Date: January 2024

Application Programming Interfaces for Knowledge Platforms (API4KP)

Version 1.0

OMG Document Number: formal/24-01-10

Standard Document URL: https://www.omg.org/spec/API4KP/

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

DISCLAIMER OF WARRANTY

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process, we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page https://www.omg.org, under Documents, Report a Bug/Issue.

Table of Contents

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at https://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

https://www.omg.org/spec

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
9C Medway Road, PMB 274
Milford, MA 01757
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Issues

All OMG specifications are subject to continuous review and improvement. As part of this process, we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page https://www.omg.org, under Documents, Report a Bug/Issue.

1 Scope

1.1 General Scope

This OMG specification defines the Application Programming Interfaces for Knowledge Based Systems and Platforms (API4KP), in response to the OMG's Application Programming Interfaces for Knowledge Bases (API4KB) RFP. The purpose of these APIs is to facilitate the development and integration of knowledge graphs and knowledge-based systems in a broader enterprise framework. They provide a standard interface between client applications, knowledge resources and the platforms used to manage and deliver them - including but not limited to editors, repositories, and reasoners/ rule engines.

1.2 Background Information

The development of 'Knowledge Driven' applications is rooted in the discipline of Knowledge (i) Representation and (ii) (Automated) Reasoning and can be augmented by (iii) Knowledge Acquisition and (iv) Knowledge Management and Delivery. Knowledge Representation and Reasoning (KRR) is part of the broader field that lies at the intersection of Artificial Intelligence (AI), Linguistics and Natural Language Processing (NLP), Machine Learning (ML) and Data Mining, Management, and Governance. Over the years, interest in the different sub-fields has shifted, resulting in a variety of approaches and techniques. Specifically, with respect to KRR, different paradigms (e.g., rules, constraints, ontologies, processes, etc.) have been the subject of attention, resulting in a variety of knowledge representation languages with different expressivity and underlying logic formalisms, with different trade-offs between expressivity, specificity, and tractability. Adoption, then, has been largely influenced by the availability of tooling, from editing to reasoning, both proprietary and open source. Despite the great success of some of these tools and increasing demand for knowledge graph-based decision support, interoperability has been limited, and even then, hardly goes beyond the scope of the individual languages and applications. For these reasons, although a vast amount of 'knowledge' from a variety of domains has been captured over time, the artifacts (documents, graphs, databases) that carry it vary in languages and formats as well as richness and expressivity, and their combined use is not easily supported except by complex, one-off orchestration of a variety of tools.

This specification addresses these shortcomings by providing a uniform abstraction layer that, from a client perspective, simplifies and normalizes the way KRR artifacts are accessed, manipulated, assembled into rich knowledge graphs, and related systems to which a variety of analytics, reasoning, and rules, can be applied for question answering and computation.

1.3 Scoped Features

Features that are considered in-scope for this specification include:

- APIs for Knowledge Platforms, to be used in the development of knowledge-based applications.
- Semantics of the operations exposed by means of the APIs, including decomposition of the operations into simpler actions.
- Definition of 'Knowledge Base', 'Knowledge Resource' and related concepts.
- Definition of 'Knowledge Platform' in terms of the functional roles of its major components.
- Information models realizing descriptions ('metadata') of knowledge resources minimally viable for knowledge management and delivery, including vocabularies, in the form of ontologies expressed in the W3C Web Ontology Language (OWL), to designate knowledge representation languages/notations and related concepts.

Features that are considered beyond the scope of this specification include:

- Actual specification of KRR languages, and mappings thereof.
- Algorithms for knowledge-based reasoning.
- Implementation of knowledge platform components.
- Development of adapters for candidate components that do not implement the API4KP directly.
- Knowledge-based applications.
- Individual Knowledge Artifacts, including Domain Models, Knowledge Graphs, and Knowledge Bases.

2 Conformance

API4KP defines conformance based on the following orthogonal criteria:

1. **Completeness**
   As the specification defines several modules with operations grouped into component services, implementations are allowed to cover all (and only) those modules that are relevant to the environment they are deployed in.

2. **Coverage**
   The variety of knowledge representation languages and formats for which capabilities are exposed may vary. Some environments may focus on a single language (*e.g.,* OWL), while heterogeneous environments may support operations across a broader variety of notations.

Completeness and accuracy are combined to define levels of conformance for each API4KP module, across the modules, as specified in Tables 2.1-2.5.

Table 2- 1: API4KP Conformance Levels - Information Exchange

Level 0	Implementations are able to exchange knowledge artifacts, including surrogates
Level 1	The operations exchange knowledge artifacts using the API4KP standard data structures, using terminology from the API4KP standard vocabularies.
Level 2	The implementation supports the API4KP knowledge surrogate and knowledge carrier concepts for the exchange of meta-knowledge.

Table 2- 2: API4KP Conformance Levels - Repository Service

Conformance Level 0	PIM Compliance: Equivalent functionality is provided ‘as a service’, but the actual interfaces do not match any PSMs that can be mapped to the specification.
Conformance Level 1	PSM Compliance: Functionality is provided through a PSM implementation of the standard PIM
Conformance Level 2	Full Compliance: All operations are supported.

Table 2- 3: API4KP Conformance Levels – Transrepresentation Service

Conformance Level 0	PIM Compliance: Equivalent functionality is provided 'as a service', but the actual interfaces do not match any PSMs that can be mapped to the specification.
Conformance Level 1	PSM Compliance: Functionality is provided through a PSM implementation of the standard PIM.
Conformance Level 2	Full Compliance: All operations are supported.

Table 2- 4: API4KP Conformance Levels - Knowledge Base Service

Conformance Level 0	PIM Compliance: Equivalent functionality is provided 'as a service', but the actual interfaces do not match any PSMs that can be mapped to the specification.
Conformance Level 1	PSM Compliance: Functionality is provided through a PSM implementation of the standard PIM.
Conformance Level 2	Full Compliance: All operations are supported.

Table 2- 5: API4KP Conformance Levels - Reasoning Service

Conformance Level 0	PIM Compliance: Equivalent functionality is provided 'as a service', but the actual interfaces do not match any PSMs that can be mapped to the specification.
Conformance Level 1	PSM Compliance: Functionality is provided through a PSM implementation of the standard PIM.
Conformance Level 2	Full Compliance: All operations are supported.

API4KP Versioning Strategy

Conformance is defined for a specific version of API4KP. Compatibility rules are further defined across versions.

Every version of the API4KP standard is marked according to the release date. The current version, as per this version of this document, is **`20230201`**. URIs associated to the specification, such as Ontology version IRIs, shall use the same date-based version as a version tag. Version agnostic URIs, such as Ontology IRIs, should NOT include the version tag.

Every release of the specification will define an API version, which shall be versioned according to the SemVer standard. Specifically, increments in Major versions will denote breaking changes; increments in Minor versions will denote added functionality; increments in Patch version will denote fixes to the current functionalities.
As of this document, the API version is **`1.0.0-Beta2`**. This approach generally follows the OMG standard recommendation for version management.

Public, stable implementations of the APIs that need to maintain backwards compatibility, or compatibility across versions, should at a minimum use the Major Version number.
As of this document, the Major API version should be 1.0.0.
For example, the base URL for a web implementation should be **`[base_url]/api4kp/1.0.0/`**, while packages should refine **`org.omg.spec.api4kp.1.0.0`**. Implementations that declare, or imply, a specific version of the specification may omit the version tag.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications may not apply.

API4KP-23 – Revise references cited in the revised ontologies

Reference	Description
[Commons]	Commons Ontology Library, v1.0. Available at https://www.omg.org/spec/COMMONS.
[Dublin Core]	DCMI Metadata Terms, Issued 2020-01-02 by the Dublin Core Metadata Initiative. Available at https://www.dublincore.org/specifications/dublin-core/dcmi-terms/.
[DOL]	Distributed Ontology, Model, and Specification Language (DOL™), version 1.0. Available at https://www.omg.org/spec/DOL/.
[ISO 704]	ISO 704:2022 Terminology Work – Principles and Methods
[ISO 1087]	ISO 1087-1:2019 Terminology — Vocabulary — Part 1: Theory and application
[LCC]	Languages, Countries and Codes (LCC), v1.2. Available at https://www.omg.org/spec/LCC/.
[MOF]	Meta Object Facility (MOF™), v2.5.1. Available at https://www.omg.org/spec/MOF/2.5.1/.
[XMI]	XML Metadata Interchange, v2.5.1. Available at https://www.omg.org/spec/XMI/.
[OpenAPI 2]	OpenAPI Specification v2, Available at https://swagger.io/specification/v2/.
[OpenAPI 3]	OpenAPI Specification v3, Available at http://spec.openapis.org/oas/v3.1.0
[OWL 2]	OWL 2 Web Ontology Language Quick Reference Guide (Second Edition), W3C Recommendation 11 December 2012. Available at https://www.w3.org/TR/owl2-quick-reference/.
[RDF Concepts]	RDF 1.1 Concepts and Abstract Syntax. Richard Cyganiak, David Wood and Markus Lanthaler, Editors. W3C Recommendation, 25 February 2014. Latest version is available at https://www.w3.org/TR/rdf11-concepts/.
[RDF Schema]	RDF Schema 1.1. Dan Brickley and R.V. Guha, Editors. W3C Recommendation, 25 February 2014. Latest version is available at https://www.w3.org/TR/rdf-schema/.
[SemVer]	Semantic Versioning. Available at https://semver.org/
[SKOS]	SKOS Simple Knowledge Organization System Reference, W3C Recommendation 18 August 2009. Alistair Miles and Sean Bechhofer, Editors. Available at https://www.w3.org/TR/2009/REC-skos-reference-20090818/.
[UML]	Unified Modeling Language™ (UML®), version 2.5.1. Available at https://www.omg.org/spec/UML/.
[Unicode]	*The Unicode Standard, Version 5.0*, The Unicode Consortium, Addison-Wesley, 2006, as updated from time to time by the publication of new versions. (See https://www.unicode.org/versions/Unicode13.0.0/ for the latest version and additional information on versions of the standard and of the Unicode Character Database).
[UTF-8]	RFC 3629: UTF-8, a transformation format of ISO 10646. F. Yergeau. IETF, November 2003, http://www.ietf.org/rfc/rfc3629.txt
[XML Schema Datatypes]	XML Schema Part 2: Datatypes. W3C Recommendation 28 October 2004. Latest version is available at https://www.w3.org/TR/xmlschema-2/.

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply. A proper formalization of the concepts evoked by these terms, together with additional definitions, elucidations examples, and related concepts is provided in

the normative ontologies. This specification also depends on the Distributed Ontology, Model, and Specification Language (DOL™), including terminology defined therein and, in particular, terms and definitions that are formalized in the DOL ontology.

Activity −Intentional process, executed with the active participation of one of more agents that carry out a plan.

Agent − Entity that has the capability (potential) to initiate or participate in an activity.

Concept − Atomic (non-decomposable) unit of knowledge created by a unique combination of characteristics.

Environment − Mathematical structure of mappings and members, where the domain and codomains of the mappings are members of the environment.

Idempotency − Property of an operation such that it will yield no additional result (and/or side effect) if it is executed more than once using the same information as inputs.

Immutable Entity − Entity whose state does not, or cannot, change over time without preserving the identity of the entity.

Information Asset − Knowledge asset used by agents to acquire, represent, organize, exchange, store, retrieve and distribute data, about some domain of interest, using structured formats.

Knowledge − Cognition (know-what), pragmatics (know-how) and understanding (know-why) about the nature and/or behavior of something that, when internalized by an agent, has the potential of generating actions in situations that the knowledge applies to.

Knowledge Artifact − Digital or physical object that is specifically constructed to carry one or more (expressions of) knowledge assets.

Knowledge Asset − Work of knowledge that is a knowledge resource considered valuable by a party.

Knowledge Carrier – Role of a physical or digital object (Artifact) that carries a knowledge asset.

Knowledge Expression − Expression of a piece of knowledge in some language, i.e., using a combination of signs and symbols that conform to the rules of the grammar of that language.

Knowledge Fragment − Proper part of a knowledge expression that is not the realization of a knowledge asset itself.

Knowledge Manifestation − Concept that abstracts a specific class of knowledge artifacts, defining the common qualities of its members and their content.

Knowledge Platform − Computing environment designed to host reasoners and consume knowledge artifacts.

Knowledge Representation and Reasoning Language – Machine-executable language used in knowledge expressions to express works of knowledge.

Knowledge Resource − Immutable, identifiable, versionable entity that is, expresses or carries some piece of knowledge.

Mutable Entity − Continuant entity whose state, determined by the configuration of its qualia, changes over time while maintaining a principle of identity.

Note that for the purposes of this specification, knowledge resources are considered immutable. This is important with respect to the operations on knowledge graphs and/or knowledge bases specified herein to preserve idempotency.

5 Symbols

API	Application Programming Interface
AST	Abstract Syntax Tree
BPMN	Business Process Modeling Notation
CMMN	Case Management Modeling Notation
DMN	Decision Modeling Notation
DOL	Distributed Ontology Language
FRBR	Functional Requirements for Bibliographic Records
KA	Knowledge Artifact
KB	Knowledge Base
KBS	Knowledge Based System
KR	Knowledge Resource
KP	Knowledge Platform
KRR	Knowledge Representation & Reasoning
MIREOT	Minimum Information to Reference an External Ontology Term
RDF	Resource Description Framework
ReST	Representational State Transfer
OWL	Web Ontology Language
SKOS	Simple Knowledge Organization System
TTL	Terse Triple Notation
WoK	Work of Knowledge
XML	eXtensible Markup Language

6 Additional Information

6.1 Acknowledgements

The submission team for this specification includes 88 Solutions and Thematix Partners. Major contributors include: angos.ai, Fraunhofer FOKUS, Mayo Clinic, and Micro Focus, with significant support from the Freie Universität Berlin, Universität Leipzig, and Raytheon Technologies.

Primary Contacts:

Elisa Kendall
Thematix Partners LLC
ekendall@thematix.com

Manfred Koethe
88 Solutions
manfred@88solutions.com

Additional Contributor and Supporter Contacts:

Pete Rivett
agnos.ai UK Ltd
pete.rivett@agnos.ai

Adrian Paschke
Fraunhofer FOKUS and Freie Universität Berlin
adrian.paschke@fokus.fraunhofer.de

Ralph Schäfermeier
Fraunhofer FOKUS and Universität Leipzig
ralph.schafermeier@gmail.com

Davide Sottara
Mayo Clinic
sottara.davide@mayo.edu

John Draga
Micro Focus
John.Draga@microfocus.com

Cory Casanave
Model Driven Solutions
cory-c@modeldriven.com

Roy Bell
Raytheon Technologies
Roy_M_Bell@raytheon.com

The submission team would also like to thank Tara Athan (Athan Services), colleagues at Arizona State University, and Cognitive Medical Systems for their contributions over the last several years.

This page intentionally left blank.

7 Application Programming Interfaces for Knowledge Platforms

7.1 Overview

The purpose of the API4KP interfaces is to expose the API4KP operations, as functionalities provided ‘as a Service’ by one or more software components of a Knowledge Platform.

The APIs follow a number of principles, which are reflected in the signature of the operations. The principles are also intended to facilitate implementations across multiple paradigms - in particular local Virtual Machine vs. REST vs. SOAP. The well-known mappings between general API and web-service oriented APIs will be used to support non distributed applications in a consistent and predictable way.

- **Resource Orientation** − For each operation, the verbal definition is equivalently specified in terms of a nominal definition. For example, ‘translate’ (operation - verb) is the execution of a ‘translation’ (operation - name) by a ‘translator’ (server - entity).

  As a consequence of this approach, servers, and capabilities thereof, are represented by an explicit object (resource in the ReST sense) which can be used for the discovery and configuration of knowledge platforms.
  ‘Capability’ resources materialize the concept of Task as a class of operations that an actor, as represented by a ‘Server’ object, can perform. Capability resources describe internal objects that can instantiate ‘Action’ (Task instances) objects which, when applied to appropriate inputs including (carriers of) Knowledge Resources, result in Operations being executed.

