ki:Format	N	N	Y
:location	:Address	N	N	Y

Table 5. Knowledge Resource Configuration Metamodel

Lifting and lowering are utility actions for changing the knowledge source level, e.g. parsing and IO. Horizontal actions are useful e.g. for constructing structured knowledge sources, while higher-order actions are needed to specify more complex operations e.g. querying.

In the metamodel, we define two void actions that have side-effects on the state of mutable knowledge resources:

Put: void action whose input is a mutable knowledge source and has the side-effect of setting the mutable knowledge source to a particular specified state

Update: void action whose input is a mutable knowledge source and has the side-effect of setting the mutable knowledge to a new state that is the result of applying a side-effect-free action to the current state

A side-effectful operation can be considered idempotent if its successful execution multiple times (synchronously) leads to no additional detectable side-effects beyond that of the first execution. Note that this is a different, but related, concept of idempotence than that for side-effect-free actions. An Update action based on an idempotent side-effect-free action is idempotent in this sense, an important factor in failure recovery.

3 Structured Knowledge Resources

We generalize the DOL concept for structured OMS to define a concept of structured knowledge resource for each level of abstraction. In DOL, a structured OMS "results from other basic and structured OMS by import, union, combination, ... or other structuring operations". In API4KP, A *structured knowledge resource* is a collection whose components are knowledge resources of the same level of abstraction; structuring knowledge operations are described in Sec. 2.4.

Property	Range	T:Environment
:hasIdentifier	:Identifier	Y?
:mapping	T:Mapping	Y*
:focus	T:	Y?
:preserves	T:EquivalenceRelation	Y*
:isOutputOf	ev:	Y?

Table 6. Generic Environment Metamodel. The generic prefix T: specifies the member type. Specific environments include lang:Environment (a system of mappings between the abstract syntax of languages) .

Property	Range	T:Mapping
:hasIdentifier	:Identifier	Y?
:location	:Address	Y?
:start	T:	Y
:end	T:	Y
:preserves	:EquivalenceRelation	Y*
:usesLanguage	map:Language	Y*
:isBasic	xsd:boolean	Y
:components	T:MappingList	Y?

Table 7. Generic Mapping Metamodel

Structured Knowledge Expression: collection of knowledge expressions (either structured or basic), which are not necessarily in the same language and may themselves have structure. Example KE2: a heterogeneous collection of streaming data and RDF graphs, together with static OWL ontologies and CL texts, and ECA rules describing actions of a CDS. Example KE3: the OWL 2 DL ontology series KA1, viewed as a collection of expressions rather than an equivalence class.

Structured Knowledge Manifestation: collection of knowledge manifestations (either structured or basic), which are not necessarily in the same language or dialect and may themselves have structure. Example KM2: a heterogeneous structure of RDF Turtle, OWL Manchester as sequences of string tokens, and XMPP, OWL/XML, ECA RuleML and CL XCL2 (the XML-based dialect of Common Logic Edition 2) as XML DOM documents embodying example KE2.

Structured Knowledge Item: collection of knowledge items (either structured or basic), which are not necessarily in the same language, dialect, format or location, and may themselves have structure. Example KI2: a heterogeneous structure of an RDF triple store, network connections to binary input streams cached in a MySQL database, RuleML XML files on a local hard drive and CL XCL2 files on a network server in a content management system, exemplifying example KM2.

Structured Knowledge Asset: collection of knowledge assets (either structured or basic), which are not necessarily according to the same environment, but where there is a unique language that is the focus of the environment of

Property	Domain	Range	Inverse
:hasEvent (*)	op:	ev:	:isEventOf (1)
:executes (*)	:Application	ev:	:isExecutedBy (1)
:input (?)	ev:ActionEvent	:	:isInputOf (*)
:output (?)	ev:	:	:isOutputOf (?)
:atTime (1)	ev:	xsd:dateTime	

Table 8. Knowledge Resource Operation and Event Properties

each component. Example KA2: a heterogeneous structure of assets conceptualized from the RDF, OWL and CL expressions of example KE2 according to an environment that provides translations from RDF or OWL into CL, and an ontology-based data access (OBDA) source schema providing a mapping from XMPP schemas to OWL.

To assist in defining operations on structured knowledge sources while still maintaining generality, the collection structure of a structured knowledge resource is required to arise from a monadic functor (monad). Collection structures that satisfy these requirement include sets, bags and sequences, but other useful structures also meet these requirements.