- **Strategy Orientation** − A second consequence of the resource-oriented approach is that each capability, in its nominal definition, can also be materialized by an object, which acts both as a resource for further discovery/configuration, as an uniquely identifiable and classifiable invocation target, and as a wrapper of the actual implementation of an operation, following the principles of the Strategy pattern.

  This approach also favors the decomposition of the implementation into indexable sub-components, e.g., to handle different languages in different ways. These objects materialize the API4KP notion of ‘Method’.

  - **(Optional) Configurability**
    Most implementation strategies are expected to provide a ‘default’ behavior, but a Client should be able to configure the behavior by passing additional parameters which depend on the class of the agent (e.g., DL Reasoner) and/or the strategy being implemented (e.g., tableaux-based classification)

- **ReST-fulness** – Operations follow principles and best practices for distributed APIs such as statelessness and idempotence. ‘getX’ and ‘listXs’ operations follow GET semantics; ‘setX’ operations follow PUT semantics; ‘addX’ and ‘createX’ operations follow with POST semantics; ‘isX’ follow HEAD semantics; ‘deleteX’, to align with DELETE semantics, are not aimed at removing resources, but rather to ensure that resources are not present.

- **Support for Provenance** − Since the outcome of most operations depends heavily on the ability of a given server and strategy to handle different languages, a server is expected (but not required) to provide an ‘Explanation’ of how it produced the actual result.

- **Semantic Web Orientation** − Individual entities in the API4KP conceptual space are expected to be denoted using URIs, including both version-agnostic and version-specific URIs.

- **Functional Programming Orientation** − The APIs expose services that can be formalized using concepts from the functional programming paradigm.
  - ***Monads*** allow to bind pure functions to generic data structures, by providing a computational context that handles the application of a function across different data types.
  - ***Functors*** (reified functions) represent units of behavior comprised by the API4KP specification.
  - ***Composition*** is the mechanism by which an Operation, exposed through an API, is defined in terms of atomic functions.
  - ***Application*** is the act of using a function on some data (as opposed to passing the data as arguments to the function)
  - ***Currying*** The incremental application of a function to multiple arguments, in terms of composition of multiple unary functions.
    Example :
    `f( X, Y ) = g o h`
    where:
    `h : X → g`$_X$`( Y ) = g( Y | X )`
    `g : Y → v`$_{Y,X}$`( ) = v( | X, Y )`
    `f( X, Y ) = eval ( v`$_{Y,X}$ `)`

    That is, `h` applied to `X` returns a unary function object that, when applied to `Y`, returns a nullary function object. When evaluated (lazily), this last object returns the same value as the (eager) evaluation of `f` on the same arguments `X` and `Y`. Currying generalizes to an arbitrary number of arguments by induction.

7.2 Architectural Styles and Approach

Software communication requires a message transport, a message protocol, and a destination address. A message transport plays the role of carrier - in this case, a physical means to convey a message. Examples of transport mechanisms include Ethernet, shared memory, and the Peripheral Component Interconnect (PCI) bus. Wi-Fi and Bluetooth are examples of wireless transports. All three concepts of transport, protocol, and destination address are interrelated. A change to one affects the others. For example: if the source and destination happen to be in the same virtual address space; the source will know the virtual address of the receiving function and will know how to invoke it and how to pass a message through parameters. Nearly all CORBA implementations automatically recognize when this optimization can be used. A simple function call is not possible when the source and destination are physically separated. Distributed software communication is accomplished by serializing the message according to a protocol that destination understands. A remote destination addresses could be something simple like the combination of an IP address and port number or it could be expressed as a URL, or an IRI, or a CORBA Interoperable Object Reference (IOR). All physical message transports including Ethernet and Wi-Fi just transport bytes. It is left up to the source and destination to agree on a message protocol. The protocol specifies the set of rules for how these messages will be serialized for transport. A protocol could be ad-hoc and unsophisticated or it might have the sophistication necessary for the source and destination to remain oblivious of the distributed nature of the system. This in turn allows the developer to choose the style that best fits the application.

The API4KP PIM is designed to support, at a minimum, the following bindings:

1. (Java) Interfaces
2. SOAP (WSDL)
3. ReST
4. OMG IDL

Both CORBA and Java RMI serialize messages in compliance with the Internet Inter-ORB Protocol (IIOP) standard. This is a comprehensive standard, which is usually implemented with support from automated tools. On the other hand,

web services only specify the use of XML and leaves up to the software developer(s) to ensure that the source and the destination have the same understanding of the XML content. There are many tools to support this understanding. WSDL specifies that messages will comply to an XML schema. If WSDL is not being used and no XML schema has been specified; the source and destination must at least have some sort of mutual understanding of an implied schema or else they run the risk that parts of their messages will be lost or misunderstood. In this context, ReST should be interpreted as an architectural style based on Resource orientation and functional interactions, generalizing the common notion of ReST-on-HTTP used for web services. As an architectural paradigm, it is not incompatible with object/operation-oriented architectures. In fact, a ReSTful approach is *primarily* compatible with this functional approach to knowledge bases and reasoning.

7.2.1 Cross-Architecture Alignment

The problem of *deterministically* aligning the different styles with a common abstract PIM is addressed using a fully model-driven approach. The architecture metamodel is partitioned in three main components, and *Translation*, *Projection* and *Structuring* operations are used to generate a variety of PIM, PSM and implementation artifacts. Given a distinction between:

- The *specification of the operations*, representing the behavioral component of the architecture, and driving the APIs.
- The *modeling of the datatypes*, exchanged through the operations, representing the structural component of the architecture, thus driving the ReST resource types.
- The *definition of the controlled terms*, which provide the semantic component of the architecture through a binding to a common set of API4KP ontologies.

Different representation languages, specific to each asset type, are used to define the primary artifacts:

- OpenAPI v2 (Swagger) was used as the basis for the initial version of this specification for the operations. While mainly used for ReST architectures, OpenAPI specs are designed to be mapped to object-oriented languages (e.g., Java), and thus can be reverse engineered into operation-oriented interfaces, expressed using IDL.
  OpenAPI v3 specifications are included herein, derived from the OpenAPI v2 specification by means of a functional upgrade.
- UML Class models are used for the datatypes. The class models deliberately use minimal capabilities (classes, attributes, unidirectional associations, limited inheritance, no cycles) in order to enable mapping to tree-oriented models such as XML and JSON schemas.
- The semantic elements are defined in OWL. Within a specific implementation of a Knowledge Platform, it is recommended that ontologies are first MIREOT-ed to derive SKOS vocabularies, then flattened into enumerations that can be bound to specific attributes/elements of the UML/XSD models. Acknowledging that there may be alternative approaches, it is expected that the mappings preserve traceability to the URI of the original ontology entity.

Notice that the vocabularies are late-bound to the schemas, and the schemas are late-bound to the operations. This approach has been adopted because the native modeling capabilities of OpenAPI are mainly designed for information interchange rather than processing or, to an even lesser extent, modeling. OpenAPI adopts a limited version of JSON schema which provides a very limited variety of datatypes (e.g., no distinction between String and URI). For similar reasons, neither XSD nor OpenAPI (nor UML) provide a concise way to formalize concepts and their relationships. In general, more expressive languages have been used for the different parts of this API4KP specification, effectively making the specification itself a composite knowledge artifact in the API4KP sense.

Figure *1* provides an overview of the metamodeling environment, described using the API4KP/DOL concepts of Asset, Language, Profile, Serialization, Format, and mapping thereof.

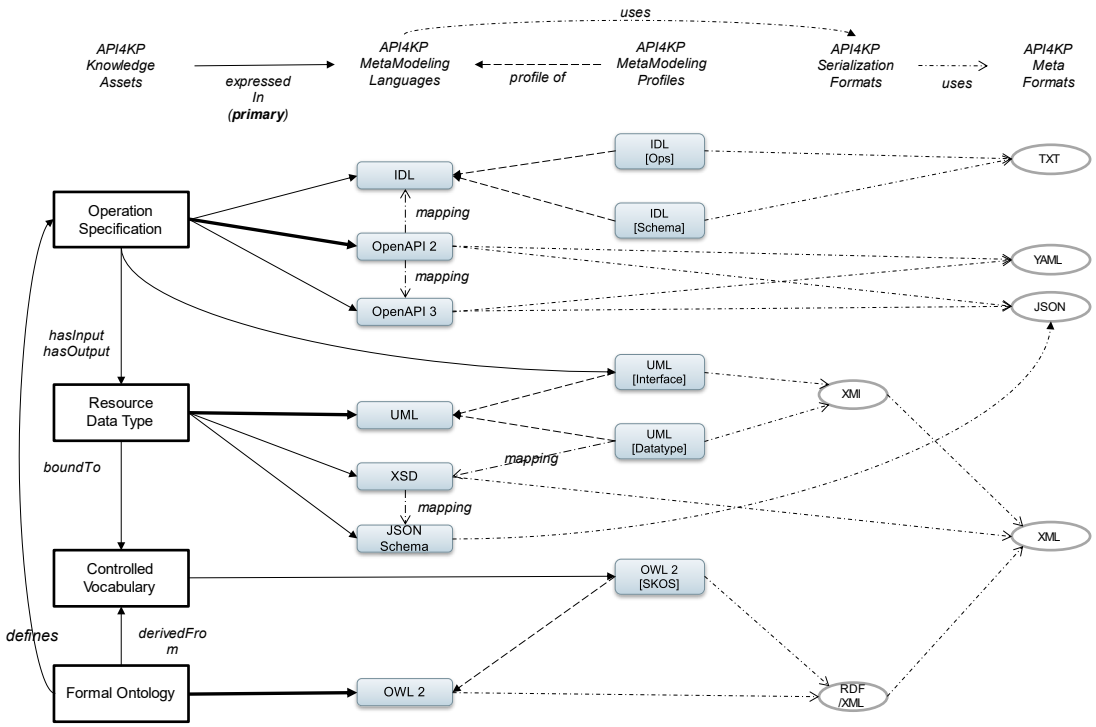

Figure 1: API Metamodeling Environment

Figure *2*, then, provides a view of how the components of API4KP are integrated with and generated from one another, as described above, up to a platform-specific implementation. This chain(s) of derivations and integrations, as noted operations that can themselves be defined using API4KP semantics, is compatible with the API4KP environment, but not the only possible one.

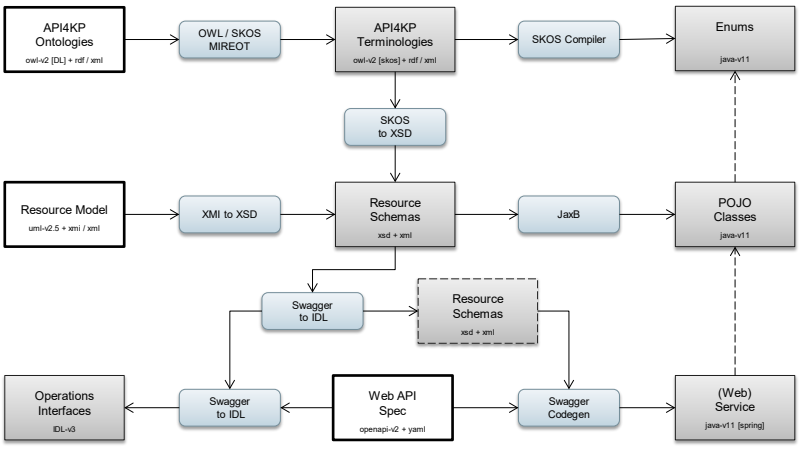

Figure 2: API Construction/Generation Overview

7.2.2 Proxy-Oriented Approach

Despite their prominence, object-orientation and web services are not the only possible communication paradigms. Additional alternatives, distinguished according to the degree of coupling between the client and the server, are described in Annex D. API4KP takes a proxy-oriented approach, and generalizes the OpenAPI-based code generation framework, based on the architectural pattern shown in Figure *3*.

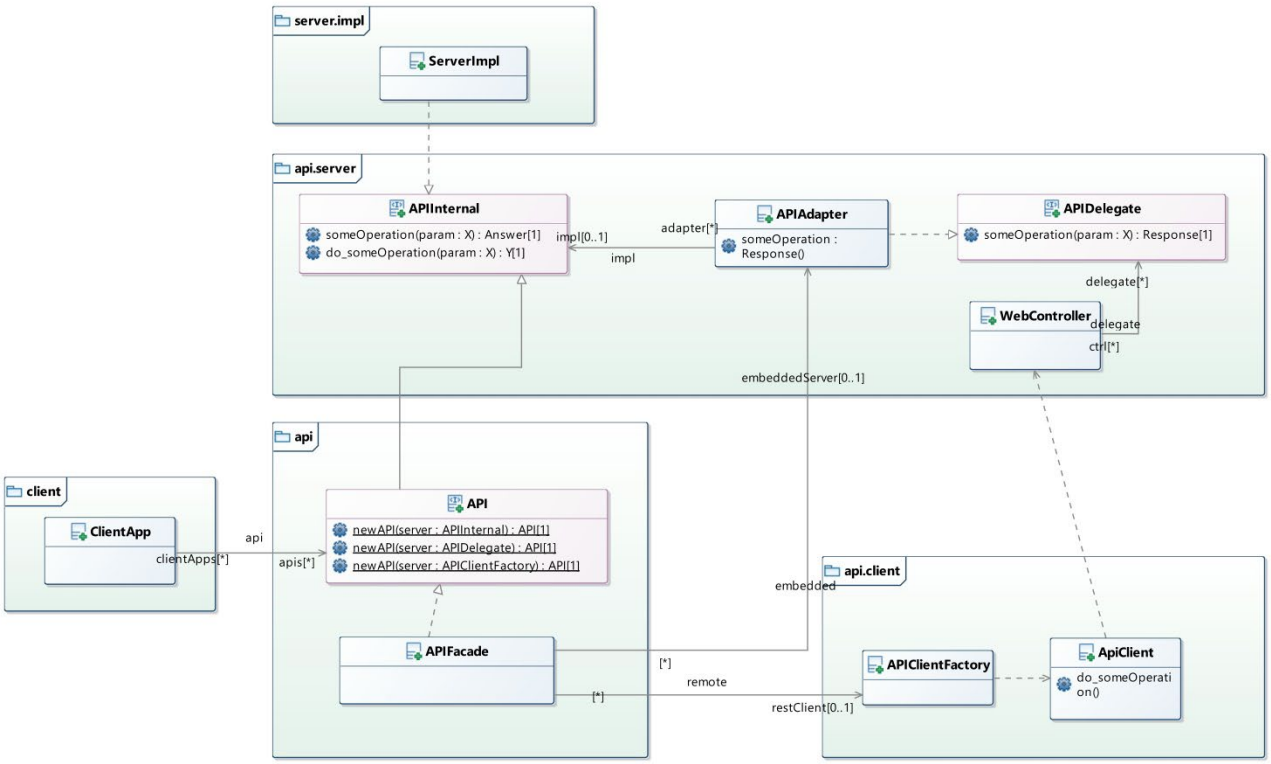

Figure 3: Proxy Pattern for Code Generation

Given that OpenAPI is primarily a web service framework, APIClient and APIDelegate are client/server stub/skeletons that handle the (web) transport: the client application interacts with the APIClient, and the server is an implementation of the interface APIDelegate. As the server implements the operations, it is necessary to wrap the results in a Response – a data structure that not only contains the result, but also additional metadata such as HTTP response codes and headers. While useful, the metadata is web-specific and overfits only one of the API4KP integration patterns. On the other hand, the client application is presented with a plain operation with a core return type Y: if necessary, the metadata can be queried explicitly from the APIClient. This approach does not fit the monadic approach of API4KP.

In the approach taken herein, the Server is expected to implement an APIInternal interface, which allows the server to either return the result directly – improving compatibility with servers not designed for web integration – or, alternatively, to wrap the result in a monadic, rather than web-oriented, wrapper Answer. A Client is presented with an API interface that extends APIInternal directly, supporting the direct, by-reference integration pattern. Alternatively, and transparently, *APIFacade* implements API, but delegates to APIClient, which in turn delegates to a web server, supporting a web-mediated interaction. The role of the Façade is to extract the information from APIClient, and package it into an Answer. In order to expose Answer-enabled servers as webservers, a further *ApiAdapter* is needed. This adapter transforms an Answer into a Response, which is then transformed back into an Answer client-side.
This capability also allows to deploy a server designed for the web, i.e., one that implements APIDelegate natively or through an APIAdapter – as an embedded client-side component.

7.2.3 Common Datatypes

The APIs use a limited number of data structures across input and/or outputs, which can be divided into basic value types, reusable Data Structures and (ReST-enabled) Resources.

7.2.3.1 Primitive Datatypes

API4KP uses a limited number of primitive datatypes. In addition to the basic UML String, Integer and Boolean. additional datatypes are defined as follows:

- **DateTime –** The representation of an atomic point in time, with arbitrary granularity and optional time zone, as defined in the standard ISO 8601, and mapped to the W3C XML Schema 'dateTime' datatype.

- **URI / URL –** The unique identifier of a resource on the web, or address thereof, as specified in https://www.ietf.org/rfc/rfc3305 and predecessors.

- **UUID** − A universal ID, conforming to the structure specified in https://tools.ietf.org/html/rfc4122

- **Bindings** − A Collection of Entries. An Entry is a key/value pair, where the key is a locally unique identifier, associated to a value of Any type. Bindings are primarily intended to capture variable assignments, such as may result from the execution of a Query. In this context, values can either be NULL (free/unassigned variable), or immutable (bound variable).

- **Any** − Marker datatype used to describe values that have an undefined, unstructured and/or unconstrained form. This datatype is mapped to the W3C XML Schema 'anyType'.

7.2.3.2 Signs, Identifiers, Terms and Descriptors

An Identifier is a symbol used to *identify* one and one entity[1] within some context. Depending on the context – e.g., universal vs local, web vs internal – it may convenient, or even necessary, for the identifier to have a specific form.

In API4KP, URIs and UUIDs are considered the primary, general purpose form of identifiers. URI should be preferred when the identifier is intended to be persistent and/or dereferenceable. UUIDs are primarily used within the scope of API-mediated interactions because they do not require a central authority to be minted and/or assigned. UUIDs and other forms of identifiers with internal structure such as OIDs and DIDs can also be used for more permanent entities, as long as they are mapped to their canonical URI form.
API4KP also defines a number of Structured Identifiers, as a way to uniquely denote Resources of interest, while providing a minimal descriptive context. The Resources of Interests include formal Knowledge Resources as managed by a Knowledge Platform, Semantic entities from a Business Domain of reference ("Concepts"), and versions and representations thereof.

[1] Conversely, an entity can have multiple identifiers

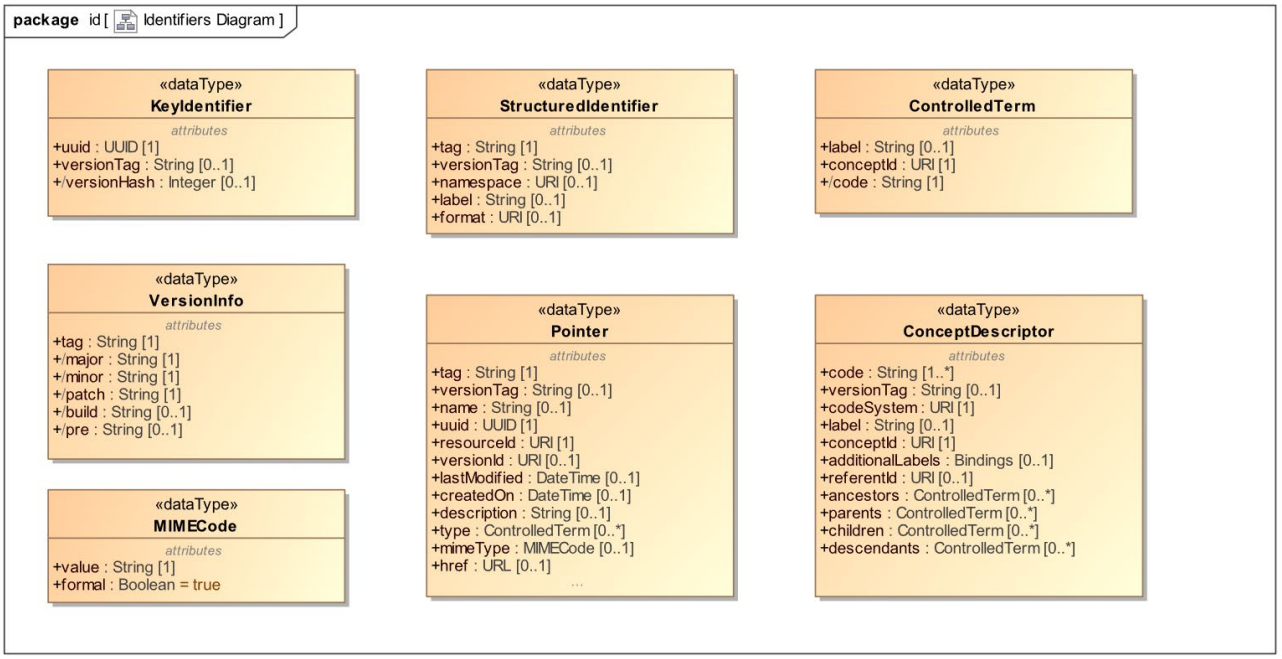

Figure 4: Identifier and Resource Identifier Descriptions

ConceptDescriptor

A Structured Designator that supports the bridging of Concepts, as atomic fragments of Semantic Knowledge Resources (e Ontologies, Concept Schemes, Vocabularies) and Terms, syntactic representations used in other forms of Knowledge Resources and/or the APIs used to process them.

ConceptDescriptor is influenced by the W3C Ontolex model and cover all three dimensions of the semiotic triangle. In particular, Concepts are universal individuals (in the SKOS sense) which are evoked by Terms and are the intensional counterpart of their extensional Referent Entities – classes, relationships and/or known individuals. More specifically, Concepts can be organized in Concept Schemes, which usually provide a codification system and a 'broader/narrower' relationship, and/or (formally) defined in one or more Ontologies.
As a datatype, ConceptDescriptor allows to carry references to other Concepts in the neighborhood of the given Concept, which can be used to support various reasoning tasks.

Remark: since Concepts are considered fragments of a Semantic Knowledge Resource, they should not have a version that is independent from the version of the scoping Resource.

Attribute	Description
code	The primary, local identifier of the Concept, within the scope of the defining Semantic Knowledge Resource. May or may not coincide with the local part of the conceptId.
conceptId	A universal identifier associated to the Concept which, by definition, is not scoped by the defining Semantic Knowledge Resource. Concept Ids are usually version agnostic.
codeSystem	The universal identifier of a Concept Scheme, where this Concept has been scoped. If present, the referenced Scheme should be the context which assigned the code and/or the conceptId. When possible, this URI should be version specific.
versionTag	An optional version identifier associated to the specific Concept. If present, should be consistent with the version of the scoping Resource, which should be inferable from the respective URI.
referentId	The URI of the denoted Referent entity, possibly dereferenceable to, and resolvable within a formal Ontology.
label	The (primary) term used to evoke this Concept, in the context of use of this ConceptDescriptor.

additionalLabels	Additional terms associated to this Concept, as key/value pairs where the the key denotes the type/role of the label. The key should be derived from an annotation property (e.g. skos:altLabel), or a language code (e.g. 'us-en').
parents	References to other Concepts $P_{j=0..N}$ in the same Scheme, such that this Concept 'has broader' P_j
children	References to other Concepts $C_{j=0..N}$ in the same Scheme, such C_j 'has broader' this Concept
ancestors	The closure of the 'has broader' relationships in the context of the defining Scheme
descendants	The closure of the inverse 'has broader' relationship in the context of the defining Scheme

ControlledTerm

A simple Structured Designator that can be used to denote an entity, and/or evoke a Concept. Combines a formal identifier with a (contextual) human readable label.

Attribute	Description
conceptId	The universal identifier of either an evoked Concept, or a Referent entity. Given that either choice has different formal properties, the choice should be consistent with the context of use of the ControlledTerm
code	A local identifier associated to the denoted/evoked entity.
label	The (primary) term used to evoke this Concept in its context of use

KeyIdentifier

Structured Identifier that identifies a specific version a Knowledge Resource. A KeyIdentifier is designed for internal use by API4KP services, to index and retrieve efficiently resources, including KnowledgeBases and components thereof.
KeyIdentifer can be considered the minimal counterpart of a Pointer, from which it can be derived.

Attribute	Description
uuid	The universal identifier associated with the resource, across its versions
versionTag	The component of the identifier associated to the specific version of the resource
/versionHash	A compact binary encoding of the versionTag, based on a hash function

MIMECode

A designator of the characteristics of the manifestation of a Knowledge Resource. In API4KP, (generalized) MIME codes are terms that denote Media Types (https://www.iana.org/assignments/media-types/media-types.xhtml). More specifically, MIME "types" are associated to the categories of Knowledge Artifacts such as text or software, while the "subtypes" are correlated with the characteristics of the Knowledge Expression carried by a Knowledge Artifact. Moreover, A MIME Code is considered *formal* if its sub-type code can be parsed, and used to denote unambiguously the syntactic components of an Expression (language, profile, serialization, format, lexicon, alphabet and encoding). Informal MIME codes are considered pre-coordinated, and their interpretation is predetermined.

Attribute	Description
value	The string-based representation of the MIME type code
formal	If true, denotes a post-coordinated (parsable) MIME code

Pointer

Structured Designator that identifies a specific version a Knowledge Resource, while providing a minimal description of the Resource itself.

Attribute	Description
tag	A version-agnostic Identifier associated to the Resource. Tags are not required to be universal identifiers.
versionTag	An identifier of the specific version of the denoted Resource, to be used in combination with the tag and/or uuid.
uuid	A version-agnostic, universal Identifier associated to the Resource, in the form of a UUID.
name	A human readable, informative name that designates the Resource.
resourceId	A version-agnostic URI that identifies the Resource – also known as "series" Identifier.
versionId	A version-specific URI that identifies the Resource.
description	A human readable, informative, contextual description of the Resource.
lastModified	The date/time of the creation of this version of the denoted Resource.
createdOn	The date/time associated to the creation of the first, original version of the denoted Resource.
type	One or more ControlledTerms that denote classifiers (e.g., OWL Classes) that apply to this Resource (e.g., such that the denoted Resource can be considered an instance of).
mimeType	A pre-coordinated basic descriptor of the representational characteristics of the denoted Resource. When the Pointer denotes a Knowledge Asset, the mimeType can be omitted, or be used to denote a canonical representation of the Asset.
href	A URL where the Resource can be accessed.

StructuredIdentifier

Structured Identifier that identifies an Entity as it is known to the Business *Clients* of a Knowledge Platform. Identifiers of this kind should be generally considered "metadata" by the API4KP services.

Attribute	Description
tag	The textual form of the business identifier associated to the denoted Resource.
versionTag	The identifier of the specific version of the denoted Resource.
namespace	The URI of the namespace that scopes this identifier, as a proxy for the identification authority that assigns and/or allows to dereference the identifiers.
label	A human readable name associated to the denoted Resource.
format	A URI that denotes the grammatical rules that the tags should conform to, e.g., to distinguish OIDs from DIDs.

VersionInfo

VersionInfo are post-coordinated structures used to parse and process the version-specific part of the Identifier of a particular version of a Knowledge Resource, usually referenced as a 'versionTag'.
Since API4KP recommends the use of semantic versioning, the structure is borrowed directly from the SemVer 2.0 specification, and its use should be consistent with that specification.
Other strategies such as Calendar-based versioning should be aligned, for example using Year/Month/Day as Major/Minor/Patch components.

Attribute	Description
tag	The pre-coordinated version tag
major	The major component of a structured version tag
minor	The minor component of a structured version tag
patch	The patch component of a structured version tag
build	The build component of a structured version tag
pre	The pre-release component of a structured version tag

7.2.3.2.1 Correlation Between Identifiers

Implementations should, where possible, correlate the elements of structured designators such as Pointer and ConceptDescriptor. An entity should have a primary, universal URI, and optionally a version URI. These URIs should be decomposable into their namespace, tag and versionTag components. When appropriate, UUIDs should be derived functionally from the primary and/or version URIs, according to the UUIDs v5 methodology.

Example:
A Pointer with a decomposable URI and a versionTag:

```
resourceId: https://ckm.m.e/a/{tag}
versionTag: {versionTag}

// derived
versionId: https://ckm.m.e/a/{tag}/versions/{versionTag}
tag: {tag}
uuid: isUUID(tag) ? { tag } : UUID.v5from( {resourceId} )
```

Example:
A Pointer to a Resource natively identified by a UUID (with an optional versionTag):

```
uuid : {uuid}
versionTag: {versionTag}

// derived
resourceId: urn:uuid:{uuid}
versionId:  urn:uuid:{uuid}{versionTag}
tag: {uuid}
```

Example:
A ConceptDescriptor referencing a coding system that differentiates between the system namespace and the entity namespace:

```
resourceId: {entityNs}/{tag}
codeSystem: {systemNs}
tag: {tag}
versionTag (system scope): {systemVersionTag}

// derived

uuid: isUUID(tag) ? { tag } : UUID.v5from( {resourceId} )
```

7.2.4 (Canonical) Knowledge Surrogates

A Knowledge Surrogate is a ‘metadata’ Knowledge Artifact about other Knowledge Resource(s) that carries a relevant subset of syntactic, structural, and semantic information that is relevant to some API4KP operation. There are numerous approaches to ‘Metadata’ models in and for Knowledge Management. The APIs formalize the notion of metadata record as ‘Knowledge Surrogate’, which is, and thus is processed as, a kind of Knowledge Artifact. The ‘canonical’ API4KP Surrogate model is a Schema (thus a kind of Knowledge Representation Language) which is mappable to other metadata models, and transrepresentable to/from those models by means of API4KP operations. The canonical model ensures that, whatever metadata model is natively adopted by a particular organization, the metadata that supports different API4KP operations is isolated and can be exchanged/exposed in a predictable way.