3.1 Monads

In seminal work that established a theoretical foundation for proving the equivalence of programs, Moggi [8] applied the notion of monad from category theory [5] to computation. As defined in category theory, a monad is an endofunctor on a category C (a kind of mapping from C into itself) which additionally satisfies some requirements (the monad laws). In functional programming, monads on the category with types as objects and programs as arrows are employed. For example, the List[_] typeclass is a monad, e.g. List[Int], a list of integers, is a type that is a member of the List[_] monad.

Each monad M has functor M and two natural transformations as follows (exemplified for the List monad where lists are denoted with angle brackets)

- unit: A ⇒ M[A] lifts the input into the monad (e.g. unit(2) = ⟨2⟩)
- join: M[M[A]] ⇒ M[A] collapses recursive monad instances by one level (e.g. join(⟨⟨1, 2⟩, ⟨3, 4⟩⟩) = ⟨1, 2, 3, 4⟩)
- M: (A ⇒ B) ⇒ (M[A] ⇒ M[B]) takes a function between two generic types and returns a function relating the corresponding monadic types (e.g. $List($ s ⇒ 2*s)(⟨1, 2⟩) = ⟨2, 4⟩)

Note that we choose the category-theory-oriented unit and join transformations [16] as fundamental in this development of the monad laws because it is useful for later discussion on structured expressions, whereas the functional-programming-oriented treatment based on unit and bind :≡ join o M (aka flatmp), is more concise. Monads of relevance to API4KP include, but are not limited to

Try: handles exceptions, has subclasses Success, which wraps a knowledge resource, and Failure, which wraps a (non-fatal) exception

IO: handles IO side-effects, wraps a knowledge resource and an *item configuration*

Task: handles general side-effects, wraps a knowledge resource and a description of a side-effectful task

Stream: a.k.a. Observable handles concurrent streams, wraps a sequence of knowledge resources that become available over time

State: handles state, wraps a knowledge resource (the state) and implements state transitions

These monad functors may be composed; for example, given a basic knowledge expression type E, the type (State o Try o List) [E] :≡ State[Try[List[E]]] may be defined. In general, the composition of monads is not necessarily a monad.

3.2 Nested Monadic Structures

In DOL, the concept of structured expression using sets is introduced. For example, let B be the category of (basic) CL text expressions, and OptionallyNestedSet[B] :≡ B + NestedSet[B], where NestedSet[B] :≡ Set[OptionallyNestedSet[B]] ≡ Set[B + NestedSet[B]] is the recursive type definition of set-structured CL expressions. An instance of type NestedSet[B] is a Set whose members are either basic leaves (of type B) or structured branches (of type NestedSet[B]).

The Set monad is appropriate for defining structured expressions in monotonic logics, like CL, because the order and multiplicity of expressions in a collection has no effect on semantics. The semantics of CL is provided by the CL interpretation structure that assigns a truth-value to each basic CL text expression. The truth-value of a set of CL text expressions is true in an interpretation J if each member of the set maps to true in J. The truth value J(y) of a NestedSet-structured CL expression y is defined to be J(flatten(y)), where flatten(y) is the set of leaves of y.

We generalize this approach for defining the semantics of structured expressions to an arbitrary language L with basic expressions E and NestedM structured expressions. We assume that

- M is a monad on the category of types,
- model-theoretic semantics is supplied through an interpretation structure J defined for basic expressions in E and simply-structured expressions M[E + **0**], where **0** is the empty type.
- a post-condition contract for side-effects is specified by a truth-valued function P(F, y) for all supported void knowledge actions F and all y in E + M[E + **0**].

Let N[_] be the NestedM monad corresponding to the minimal (finite) fixed point of N[E] :≡ M[E + N[E]], where A + B is the coproduct[13] of types A and

[13] The coproduct, a.k.a. disjoint union, A + B can be treated as the type (False x A) | (True x B), with the first (Boolean) argument of the pair providing the intention of left or right injection (inl and inr). The operation f + g on functions f and g means (f+g)(inl(a)) :≡ f(a) and (f+g)(inr(b)) :≡ g(b).

B. We name the NestedM monad by prepending "Nested" to the name of the underlying monad; thus, NestedSet[E] :≡ Set[E + NestedSet[E]].