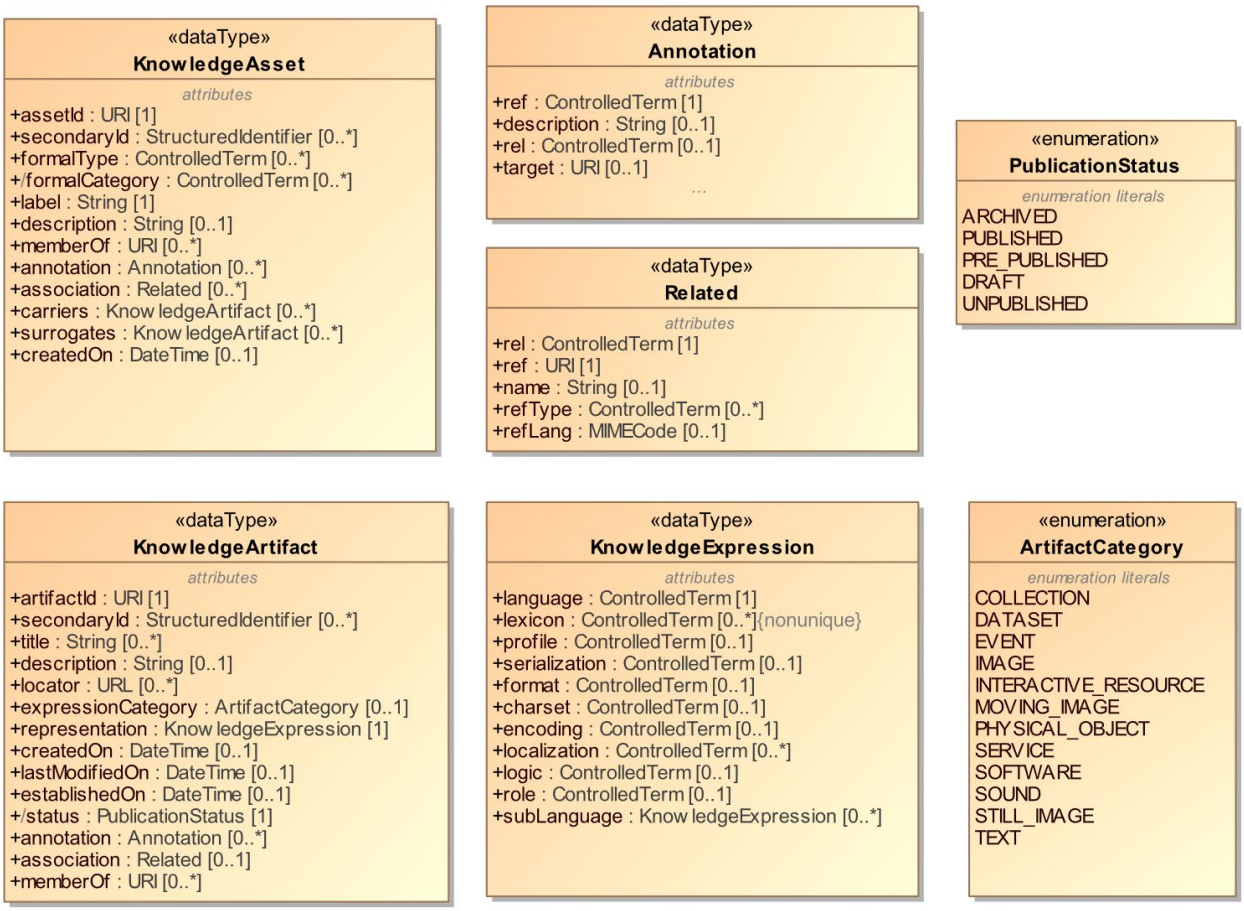

Figure 5: Knowledge Assets, Artifacts and Expressions

Figure *5* provides a view of knowledge assets and associated artifacts as used in API4KP. It is important to consider that, strictly speaking, the Surrogates are Representations of, but distinct from, the actual Knowledge Resources. Knowledge Assets are immaterial until expressed in some Language: a Surrogate is the only way to acquire some information without access to a representation of that knowledge itself. Knowledge Artifacts are documents, files and (in a sense beyond the scope of API4KP) even physical objects such as books and diagrams that the Knowledge Artifact Surrogate is a proxy for.

7.2.4.1 Knowledge Asset (Surrogate)

A Surrogate that focuses on the Knowledge Asset, i.e., the knowledge content of a Knowledge Resource, regardless of the availability of any representation of that Knowledge.

Attribute	Description
assetId	The version-specific URI that identifies the version of the Knowledge Asset described by this Surrogate. Should be decomposable into a KeyIdentifier.
secondaryId	Zero or more business identifiers associated to the Knowledge Asset.
formalType	Controlled Term that denotes a subclass of api4kp:KnowledgeAsset, according to a classification that is based on, or implies, the formal semantics of the work of Knowledge. Example: dol:Ontology, as in a logic-based axiomatic theory.
formalCategory	Controlled Term that denotes a generalized classifier that classifies the Knowledge Asset.
label	A human readable name, possibly contextual, associated to the Knowledge Asset.

description	A human readable description of the Knowledge Asset. Descriptions are informative and for human consumptions: they should not be used for Knowledge Processing and should not be confused with Knowledge Expressions that use some Natural Language representation.
memberOf	A reference to one or more collections that this Knowledge Asset if member of. Note that aggregates of Knowledge Assets should not be confused with Set-oriented Composite Knowledge Assets.
annotation	Structured Annotations used to describe Asset/Concept relationships (see section 7.2.5.1).
association	Structured Links used to describe Asset/Asset relationships (see section 7.2.5.2).
carriers	The association between a Knowledge Asset (Surrogate) and its Knowledge Artifact (Surrogates), to reflect the association between the described entities.
surrogates	The association between a Knowledge Asset (Surrogate) and other Surrogates of the same Asset, possibly including this Surrogate ("self").
createdOn	A dateTime that reflects the moment when the Knowledge Asset was first created, to a sufficient degree of precision.

7.2.4.2 Knowledge Artifact (Surrogate)

A Surrogate that focuses on a Knowledge Artifact, i.e., any one of the concrete manifestations of a Knowledge Asset, including Assets that are Descriptions of other Assets.

Attribute	**Description**
artifactId	The version-specific URI that identifies the version of the Knowledge Artifact described by this Surrogate. Should be decomposable into a KeyIdentifier.
secondaryId	Zero or more business identifiers associated to the Knowledge Artifact.
title	Human readable, often official, designations associated to the Knowledge Artifact.
description	A human readable description of the Knowledge Artifact. Could be used as an informative summary of the Artifact content for human consumption but should not be used for processing.
memberOf	A reference to one or more collections that the denoted Knowledge Artifact is member of.
locator	Any URL where (copies of) the Knowledge Artifact can be acquired.
association	Structured Links used to describe Asset/Asset relationships (see section 7.2.5.2).
annotation	Structured Annotations used to describe Asset/Concept relationships (see section 7.2.5.1).
status	Publication Status of the described Knowledge Artifact.
createdOn	A dateTime that reflects the moment when the Knowledge Artifact was first created, up to a sufficient degree of precision.
lastModifiedOn	A dateTime that reflects the moment when the given version of the Knowledge Artifact was created, up to a sufficient degree of precision.
establishedOn	A dateTime that reflects the moment when the given version of the Knowledge Artifact was published to a Knowledge Platform, up to a sufficient degree of precision.
expressionCategory	A classification of the material/digital form of this Knowledge Artifact, based on the DCMI (Artifact) Types taxonomy.
representation	The description of the syntactic characteristics of the Knowledge Expression carried by the described Artifact.

It is important to remark that publication statuses are *derived.* Instances of an API4KP KnowledgeArtifact describe specific and immutable versions of an actual Knowledge Artifact Resource. As the Artifact evolves in terms of quality and maturity, different versions should be established, and the version tag should reflect the publication status.
In particular:

- Unpublished Artifacts do not "exist" as Resources, and thus do not have a stable version nor a status.
- Draft Artifacts are likely to have "SNAPSHOT" versions, not all of which may be tracked explicitly, which would be associated to a specific timestamp ("lastModifiedOn").
- Pre-Published Artifacts versions (aka "Final Draft" or "Release Candidate") would be tagged with a version tag that denotes the candidacy status.
- Published Artifact versions would have a stable version tag.

Also note that the attribute “establishedOn” allows to differentiate the time an Artifact becomes available to/through a Knowledge Platform, even if the publication process has not been managed through the Knowledge Platform itself.

7.2.4.3 Knowledge Expression (Representation)

A Surrogates that focuses on the aspects of the formal language(s) used to construct a Knowledge Expression, as the representation of a Knowledge Asset and carried by a Knowledge Artifact.
As a whole, the Representation metadata correlates to the minimal required capabilities of a Knowledge Platform that is expected to parse the Artifact content, before it is processed.

Attribute	Description
language	Term that denotes the abstract syntax of the (primary) representation language used in the Expression. Example: OWL2
lexicon	Term that denotes the vocabularies, and implies the Concept Schemes / Ontologies, from which the Terms used in the Expression have been sources Example: SNOMED-CT, FIBO as medical and financial ontologies, respectively, providing terms (denoting concepts) that can be used to construct sentences (axioms) in OWL2
profile	Term that denotes a well-known restriction of the primary language, usually trading expressivity for complexity. Example. OWL2-RL – a simpler but less computationally intensive subset of OWL2
serialization	Term that denotes the concrete syntax used in conjunction with the primary language Example: RDF/XML, Turtle for OWL2 ontologies
format	Term that denotes the meta-format used to define the language’s serialization Example: The RDF/XML serialization of OWL2 is based on XML
charset	Term that denotes the Character Set used to construct the representation of the Terms, assumed to be compatible with the serialization Example: UTF-16, Windows-1252
encoding	Term that denotes any additional re/encoding of the serialized expression Example: “Default” for the given charset; Base64
localization	Term that denotes the natural language(s) used in the non-computational aspects of the Expression
logic	Term that denotes the formalism sufficient and necessary to capture the Expression, possibly associated to the computational complexity of the Expression.
subLanguage	Association between a primary and one or more secondary representations. A secondary representation denotes the language used to construct fragments, which are woven into a primary expression, usually to decorate, complement or supplement the primary expression. Example: fragments of MathML injected into a set of OWL2 axioms
role	Term that classifies a secondary representation, with respect to a scoping primary representation. Example: MathML as a (mathematical) functional expression language, extending OWL2

7.2.5 Knowledge Resource Relationships

Knowledge Resources are related in different ways other than the ‘vertical’ Artifact – Asset stack.. Surrogates can carry those relationships, and linked Surrogates form a special kind of Knowledge Base that is at the core of the Semantic Knowledge Asset Repository APIs.

Relationships can be partitioned in two main categories:

- Knowledge Resource to (Domain) Concept relationships
- Knowledge Resource to Knowledge Resource relationships

7.2.5.1 Resource to Concept Relationship

Knowledge Resource to Domain Concept relationships, also called informally “Annotations”, can be used to highlight focal Concepts that are part of that Resource’s Knowledge Asset for purposes such as searching and querying. Semantically, Annotations should be considered as reified RDF statements (“triples”) where the subject is the annotated Knowledge Resource. Annotations are expected not only to be consistent with the formal semantics of the target ontology, but also with the business domain semantics.

API4KP Annotations are generally aligned with the Web Annotation Data Model, which should be adopted for more complex use cases.

Annotation

Attribute	Description
rel	A ControlledTerm that denotes the ‘property’ that relates the subject Knowledge Resource and the object Concept. If omitted, the Concept should be considered a “semantic tag”.
ref	A ControlledTerm that evokes the Concept related to the Knowledge Resource.
description	An optional, human readable representation of the referenced concept. Enables the preview of the annotation without having to dereference the object Concept.
target	An optional identifier of the Component or Fragment in the source Knowledge Resource that this annotation more specifically applies to. If not specified, should be assumed to coincide with the source Resource itself.

7.2.5.2 Resource to Resource Relationship

Related (Resource) is a Link-like data structure that follows the principles of HATEOAS and is used to establish associations between Knowledge Resources that are managed using the API4KP. A Related instance should be considered a reified Statement that points to a target Resource by means of its Identifier.

Related

Attribute	Description
rel	A ControlledTerm that denotes the specific semantic association between the subject Resource (denoted by the identifier in KnowledgeAsset or KnowledgeArtifact) and the target (“ref”) Knowledge Resource. The denoted relationship should be, or be consistent of, a subproperty of api4kp:associatedTo.
ref	The Identifier of the (version of the) target Knowledge Resource.
name	A human readable designation of the target resource.
refType	An optional ControlledTerm that classifies the target Resource.
refLang	An optional code that describes the syntactic form of the target Artifact, when the target Resource is a Knowledge Artifact.

Semantically, associations between Resources can be partitioned in 5 sub-categories, and further refined using hierarchies of properties defined in the API4KP-rel ontology.

- **Version (Series):**
  Relationship between two individual Knowledge Resources that are versions of the same Mutable entity and are partially ordered in a Series. As such, version-related Resources share essential characteristics such as subject or asset type/category.
  Newer versions are usually meant to replace the older ones. Succession between Assets often implies derivation from the predecessor version; succession between Expressions often, but not necessarily, implies derivation.

- **Derivation**
  Relationship between two Knowledge Resources A and B, which implies not only that B was used in the intellectual effort of creating A, but also that A exhibits some of the key concepts of B.
  A derivative Resource may derive from multiple source Resources at the same time, but derivatives are generally independent: as a consequence, derivatives are usually not used together with their sources.
  Derivation implies that the two Resources are distinct, so they do not need to share key characteristics.
- **Dependency**
  The Dependency of a Resource A on a Resource B implies that the use of A is impacted by the ability to acquire and use B at the same time. Resources may or may not be distributed together, allowing for late resolution and binding.
  The strength of the dependency, implied by the specific dependency relationship, determines if B is optional (A can still be used if B is not available), recommended (A can be used without B, although less effectively) or mandatory (A cannot be used without B).
- **Component**
  Structural relationships are used in the definition of Composite Assets and Artifacts. Resources that are structurally related cannot be used without each other and are retrieved and used together. In fact, removing or even rearranging the components results in a different Resource.
- **Role**
  Component Links allow to define the role that the target Resource plays with respect to the subject Resource. Role is used primarily in the relationships between a Composite and its Components.
- **Variant**
  A is a variant of B if and only if A and B are alternative, distinct representations of the same Knowledge Asset.
  In particular, Variance is based on characteristics of the Expression, not the carrier Artifact.
  Variance MAY be associated to derivation, especially when a Resource B is obtained from a Resource A by means of a ‘horizontal’ transrepresentation operation.
  Note: If asserted between Knowledge Assets, variance implies equivalence.

More details on the usage of relationships in combination with API4KP operations can be found in Annex A.

7.2.6 Knowledge Carrier

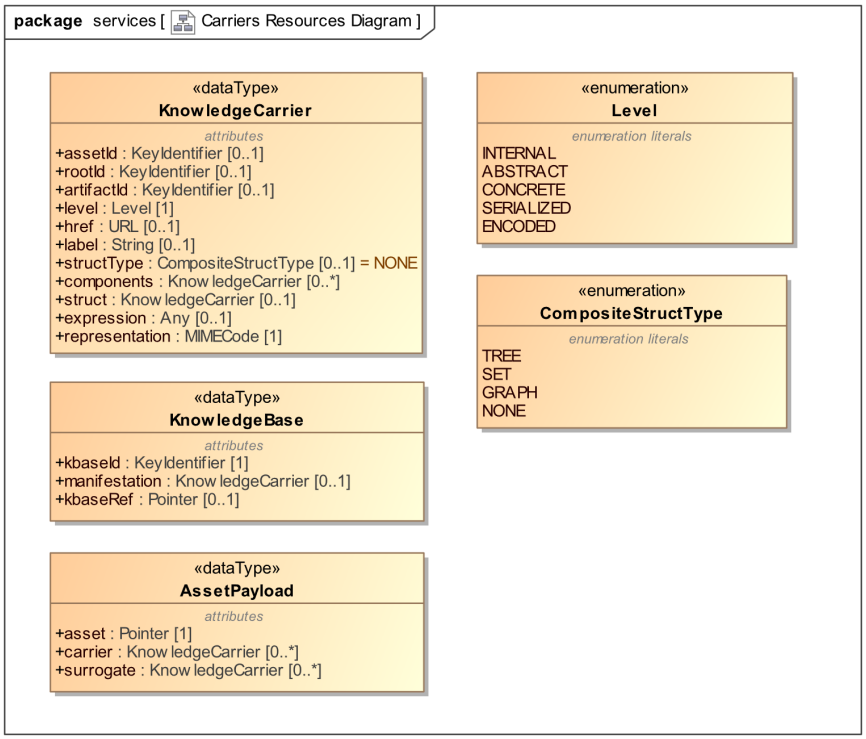

Figure 6: Structure of a Knowledge Carrier

The KnowledgeCarrier structure, given in

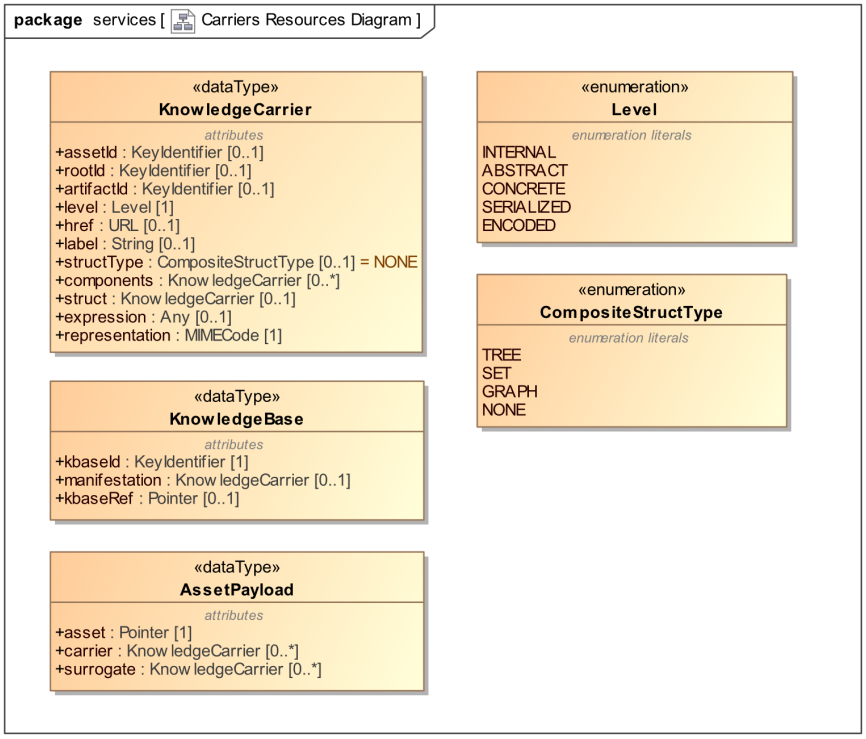

Figure *6*, is the runtime counterpart of KnowledgeAsset. As a Surrogate, KnowledgeAsset provides information on Knowledge Resources 'at rest'. KnowledgeCarrier, conversely, provides runtime metadata for Knowledge Resources 'in motion', as they are processed using the Operations exposed by the APIs. KnowledgeCarriers are initialized with information extracted from a KnowledgeAsset/Artifact surrogate and updated by the same operations that manipulate the carried Knowledge Resource.

7.2.6.1 Composite Knowledge Carrier

A Composite Knowledge Carrier is a Knowledge Carrier, and contains Knowledge Carriers, to support computation with Composite Knowledge Resources. While the components are stored in a flat list, a Composite KnowledgeCarrier delegates to a dedicated component, a "Struct", the responsibility of tracking the actual internal structural relationships. Structs are Expressions, and can be implemented using, e.g., extensional RDF graphs or intensional sequence of DOL structuring operations. Composite Knowledge Carriers enable the distribution (map/reduce) of operations from the composite to the components. To this end, Composite KnowledgeCarrier categorizes the Struct in terms of its topology and maintains a reference to the ID of the root component.

(Composite) KnowledgeCarrier

Attribute	Description
assetId	The identifier of the version of the carried Knowledge Asset.
artifactid	The identifier of the version of the wrapped Knowledge Artifact.
level	The level of abstraction of the wrapped Artifact (see 7.2.6.2).
label	A human readable designation of the wrapped Artifact.
href	The URL where the Artifact can be retrieved, if not embedded as "expression".
expression	The wrapped Knowledge Artifact, at the given parsing Level.
representation	A (formal) MIME Code that summarizes the representation aspects of the wrapped Artifact.
rootId	Only applies to TREE-oriented Composite Resources, identifying the "root" component.
structType	A summary descriptor of the topology of the Composite Artifact, or NONE for atomic Artifacts.

struct	The representation of the Structure of a Composite Artifact, itself wrapped in a KnowledgeCarrier. Atomic Artifacts should not have a Struct.
components	The Components of a Composite Artifact, each wrapped in a KnowledgeCarrier. In the case of Atomic Artifacts, any attempt to access the Components should return an empty collection, or a singleton that only comprises the atomic Artifact itself.

KnowledgeBase

A structured datatype that acts as a proxy for the manifestation of a named Knowledge Base. Note that Knowledge Bases are defined as Composite Knowledge Assets, which are manifested as (Composite) Knowledge Artifacts for computational purposes. A KnowledgeBase structure can either embed the Artifacts or reference their location.

Attribute	Description
kBaseId	The identifier of the version of a Composite Knowledge Asset, as a Knowledge Base.
manifestation	An embedded (Composite) Knowledge Carrier, which realizes the Knowledge Base.
kBaseRef	A Reference to the (Composite) Knowledge Base, when the Knowledge Base is not embedded. The Pointer should include a dereferenceable URI, or a href URL, to support the resolution of the KnowledgeBase content.

AssetPayload

A specialized KnowledgeCarrier designed to hold any number of Artifacts and Surrogates which co-reference the same Knowledge Asset, thus being mutual variants..

Attribute	Description
assetId	The identifier of a specific version of a Knowledge Asset.
carrier	The variant of Artifact(s) that embody the Knowledge Asset version.
surrogate	The variant of Surrogate(s) that embody a description (“metadata”) of the Knowledge Asset version.

7.2.6.2 Parsing Levels

One important consideration is that, while some operations may conceptually apply to Knowledge Resources at the Knowledge Asset level, computation can only happen if some kind of Knowledge Expression is involved. Knowledge Carriers use the notion of *Parsing Level* to reflect the highest *Lifting* that the Expression has been subject to. The parsing level determines the nature of the ‘expression object’ carried by the Knowledge Carrier.

- **(Internal Semantic Graph)**
  An internal, private representation that corresponds to an Agent’s internalization of a Knowledge Asset.
  Not used by Knowledge Carriers, whose primary role is to facilitate the flow of information between Agents, including servers that implement the APIs.
- **Abstract Knowledge Expression**
  An expression that conforms to the Abstract Syntax of the language used for representation.
  The Knowledge Carrier wraps an Abstract Syntax Tree / Abstract Syntax Graph representation of the Expression.
- **Concrete Knowledge Expression**
  An expression that conforms to the Concrete Syntax of the language used for representation
  The Knowledge Carrier wraps a Parse Tree representation of the Expression.
- **Serialized Knowledge Expression**
  A sequence of characters/symbols that has been generated according to the Concrete Syntax of the language used for representation.
  The Knowledge Carrier wraps a String representation of the Expression, based on a Character Set.
- **Encoded Knowledge Expression**
  A sequence/array/stream of bytes which results from a mapping of the characters/symbols to a binary encoding.
  The Knowledge Carrier wraps a binary-encoded representation of the Expression. The Encoding can

be the DEFAULT Character Set / Binary encoding provided by a platform, but also re-encodings (e.g., Base64), compressions (e.g., ZIP) and/or cryptographic encodings.

7.2.7 Monads

API4KP monads provide context around (Atomic) Knowledge Resource Objects, ensuring that operations, and chains thereof, can be applied consistently. Knowledge Representation languages and the tools that process them do not always support the context information natively because they are not generally designed for use in a hybrid environment. This gluing information is then provided as part of the API4KP infrastructure. Monad constructors ensure that necessary information such as identifiers and/or structure is available even when the languages used to express the knowledge do not support that natively; bindings then ensure that the context is maintained and propagated correctly as operations are performed on the Resources.

The role of Monads in API4KP can be summarized as follows:

- Monads are used as arguments by the public APIs.
- Monads wrap Resources expressed in a variety of notations, normalizing their use
  - Monads encapsulate the part of the API4KP specification that does not vary across logics, languages, and serializations.
- Monads bind Resources to API4KP atomic actions.
  - Operations, exposed as APIs, are defined in terms of chained, atomic, functional actions.

Conceptually, several Monads can be defined, each one highlighting a different aspect that concerns every API4KP operation, regardless of its specific nature and purpose.

7.2.7.1 Identifiable

Identifiable carries a Knowledge Resource's ID in context and ensures that IDs are propagated correctly as functions are applied to derive new Resources. Most operations transform a Resource and require assigning a new ID to the result. Some operations, however, preserve the identity of its operand(s).
For example, a *translation* action applied to Knowledge Resource preserves the ID of the Asset, but not the ID of the Expression, and thus impacts the ID of the Artifact the result will be engraved on.

```
Constructor:
      Identifiable<R,I a Identifier>
            = Dub R I | Mint R
Bind:
      Identifiable<R> >>= f
            = preserveIdentity( f ) ?
                  Dub f( R ) getId( R ) | Mint f( R )

Map:
      ( f : R → S ) → ( g: Identifiable<R> → Identifiable<S> )
            = sameAs( R, f( R ) ) ?
                  Dub f( R ) getId( R ) | Mint f( R )
Functions:
      getId : R → I
      has
d : R, I → bool
      newId : void → I
```

See also: Identity monad

7.2.7.2 Series

Series tags a Resource with version information, regardless of the actual implementation (timestamp-based, semantic versioning, incremental, etc.). Additionally, it ensures that version tags are updated properly as functions are applied to a Resource, depending on the nature of the function. An operation that does not alter its argument should not modify its version. Conversely, an operation that does not preserve Identity should also always generate a new version for the product. The new version may be set to an initial value (e.g., "0.0.1"), based on the time of execution, or derived (functionally) from a combination of the input's version and the operation.

```
Constructor:
      Versionable<R,V a VersionTag>
            = Tag R V | Init R
Bind:
      Versionable<R> >>= f
            = case:
                  preserveIdentity( f ) -> Tag f( R ) getVersionTag( R )
                  revise( f ) -> Tag f( R ) next( getVersionTag( R ) )
                  otherwise -> Init f( R )

Functions:
      getVersionTag : R → V
      hasVersion : R, V → bool
      next : V → V
      newTag : void → V
```

See also: Identity monad

7.2.7.3 Trace

Trace maintains the list of (versions of) a Resource, involved in a chain of computations, according to the Memento pattern. Trace is a specialization of List, which assumes that the elements are ordered.

When applied to Resources, Trace is used to retrieve particular versions of a Resource, as well as to apply Functions to either a specific version of a Resource, or to the entire chain.

See also: List monad

7.2.7.4 Carrier

Carrier wraps an Expression with context that contains information about the representation of the Expression itself. As a monad, it ensures that the metadata is updated consistently, according to the semantics of the action itself. The constituents of the Carrier are as follows:

```
Constructor:
      Carrier<R>
            = Tag R representationInfo
Bind:
      Carrier<R> >>= f
            = Carrier
                  f( R )
                  f'( f( R ), representationInfo ( R ) )

Functions:
      representationInfo : R → SyntacticRepresentation
```

Every action f that applies to Resources must also be implemented in a way that returns (infers, asserts, or retrieves) the representation metadata about the returned Resource. In general, this information is function of the specific f(R) and, secondarily of the metadata about R.

See also: Maybe monad

7.2.7.5 Structure

Structure allows composition of atomic Resources into complex ones, by means of parent/child (tree) and sibling (set) relationships. As described in the seminal paper[2]. Structure has two components: the structure itself, a composite tree/set organization of one of more (specific versions of specific, carrier-wrapped) Resources, and a 'manifest' structure descriptor, which is itself a Resource.

```
Constructor:
      Structure<T,S,C>
            = Empty | Construct (TreeSet T) (Manifest S)
      TreeSet<R>
            = Atomic R | TreeSet R
Bind:
      Structure<T,S,C> >>= f
            = case:
                  Empty → Empty
                  otherwise → Construct TreeSet f( T ) Manifest f'( S )
      TreeSet<R> >>= f
            = Atomic f( R ) | TreeSet f( R )

Functions:
      flatten : ( T, S ) → C
```

See also: Either Tree monad

7.2.7.6 Explain

Explain wraps a Resource and keeps an 'explanation' in context - an additional (Structured) Resource that carries additional information - e.g., provenance, or proofs - about the main Resource. The explanation is built incrementally, as operations are chained together, specializing the behavior of the classic Writer monad. Additionally, in case the Resource (or the Explanation) include variables/parameters, a Bindings structure is used to convey the associated values.

```
Constructor:
      Explain<S,E>
            = Explain S (With b)? (Explanation E)?

Bind:
      Explain<S> >>= f =
            X ← f( bind( S, b) )
            Explain X
                  (With subst( b, f'(X) ))?
                  (Explanation add( explain(X), E) )
```

[2] https://github.com/API4KBs/api4kbs/blob/master/publications/Monad_Trees.pdf

In particular, when Explains are chained through operations: i) the bindings are applied to the main structure S, ii) the associated action(s) are applied to the bound structure to derive the next Answer; iii) as needed, the bindings are propagated by substitution; iv) the Explanation for the latest action, if any, is incrementally added (structured) into the current Explanation.

See also: Writer monad

7.2.7.7 Integration

An API4KP Monad integrates *Identifiable, Series, Carrier and Explain,* and can be extended with *Series and Structure.* In the API specification, Knowledge Carriers are used in combination with a Monadic wrapper, Answer, that combines the behavior of all the API4KP monads. In particular, (Composite) KnowledgeCarrier provides the structural component, while Answer defines the monadic operations – return/of, bind/map, join/flatmap.
Answer is the same object has been introduced in Section 7.2.7, with the motivation of generalizing web-oriented operations to other integration patterns.

To support the development of the common API substrate, the following data structures harmonize some of the more popular, modern API development frameworks, while remaining compatible and aligned with the API4KP functional approach. Note that the structures are not specific to Knowledge APIs, and thus support but do not assume that the wrapped / references data is an actual Knowledge Resource.

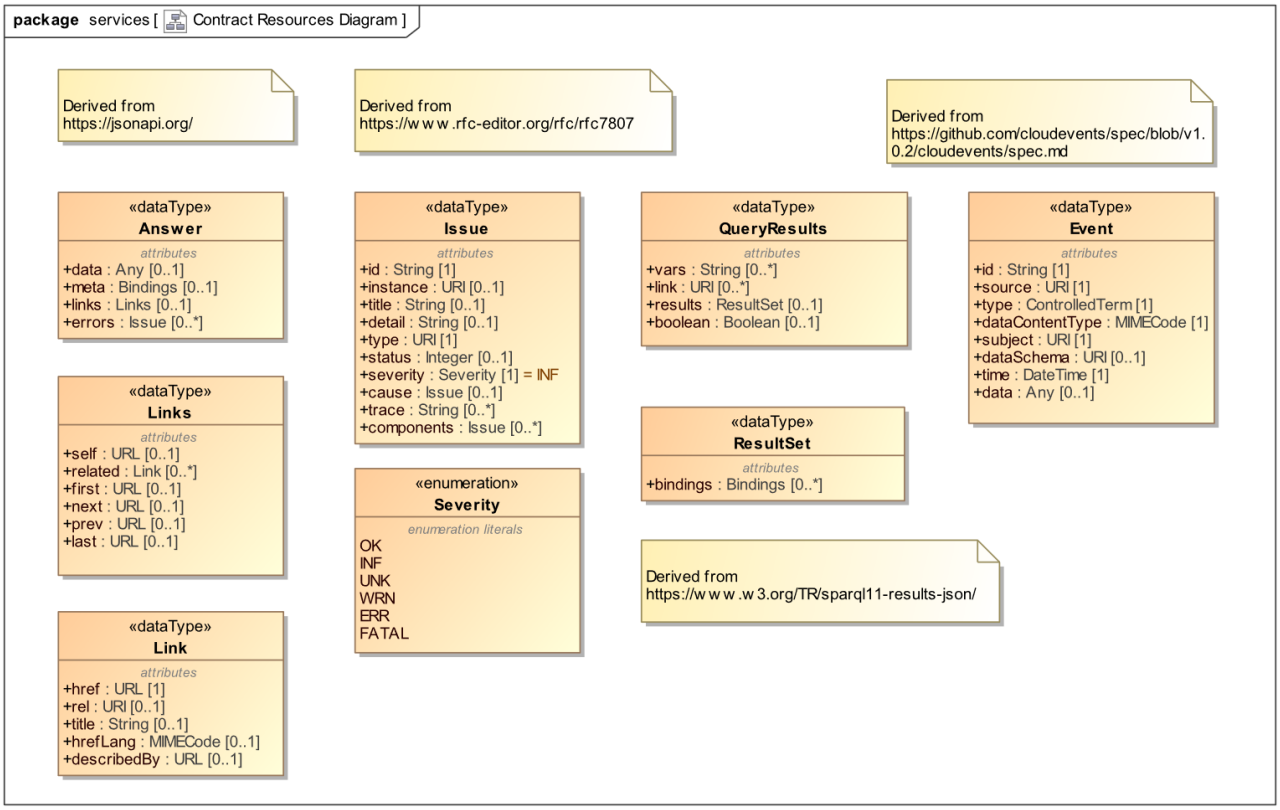

Figure 7: General Operation Patterns

Answer
The Answer data structure provides the substrate for the Answer monad.