If E is a type of basic knowledge resources, then the monad OptionallyNestedM[E] :≡ E + NestedM[E] ≡ E + M[OptionallyNestedM[E]] is the corresponding type of knowledge resources that are either basic or structured. We note that OptionallyNestedM[E] is a free monad[14] of M; this property holds for a large class of functors and does not depend on M being a monad.

NestedM is also a monad under an appropriate join transformation; this property does depend on M being a monad. Further, we take advantage of the monadic properties of M in order to "flatten" the nested structure for purposes of interpretation and pragmatics. The unit, map and join functions for NestedM are defined in terms of the unit, join, and map functions for monad M, and the constructors, recursor and bimap function of the coproduct. The details and proof[15] that NestedM structures satisfy the monad laws depends on the use of the coproduct to handle the union of types, so that the left or right intention is indicated even in the case when the types are not disjoint.

For all y ∈ Q[E] :≡ OptionallyNestedM[E], we define a flatten transformation flatten(y). Let I be the identity transformation, N[E] :≡ NestedM[E], joinN be the join natural transformation of monad N, Q_1 :≡ E + M[E + **0**], and

joinN: N[N[E]] ⇒ N[E] ∋ joinN :≡ joinM o M(I + unitM o inr o joinN)
level: Q[E] ⇒ N[E] ∋ level :≡ unitM ∘ inl + I
flatten: Q[E] ⇒ Q_1[E] ∋ flatten(y) = y if y ∈ Q_1[E],
flatten(y) = flatten(joinN ∘ M(inl ∘ level)(y)) otherwise

Then for all y ∈ Q[E], we may define the interpretation J(y) :≡ J(flatten(y)), with entailments defined accordingly. Implementations that honor the semantics must satisfy P(F)(y) = P(F)(flatten(y)), where P(F) is a function representing the post-conditions after execution of side-effectful knowledge operation F on the knowledge resource y.

The monad laws and the flatten transformation have been verified experimentally for NestedSet and NestedList monads by implementation in Java8 together with the Functional Java[16] libraries, with the source available on Github[17]. Informal tests confirm that the map and join operations are linear in the size of the collection, as expected.

3.3 Heterogeneous Structures

Suppose A and B are expression types of two languages where an environment provides a semantics-preserving transformation T from B to A. Further suppose that an interpretation mapping is defined on A + M[A + 0]. The coproduct E :≡ A + B defines the basic knowledge expressions in this environment, while

[14] http://ncatlab.org/nlab/show/free+monad
[15] `https://github.com/API4KBs/api4kbs/blob/currying/Monad_Trees.pdf`
[16] http://www.functionaljava.org/
[17] `https://github.com/ag-csw/Java4CL`

structured expressions are N[E] :≡ NestedM[E], and the coproduct Q[E] :≡ E + N[E] is the type for all expressions in this environment, basic or structured.

Using the transformation T from the environment, we may define the interpretation J_+ of structured expressions of type NestedM[A+B] in terms of the interpretations J of basic expressions in A and structuring operations. In particular,

$$J_+(\mathbf{x}) :\equiv J(\text{ NestedM}(T + \text{I})(\text{flatten}(\text{x}))\ ) \equiv J(\text{flatten}(\text{NestedM}(T + \text{I})(\text{x}))\ )$$

Notice that the expressions of type B are not required to be in a knowledge representation language. They could be in a domain-specific data model based on, e.g., XML, JSON or SQL. The semantics of expressions of type B are derived from the transformation to type A, the focus knowledge representation language of the environment. API4KP employs this feature to model OBDA and rule-based data access (RBDA).

Structured expressions can always be constructed in a monad that has more structure than necessary for compatibility with the semantics of a given language. For example, List and Stream monads can be used for monotonic, effect-free languages even though the Set monad has sufficient structure for these languages; a forgetful functor is used to define the semantics in the monad with greater structure in terms of the monad of lesser structure. A heterogeneous structure of languages containing some languages with effects and others without effects (e.g. an ECA rulebase supported by ontologies) could thus make primary use of an NestedM monad that preserves order, such as NestedList or NestedStream, while permitting some members of the collection to have a NestedSet structure.

While an immutable knowledge source (i.e. a knowledge resource) has a specific structure, as discussed above, a mutable knowledge source has structure only indirectly through the structure of its state. In general, the structure of a mutable knowledge source's state changes arbitrarily over time, but could be restricted in order to emulate common dynamic patterns. Simple examples include state as a basic knowledge resource (linear history without caching), a key-value map with values that are basic knowledge resources (branching history without caching), or a sequence of basic knowledge resources (linear cached history).