Attribute	Description
data	The actual payload – in API4KP, this usually consists of a KnowledgeCarrier.
meta	Metadata about the execution of an operation. Mostly used for "technical" metadata, supports platform-specific execution contexts. In web-based implementations, this capability is implemented using HTTP headers.
links	Server-driven links to drive further interaction.
errors	Semantic metadata about the execution of an operation. Most often used to report errors and issues.

Links
Links is a wrapper object that arranges any server-driven suggestion for further interaction with additional resources, available at the linked endpoints. Links can be paginated.

Attribute	Description
self	A link to the original resource, which generated the Answer containing this supplemental Links.
related	The collection of Link objects.
first / next / prev / last	Pagination links, used when the 'related' list is considered too long.

Link

Attribute	Description
href	The URL where the linked target Resource is expected to be available.
describedBy	A reference to the formal description of the behavior of the linked resource. Can range from a classifier Concept URI (e.g., type, role) to the location of a Knowledge Resource that specifies the nature and/or behavior of the resource.
title	A human readable designation of the target Resource.
hrefLang	A MIME type that describes the syntactic representation of the information retrievable from the target link.
rel	A term that denotes the relationship between the Resource just acquired and the linked one.

Issue
Issue is a general-purpose data structure, based on https://www.rfc-editor.org/rfc/rfc7807, which can be used to describe a variety of operation outcomes, including errors of varying severity. When using Issue, servers should distinguish between the outcomes of an operation request as opposed to the outcomes of an operation execution.
For example, a request may be successful, but return no information because there is no information to return, which could be an issue from a client perspective. Likewise, a request to perform a consistency check on an Ontology may succeed yet discover that the ontology is inconsistent – a different scenario than one where either client or server was unable to provide/access the ontology to validate in the first place.

Attribute	Description
id	An identifier of the Issue instance.
instance	A reference to the primary individual entity that this Issue is about.
title	A human readable summary of the Issue.
detail	A full representation of the issue. May be human readable or be handled as an embedded Knowledge Expression.
type	The identifier of a classifier used to categorize this issue.
status	The operation request outcome, as a HTTP status code, reflecting the status of the server from the client's perspective.
severity	The operation execution outcome, as a severity level reflecting the server's perspective on the client.

	FATAL outcomes are expected block the client's execution; ERROR outcomes require the client's intervention; WARNING outcomes expect a client's eventual intervention; OK and INF outcomes do not need nor expect the client's intervention, respectively. UNK(nown) outcomes are undetermined.
cause	An upstream Issue, which is considered to be the cause of this Issue.
trace	An explanation of the issue.
components	Sub-issues, used as components/fragments to describe this Issue in finer details.

QueryResults

A wrapper structure that provides context for a ResultSet, returned in response to a Query.

Attribute	Description
vars	The list of variables for which bindings to result values are provided.
link	Generic reference to additional information.
results	The query response, as a matrix of variable bindings. Specifically, a ResultSet is a collection of Bindings, where each element in the collection describes a different entity, while each entity is described by a variable Bindings.
boolean	The query response, for queries that have a boolean response.

Event

Event is a data structure that can wrap Event payloads, in line with the CloudEvents specification.

Attribute	Description
id	A unique identifier of this Event instance.
source	The identifier of the context where the event was originated.
type	A classifier of the Event.
dataContentType	A MIME type that describes the format of the event payload. Must complement, or be consistent, with the dataSchema.
subject	The identifier of the primary entity that the event is about.
dataSchema	The identifier of the grammar/schema used to represent the event payload.
time	The time at which the event occurred.
data	The payload that describes the event occurrence.

7.2.8 Operations - General Patterns

In line with the general API4KP principles, most operations are designed to follow a common pattern, as shown below in Figure *8*.

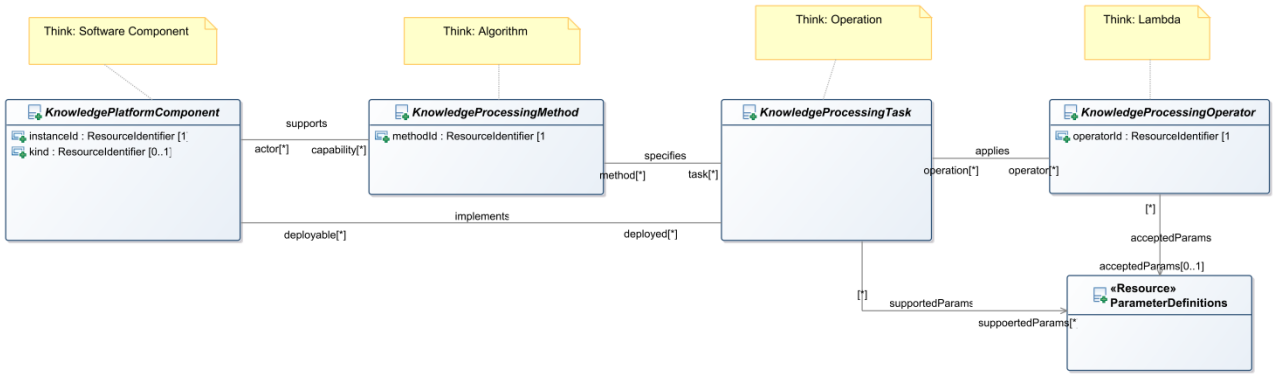

Figure 8: General Operation Patterns

API4KP Components are named Software components that implement at least one API4KP Knowledge Processing Task. The role of API4KP Component can be played by existing knowledge-oriented software, wrapped using API4KP interfaces, but also dedicated software that implements a variety of different algorithms.

An API4KP Component must provide behaviors that are compliant with the operation semantics defined in the API4KP ontology of Knowledge Processing Operations (api4kp-ops).
Service endpoints that conform to the API4KP signatures expose the operations to clients as functions. Operators are the modules that bind the signature to the underlying implementation and are usually realized with strategies that range from sub-components to “lambdas”.

The APIs allow for some degree of insight into the Knowledge Platform Implementation.

Discovery

Servers as a whole, as well as individual Operators, can return self-describing resources with metadata and other descriptive information. Knowledge Platform components – including servers - can use “manifest” data structures to describe their own capabilities. The manifests include Operator descriptors, which can be used to advertise the specific operation types provided by the platform component.

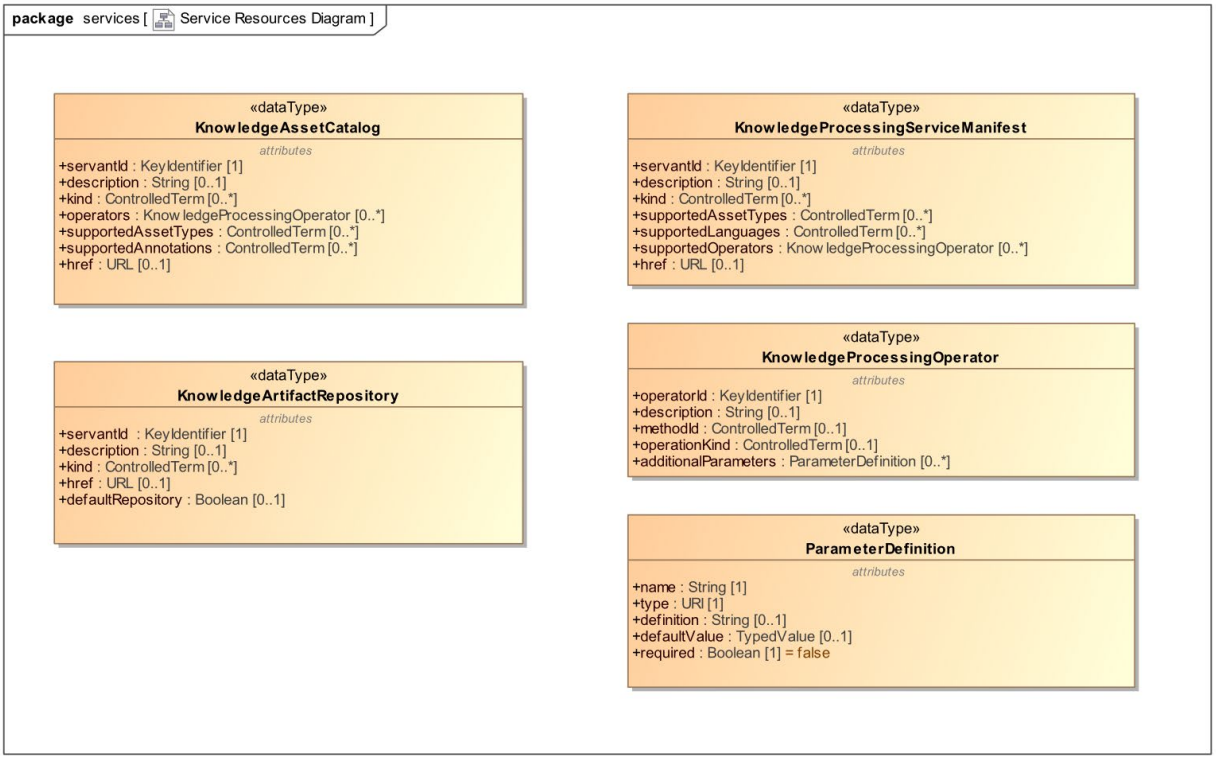

Figure 9: Service Capability Manifests

KnowledgeAssetCatalog

The manifest (summary descriptor) of a Semantic Knowledge Asset Repository Service.

Attribute	Description
servantId	A unique identifier of the server, as a specific implementation of the API4KP specification.
description	A human readable name and/or description of the server.
kind	ControlledTerm that denotes a classifier that applies to this server, according to some classification scheme implemented in a given Knowledge Platform.
href	The base URL where the server is deployed.

operators	The additional KnowledgeProcessingOperator that are embedded in the server, augmenting its capabilities.
supportedAssetTypes	The list of Types (Classifiers) of Knowledge Assets that the server has the capability to process.
supportedAnnotations	The list of Property types that this server is able to support as Annotations (Resource / Concept association).

KnowledgeArtifactRepositoryManifest

The manifest (summary descriptor) of a Knowledge Artifact Repository Service.

Attribute	Description
servantId	A unique identifier of the server, as a specific implementation of the API4KP specification.
description	A human readable name and/or description of the server.
kind	ControlledTerm that denotes a classifier that applies to this server, according to some classification scheme implemented in a given Knowledge Platform.
href	The base URL where the server is deployed.
default	Flag that denotes a 'default' repository, where requests for specific Artifacts should be routed to, unless otherwise specified.

KnowledgeProcessingServiceManifest

The manifest (summary descriptor) of a Knowledge Transrepresentation, Construction and/or Reasoning Service.

Attribute	Description
servantId	A unique identifier of the server, as a specific implementation of the API4KP specification.
description	A human readable name and/or description of the server.
kind	ControlledTerm that denotes a classifier that applies to this server, according to some classification scheme implemented in a given Knowledge Platform.
href	The base URL where the server is deployed.
supportedOperators	The KnowledgeProcessingOperators instantiated by the server.
supportedAssetTypes	The list of Types (Classifiers) of Knowledge Assets that the server has the capability to process.
supportedLanguages	The list of Representations of Knowledge Artifacts that she server has the capability to process.

KnowledgeProcessingOperator

The descriptor of any specific Knowledge Processing Operation implemented by a Knowledge Platform component. While API4KP endpoints differentiate the various Knowledge Processing Tasks structurally, the endpoints are not able to provide the semantic details of how an operation has been implemented by a specific server. The Operator descriptor, or extensions thereof, is designed to provide the additional information.

Attribute	Description
operatorId	A unique identifier of the operator, specific to the implementation (version), but common across its deployments and instatiations.
description	A human readable name and/or description of the operator.
methodId	A Term that classifies the implementation technique(s) that the operator is based on (e.g., logic-based reasoning, NLP), up to denoting the specific algorithm, if well-known.
operationKind	A ControlledTerm that classifies the specific type of Knowledge processing operation, consistent with (any extension of) the API4KP Knowledge Operations ontology.
additionalParameters	Operator-specific parameters that allow clients to further refine the behavior of the server.

Parameters

Several operations allow the client to provide component-specific Parameters.
API4KP parameters are key/value pairs of simple Strings, which are (de)serialized according to the following grammar.

```
<Parameters> := <Parameter> (',' <Parameter>)*
<Parameter> := <Key> '=' <Value>
<Key> := <STR>
<Value> := <STR>
<STR> := \w*
```

Component descriptors should include ParameterDefinitions – simple metadata objects that enumerate the supported parameters and map each parameter to a definition. The definition COULD consist in a Knowledge Asset URI, to link to formal, machine readable and/or computable definitions.

ParameterDefinition

Attribute	Description
name	The unique name of the Parameter.
type	The datatype of the Parameter value.
definition	A human readable definition of the parameter's purpose, admissible values, and general usage.
required	If true, the parameter will be considered mandatory. Clients are expected to provide a value, or a default value will be used, or the operation request will fail.
defaultValue	A representation of the value that will be assigned to the parameter, when no value is provided by the client.

Parameters can be used to drive the behavior of an operator, and could be used to refine, but must not extend nor alter the execution semantics of an operation. In particular, parameters MUST not be used to drive an API4KP endpoint to provide a function that should be exposed using a different API4KP endpoint.

Content Negotiation

APIs that return Knowledge Artifacts COULD support content negotiation to return variant formats of the resulting Artifacts, in order to meet client's preferences.

Given that APIKP APIs use (Composite)KnowledgeCarrier wrappers, content negotiation, when supported, should distinguish between the format and/or encoding of the wrapper from the language, serialization, format, and/or binary encoding used in the Artifact itself. The former is usually controlled by the implementation frameworks: over web transactions, for example, Accept and Content-Type headers are used by user agents such as browsers and REST clients. The latter should be controlled by the API4KP components. Operations that support content negotiation expose an optional "extended Accept" parameter. Implementations should distinguish between their (in)ability to support content negotiation in general (*Unsupported*), from their inability to handle individual requests (*NotAcceptable*).

Pagination and Filtering

Operations that enumerate collections of resources COULD support pagination and filtering.

Pagination is supported using optional parameters *offset* and *limit*, following the usual semantics of indexing a Collection, returning resources in the range [offset .. offset + limit]. Default values of 0 and -1, respectively, allow to access the entire collection.

Sorting and filtering is resource specific. Unless otherwise specified, sorting is performed according to the timestamps associated to the resources' identifiers. The default filter is the null filter, which returns the entire collection.

Operations are performed in the order: filtering, sorting, pagination.

API4KP Services

Each of the services mentioned below is fully documented in the OpenAPI documentation that is included by reference herein. Figure *10* provides a high-level view of the services defined for API4KP.

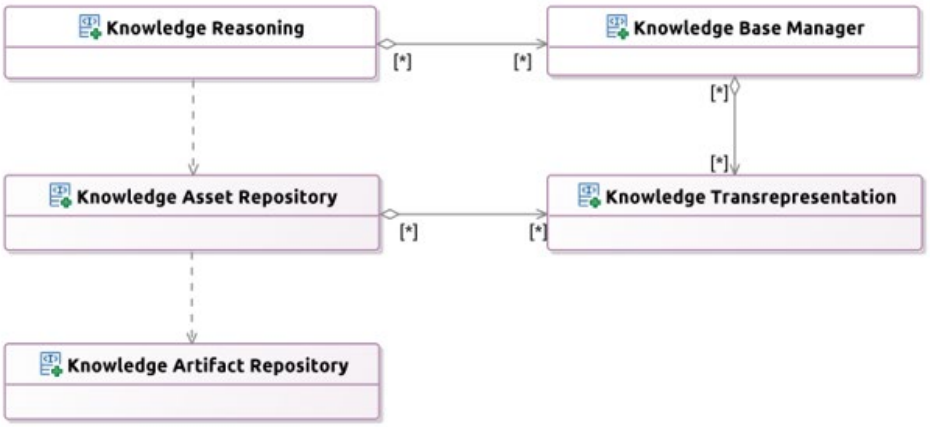

Figure 10: API4KP Services

7.2.9 Knowledge Artifact Repository Service

Knowledge Artifact Repositories storage and retrieval of (copies of) digital knowledge artifacts (KA). KARs treat KAs as black-box binary objects, so there is no limitation nor expectation on the nature of the content, or the requirements to consume it. However, identity and versioning must be supported. Identifiers must be universal, unique, and opaque, so they MUST be UUID v4 compliant. Version tags can follow different patterns (semantic versioning, incremental numbering, date/time stamps, etc...). Special considerations involve the deletion of an Artifact. For traceability and safety purposes, KARs SHOULD NOT allow Artifacts to be deleted in an unrecoverable way. Deletion itself is defined as making an Artifact no longer accessible to a client (i.e., status 404). A server SHOULD allow a deletion operation to be undone, e.g., using mechanisms conceptually similar to 'trash bins', and SHOULD at a minimum keep track of the IDs of Artifacts that were at some point managed in each Repository. For this reason, a two-phase deletion is recommended. Deleted Artifacts transition into a 'deleted' status in which they cannot be discovered nor retrieved unless a dedicated flag is set. Once in a deleted state, Artifacts MAY be deleted permanently.

The API also supports the (logical) federation of Repositories. A server instance MAY expose different repositories to a client, who should expect each repository to be independent. Whether these repositories map to actual physical repositories (e.g., different DBs), folder-like structures or logic tags/collections is left to the implementation.
The same artifact (as defined by having the same ID) COULD be stored in more than one repository, but all copies MUST be identical to each other.

With adequate rights, and if supported by the implementation, repositories can be enabled or disabled. Enabled (resp. disabled) repositories are (resp. not) available to a client, regardless of whether the (de)allocation of actual resources is involved at the implementation level.

The Knowledge Artifact Repository Service is fully specified in the API4KP OpenAPI Documentation / Knowledge Artifact Repository (https://www.omg.org/spec/API4KP/1.0/KnowledgeArtifactRepository.html).
An overview of the interfaces is provided in Figure *11*.

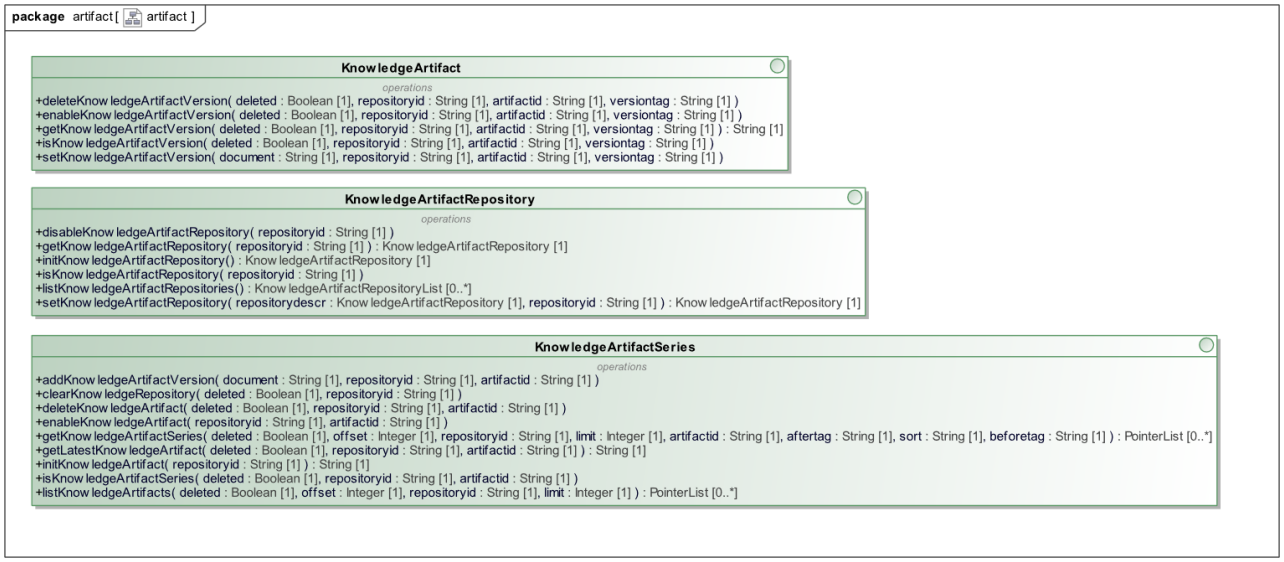

Figure 11: Knowledge Artifact Repository Service Interfaces

7.2.10 Knowledge Asset Repository Service

Knowledge Assets are immutable, versioned works of knowledge that are expressible in any form fit for consumption by a designated audience. Assets managed through a Knowledge Asset catalog and repository are usually, though not necessarily, enterprise knowledge assets. In other words, they are assets whose content is endorsed by some subject matter expert (party), and whose identification and life cycle is managed by an authority that registers them in the repository. A Knowledge Asset Repository catalogs surrogates carrying the descriptions ('metadata') of the knowledge assets and can resolve references to artifacts that are carriers of those assets. Knowledge Asset Repositories also support the discovery of knowledge assets through the same metadata structures. API4KP Asset Repositories are model driven and semantically aware. In particular, they treat the entirety of the surrogates they maintain as a Knowledge Base, which may be queried and reasoned over.

The Knowledge Artifact Repository Service is fully specified in the API4KP OpenAPI Documentation / Knowledge Asset Repository (https://www.omg.org/spec/API4KP/1.0/KnowledgeAssetRepository.html).
An overview of the interfaces is provided in Figure *12*.

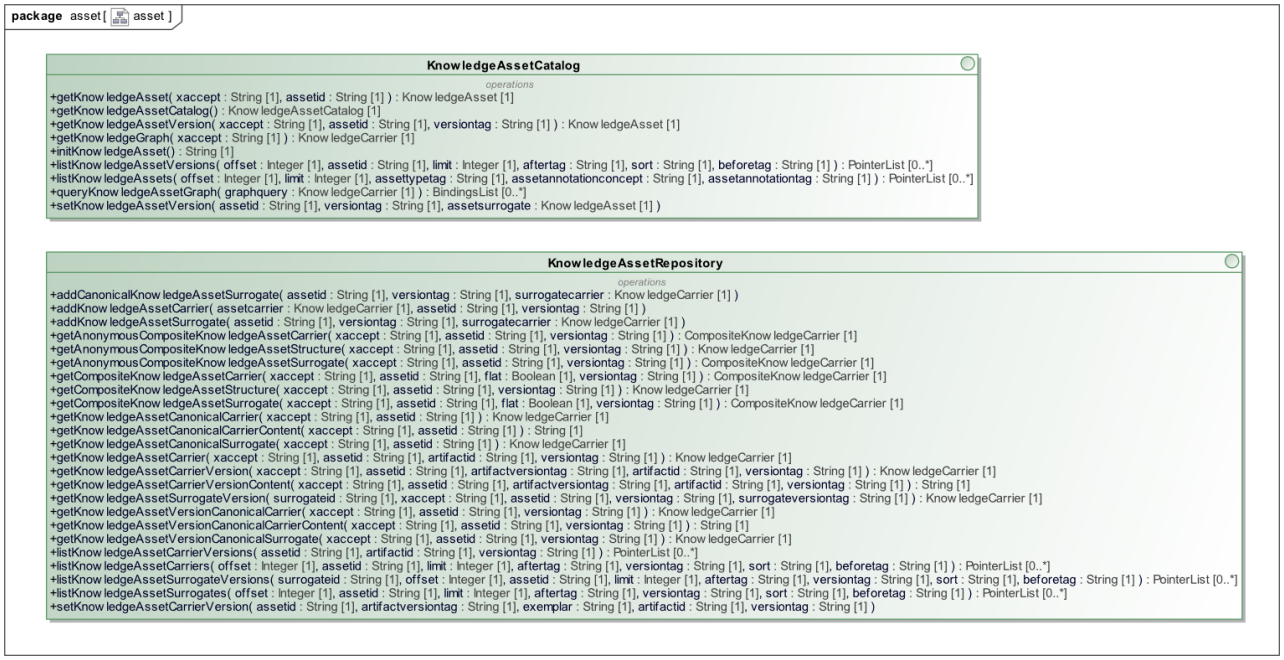

Figure 12: Knowledge Asset Repository Service Interfaces

7.2.11 Knowledge Asset Transrepresentation Service

This API defines "syntactic" manipulations of Knowledge Artifacts, based on the stratified representational aspects of the Artifacts themselves (Language, profile, syntax/serialization, meta-format, encoding). It supports both 'vertical' operations (parsing/serialization), which preserve the Asset and the Language, and 'horizontal' operations (transrepresentations) which preserve the aspects up to a certain level, but map across variants at the same level.

The Transrepresentation Service also exposes detection and validation capabilities: the former is used to infer the SyntacticRepresentation of a given Knowledge Artifact, the latter is used to validate the conformance of a Knowledge Artifact with respect to a given SyntacticRepresentation.

The Knowledge Asset Transrepresentation Service is fully specified in the API4KP OpenAPI Documentation / Knowledge Asset Transrepresentation (https://www.omg.org/spec/API4KP/1.0/KnowledgeAssetTransrepresentation.html).
An overview of the interfaces is provided in Figure *12*.

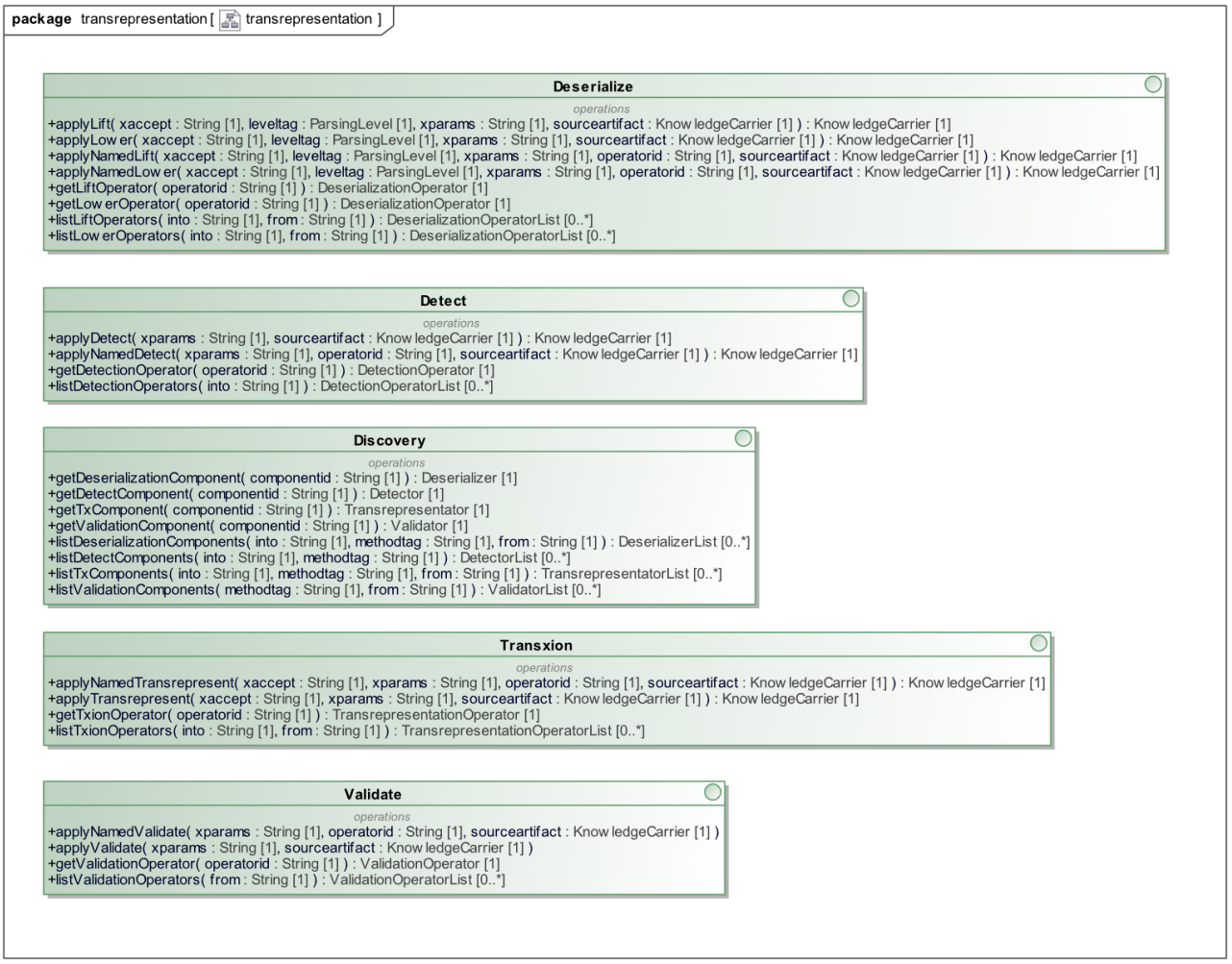

Figure 13: Knowledge Transrepresentation Service Interfaces

7.2.12 Knowledge Base Construction Service

The KnowledgeBase APIs enable the transition between Knowledge *at rest*, i.e., Knowledge in the form of Artifacts stored in a repository and not yet assembled into a Knowledge Base, and Knowledge *in motion*, i.e., Knowledge Bases deployed/paired with a runtime engine/reasoner/execution platform that is able to perform computations using that Knowledge.

The Knowledge Base Management API is inspired by the State monad. Knowledge Bases are incubated within the server from their initialization, through their construction, until their deployment. As operations are applied to manipulate the KB, new versions are constructed ensuring reproducibility and traceability. Implementations, however, are not required nor guaranteed to be transactional.

The API consists in two groups of Operators. Composition Operators allow to construct, incrementally, a Knowledge Base starting from known named (carriers of) Knowledge Assets. Transcreation operators allow to mutate Knowledge Artifacts, usually to create transient ephemeral versions which are used to prepare the KnowledgeBase for deployment but would not otherwise be treated as Assets.