4 Metamodel Appplied to the Scenario

In the connected patient scenario, an RDF stream from a biomedical device can be modelled using a Stream monad. A query registered against this RDF Stream will generate another Stream, with each Stream item containing additions (if any) to the query results due to the assertion of the newly-arrived graph. Because RDF has monotonic semantics, the accumulated query results will always be equivalent to the result of the query applied to the accumulated graphs of the stream. Cumulative queries and other cumulative operations on Streams may be implemented through *fold* operations, while windowing and filtering are implemented through map. The connected-patient system uses a heterogeneous language environment to map input XMPP data from biomedical devices into a KRR language, e.g. RDF, employing terms from a vocabulary defined in

a common ontology. Thus streaming data may be transformed into streaming knowledge which is queryable as discussed in the previous item. The structure of this system may be modelled as a NestedSet of Streams, since each device streams its output asynchronously. State, Task and IO monads are appropriate to the use case of an active knowledge base where evaluation of an operation leads to side-effects; the choice of monad depends on the nature of the side-effects and the implementation. Equivalence of such knowledge resources requires not only the same entailments, but also side-effects that are equivalent. The CDS monitoring our connected patient may be modelled using a State monad, where the sending of a message is a side-effect. The connected patient's case history may be modelled as a mutable knowledge asset because of the possibility of correction of the history without a change of case identifier. The modular nature of medical records is amenable to NestedSet (a set of laboratory test results) or NestedList (a procedure performed) structures. Although some aspects, such as the addition of new medical orders, would fit with the Stream structure, queries of the case history are not expected to produce streaming results, and so the mutable asset model is a better fit than a Stream-based model. Failure recovery in the CDS alert system may be modelled using the Try monad, so that results can be reported as Success or Failure. A Success response is a wrapper around a result from the successful execution of a request. A Failure response includes information about the nature of the failure (e.g. timeout exception) so that the system can recover appropriately, e.g. by escalating to another recipient. A possible extension of the CDS which allows a streaming model in combination with explicit state management and concurrency follows an implementation [4] that was demonstrated for sports competitions using the Prova rule engine[18].

5 Related Work

While various APIs and interface languages for different knowledge platforms and representation languages exist[19], API4KP provides a unified abstract API metamodel. Also, various ontologies and semantic extensions for Semantic Web Service interfaces [20] as well as REST interfaces [21] exist. None of them is specific to APIs for knowledge platforms and services in general. Some works present operations on structured knowledge bases (e.g. [15]), but are not exposed using APIs. General top-level ontologies and general model-driven architecture and software engineering metamodels have certain overlaps with the concepts used in API4KP, but fulfill a different purpose. They can be used for the representational analysis of API4KP. [10]. From a conceptual point of view reference models and

[18] https://prova.ws/

[19] e.g., OWL API , JSR-94 , Linked Data Platform , RuleML Rule Responder IDL , OntoMaven and RuleMaven , FIPA ACL , CTS-2

[20] e.g., OWL-S, WSDL-S, SAWSDL, SWWS / WSMF, WSMO / WSML, Meteor-S, SWSI

[21] Semantic URLs, RSDL, SA-Rest, Odata

reference architectures [22] for knowledge platforms are related and API4KB can be used in such reference architectures for the description of functional interfaces and component APIs.

So, DOL is the most closely related endeavor. The API4KP metamodel introduces the following generalizations of DOL concepts:

- Knowledge sources can have different levels of abstraction. DOL's OMS concept correspond to knowledge expressions, while we consider also the levels of asset, manifestation and item.
- Knowledge sources can be mutable or immutable. DOL's OMS correspond to immutable knowledge expressions.
- Each API4KP knowledge asset is conceptualized according to a customizable environment, instead of assuming a single logical environment in which all OMS are interpreted.
- Environment members can be any language with an abstract syntax, instead of requiring each member to have a specific semantics. Only the focus of the environment is required to have its own semantics.
- Semantics is generalized to include side-effects as well as logical entailment.
- Structured knowledge resources may have structures other than nested sets.

The variety of monad structures necessary to model the diversity of usecases demonstrates that a high level of abstraction is needed to define operations for modifying knowledge resources - adding, subtracting or modifying. Category theory provides the tools for these abstractions, through applicative functors (a generalization of monads having a binary operator allowing a structure of functions to be applied to another structure), catamorphisms (generalization of aggregation over a list to other monads) and anamorphisms (e.g. generation of a list from a seed and recursion formula) [6].

6 Conclusion and Future Work

The primary contributions of this paper are two-fold: (i) a metamodel of heterogeneous knowledge sources, environments, operations and events, providing an abstract syntax for interaction with the Knowledge Platforms at the core of KDAs and (ii) a structure of nested monads, as the conceptual basis of structured knowledge resources in the metamodel, supporting modularity, state management, concurrency and exception handling. We have used a scenario from healthcare to show the kinds of complexities that will be needed and that our metamodel in combination with monads will meet this challenge. The healthcare scenario brought up things such as input RDF streams, heterogeneous language environments, and mutable persistent storage, and we have shown how they will be accomplished. Future work on API4KP may include a generalization of the approach to include structures based on applicative functors, and operations in terms of catamorphisms and anamorphisms, as well as the population of the ontology with specifications of additional operations, especially querying and life-cycle management.

[22] e.g., the EPTS Event-Processing Reference Architecture [11] and the EPTS/RuleML Event Processing Standards Reference Model [12]

References

1. The distributed ontology, model, and specification language (dol). https://github.com/tillmo/DOL/blob/master/Standard/ebnf-OMG_OntoIOp_current.pdf
2. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: Proceedings of the Third International Conference on Information and Knowledge Management. pp. 456–463. CIKM '94, ACM, New York, NY, USA (1994), http://doi.acm.org/10.1145/191246.191322
3. IFLA Study Group on the Functional Requirements for Bibliographic Records: Functional requirements for bibliographic records : final report. http://www.ifla.org/publications/functional-requirements-for-bibliographic-records (1998), accessed: 2007-12-26
4. Kozlenkov, A., Jeffery, D., Paschke, A.: State management and concurrency in event processing. In: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009 (2009), http://doi.acm.org/10.1145/1619258.1619289
5. Mac Lane, S.: Categories for the Working Mathematician (Graduate Texts in Mathematics). Springer (1998)
6. Meijer, E., Fokkinga, M., Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. pp. 124–144. Springer-Verlag (1991)
7. Mellor, S.J., Kendall, S., Uhl, A., Weise, D.: MDA Distilled. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
8. Moggi, E.: Notions of computation and monads. Selections from 1989 IEEE Symposium on Logic in Computer Science 93(1), 55–92 (Jul 1991), http://www.sciencedirect.com/science/article/pii/0890540191900524
9. Object Management Group (OMG): OntoIOp request for proposal. http://www.omg.org/cgi-bin/doc?ad/2013-12-02
10. Paschke, A., Athan, T., Sottara, D., Kendall, E., Bell, R.: A Representational Analysis of the API4KB Metamodel. In: Proceedings of the 7th Workshop on Formal Ontologies meet Industry (FOMI 2015). Springer-Verlag (2015)
11. Paschke, A., Vincent, P., Alves, A., Moxey, C.: Tutorial on advanced design patterns in event processing. In: Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012. pp. 324–334 (2012)
12. Paschke, A., Vincent, P., Springer, F.: Standards for Complex Event Processing and Reaction Rules. In: Rule-Based Modeling and Computing on the Semantic Web, 5th International Symposium, RuleML 2011- America, Ft. Lauderdale, FL, Florida, USA, November 3-5, 2011. Proceedings. pp. 128–139 (2011)
13. Rector, A.: Knowledge driven software and "fractal tailoring": Ontologies in development environments for clinical systems. In: Proceedings of the 2010 Conference on Formal Ontology in Information Systems: Proceedings of the Sixth International Conference (FOIS 2010). pp. 17–28. IOS Press, Amsterdam, The Netherlands, The Netherlands (2010)
14. Rosemann, M., Green, P.: Developing a Meta Model for the Bunge-Wand-Weber Ontological Constructs. Inf. Syst. 27(2), 75–91 (Apr 2002), http://dx.doi.org/10.1016/S0306-4379(01)00048-5
15. Slota, M., Leite, J., Swift, T.: Splitting and updating hybrid knowledge bases. Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP'11) Special Issue 11(4-5), 801–819 (2011)

16. Wadler, P.: Comprehending monads. In: Mathematical Structures in Computer Science. pp. 61–78 (1992)