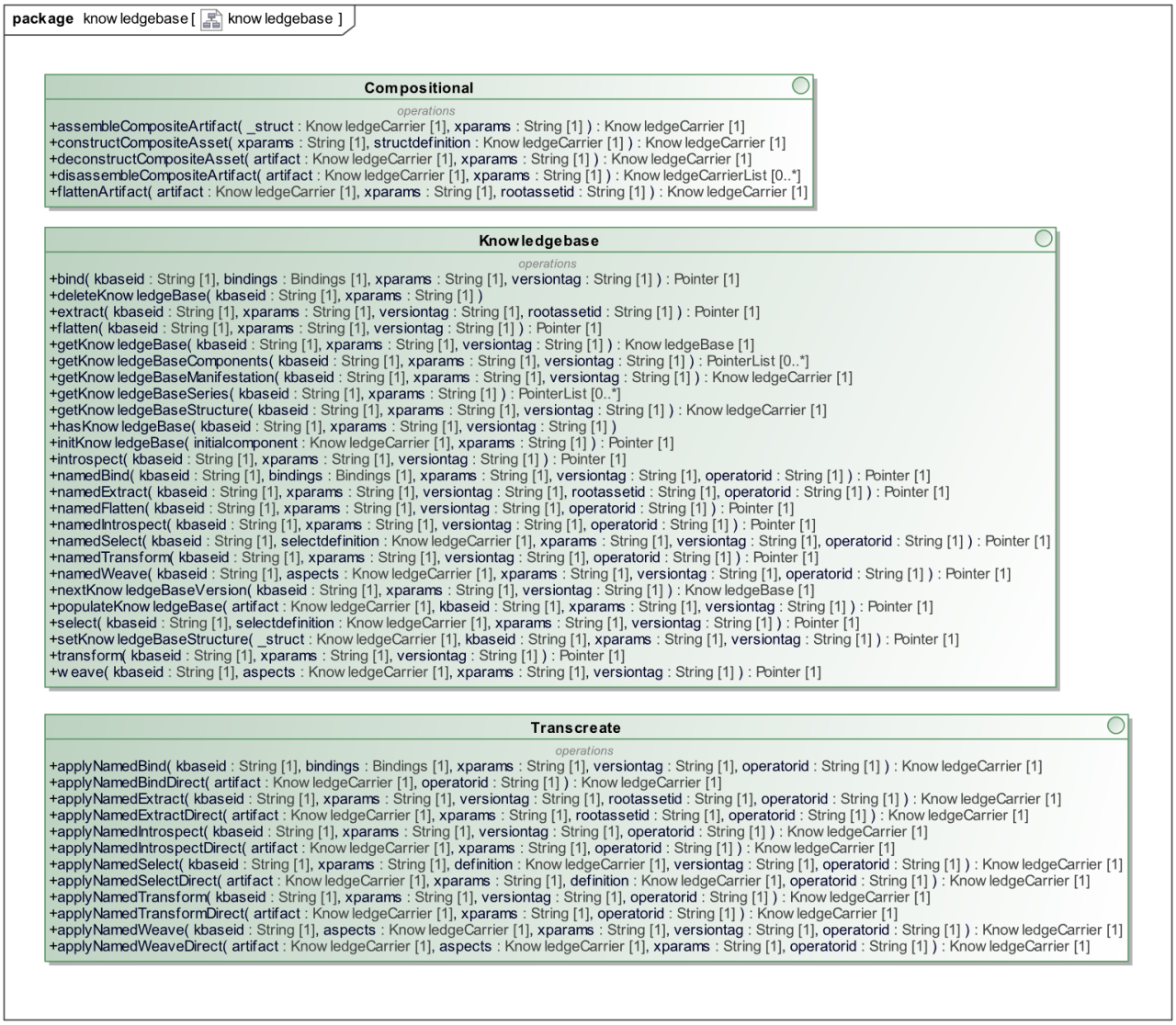

Figure 14: Knowledge Base Construction Service Interface

The Knowledge Base Construction Service is fully specified in the API4KP OpenAPI Documentation / Knowledge Base Construction (https://www.omg.org/spec/API4KP/1.0/KnowledgeBaseConstruction.html).
An overview of the interfaces is provided in Figure *11*.

7.2.13 Knowledge Base Reasoning Service

The Reasoning APIs expose the information processing capabilities of Knowledge Platform Components, typically called engines or reasoners, which are able to apply "knowledge" to "data", in order to derive new information.

Knowledge Reasoning APIs are likely to provide an abstraction layer for the proprietary API of existing engines but could also be used to expose engine-less microservices designed to work with individual, named knowledge bases.

The APIs pivot on the notion of Knowledge Base _in motion_, and consider reasoners as operators applied to the KBs. In this context, Knowledge Bases prepared for Reasoning are also called Knowledge Models, or "Models" for short, providing a connection to modern AI implementations.

The binding between the Knowledge Base and the Reasoning service can be implemented in different ways. Patterns include, but are not limited to:

- Knowledge Bases deployed within an engine backing the server.
- Knowledge Bases implemented by the server directly, through manual software development or trans-compilation process.
- References to remote/distributed Knowledge Bases that can be resolved by the server.
- Proxies/Brokers/Adapters where the server delegates the execution to another service.

The Knowledge Base Inference Service is fully specified in the API4KP OpenAPI Documentation / Knowledge Base Inference (https://www.omg.org/spec/API4KP/1.0/KnowledgeReasoning.html).
An overview of the interfaces is provided in Figure *11*.

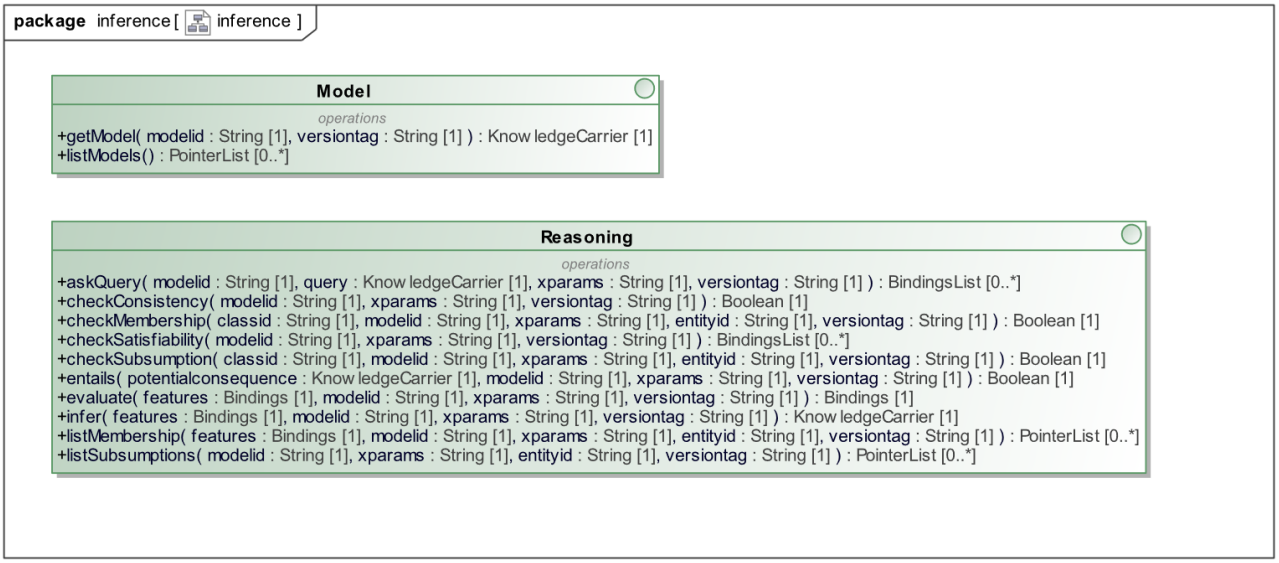

Figure 15: Knowledge Base Reasoning Service Interface

Annex A: API4KP Ontologies (normative)

The API4KP ontology is composed by a family of ontologies, that formalize the vocabulary defined in Clause 4 as well as those introduced in Annex A and that are used throughout the specification. They also drive the generation of the APIs.

A.1 Namespace Definitions

The namespaces and prefixes corresponding to external elements required for use in API4KP are provided herein. Table A-1 lists the prefixes and namespaces on which API4KP depends that are external to API4KP. Table A-2 provides the namespace declarations required for use of API4KP itself. The prefixes provided in Tables A-1 and A-2 are normative, and their use is required in any conformant extension.

API4KP-23 – Augment external namespaces to incorporate the referenced Commons ontologies and eliminate the use of Specification Metadata

Table A-1. Prefix and Namespaces for referenced/external vocabularies

Namespace Prefix	Namespace
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
owl	http://www.w3.org/2002/07/owl#
xsd	http://www.w3.org/2001/XMLSchema#
dct	http://purl.org/dc/terms/
skos	http://www.w3.org/2004/02/skos/core#
cmns-av	https://www.omg.org/spec/Commons/AnnotationVocabulary/
cmns-cds	https://www.omg.org/spec/Commons/CodesAndCodeSets/
cmns-col	https://www.omg.org/spec/Commons/Collections/
cmns-cxtdsg	https://www.omg.org/spec/Commons/ContextualDesignators/
cmns-dsg	https://www.omg.org/spec/Commons/Designators/
cmns-id	https://www.omg.org/spec/Commons/Identifiers/
dol	https://www.omg.org/spec/DOL/DOL-terms/
lcc-lr	https://www.omg.org/spec/LCC/Languages/LanguageRepresentation/

The namespace approach taken for API4KP is based on OMG guidelines and is constructed as follows:

- A standard prefix `https://www.omg.org/spec/`
- The abbreviation for the specification: in this case `API4KP`
- The ontology name (including the module)

Note that the URI/IRI strategy for the ontologies in API4KP takes a “slash” rather than “hash” approach, in order to accommodate server-side applications. Namespace prefixes are constructed as follows with the components separated by “-“:

- The specification abbreviation: `api4kp`
- An abbreviation for the ontology name

The namespaces and prefixes corresponding to the normative Application Programming Interfaces for Knowledge Platforms (API4KP) ontologies are summarized in Table A-2. These are given in alphabetical order, rather than with any intent to show imports relationships.

Table A-2. Prefix and Namespaces for the normative ontologies comprising Application Programming Interfaces for Knowledge Platforms (API4KP)

Namespace Prefix	Namespace
api4kp	https://www.omg.org/spec/API4KP/api4kp/
api4kp-kao	https://www.omg.org/spec/API4KP/api4kp-kao/
api4kp-kp	https://www.omg.org/spec/API4KP/api4kp-kp/
api4kp-krr	https://www.omg.org/spec/API4KP/api4kp-krr/
api4kp-lang	https://www.omg.org/spec/API4KP/api4kp-lang/
api4kp-ops	https://www.omg.org/spec/API4KP/api4kp-ops/
api4kp-rel	https://www.omg.org/spec/API4KP/api4kp-rel/
api4kp-series	https://www.omg.org/spec/API4KP/api4kp-series/

A.2 Ontology Overview

This section provides an overview of the terms, definitions, relationships, and additional logic specified in the ontologies that make up normative API4KP ontologies.

API4KP Core Ontology

The API4KP core ontology provides a systemic description of the vocabulary used throughout the specification. It defines foundational concepts including that of a knowledge resource (asset, expression, artifact) and the basic relationships between them, effectively serving as an upper ontology for the other modules.

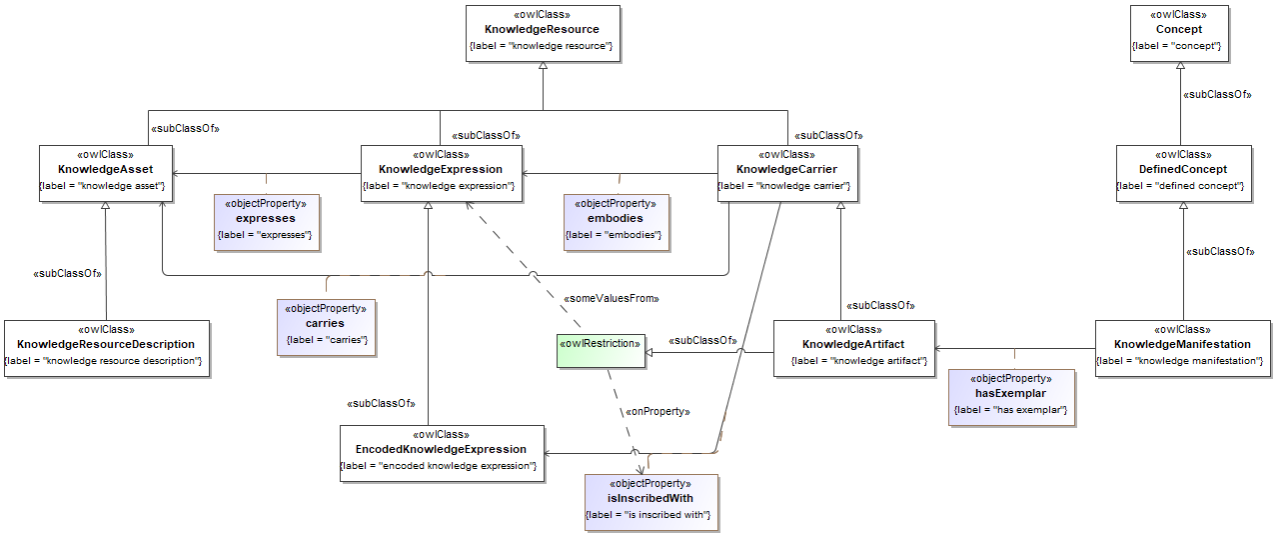

Figure 16: API4KP Knowledge Resource Hierarchy

Figure 16 provides a view of the top-level core elements of the API4KP vocabulary and how they relate to one another. Many elements that can be expressed using the ontology are either knowledge resources or concepts, where concepts are the atomic 'units of knowledge' that constitute (pieces of) knowledge.

Though many definitions exist, API4KP defines Knowledge as the 'Cognition (know-what), pragmatics (know-how) and understanding (know-why) about the nature and/or behavior of something that, when internalized by an agent, has the potential of generating actions in situations that the knowledge applies to'. The definition emphasizes the roles of semantics – both formal and domain oriented – in the use of knowledge for cognitive processing.

A knowledge resource, as defined herein, is an 'immutable, identifiable, versionable entity that is, expresses or carries some piece of knowledge'. Pieces of knowledge that are deliberately scoped and constructed for communication and processing are defined 'works of knowledge'. A knowledge asset is a 'work of knowledge that is a knowledge resource considered valuable by a party'. A knowledge expression is an 'expression of a piece of knowledge in some language, i.e., using a combination of signs and symbols that conform to the rules of the grammar of that language'. A knowledge artifact is a 'digital or physical object that is specifically constructed to carry one or more (expressions of) knowledge assets.' In other words, the core 'creative work', in the sense of an FRBR creative work, is a knowledge asset, which may be expressed in any number of ways, represented by a 'knowledge expression', and that may be embodied in any number of artifacts that carry specific versions of an expression of that asset. In this sense, Assets play the role of (Knowledge) *Content* with respect to Artifacts, and Artifacts play the role of *Carrier*.

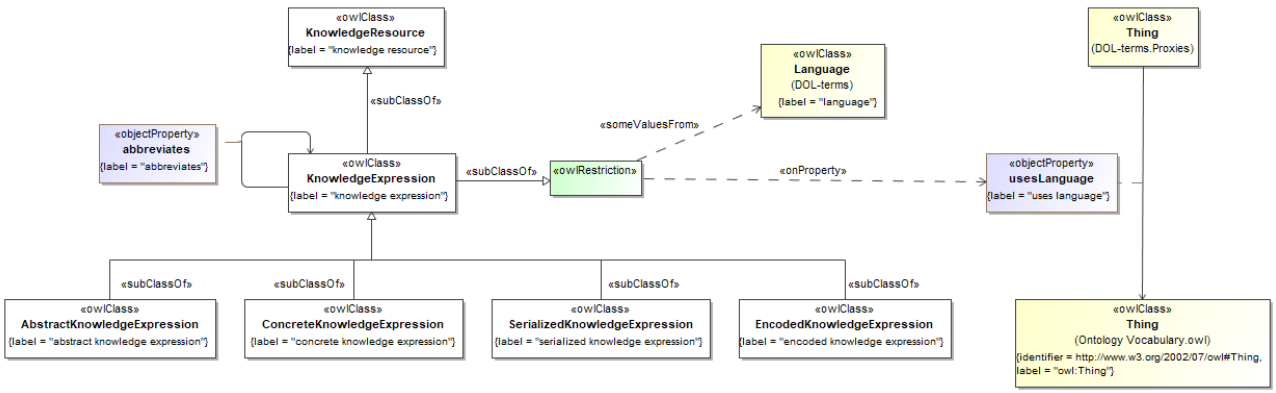

Figure 17: API4KP Knowledge Expression Hierarchy

Figure 17 provides a bit more context, depicting the various forms that a knowledge expression may take in terms of variations in encoding and serialization. In order to be embodied into Digital Knowledge Artifacts, usually for persistence, exchange and processing purposes, Knowledge Assets need to be:

- Expressed using the syntactic constructs of a (Knowledge Representation) Language, in the form of one or more sentences that conform to the rules of the Language's Grammar.
- Further concretized using markup and delimiter constructs, to ensure the unambiguous recognition of the structure of the expression, enabling the serialization of the Expression.
- Serialized into a sequence ('string') of symbols ('characters') from a given Alphabet.
- Encoded in binary form, mapping each symbol to a binary representation.

The dual process of internalizing a Knowledge Artifact leads to the 'Parsing Levels': an Encoded Expression is internalized as a sequence of bytes; bytes are decoded into Characters, to obtain a serialized String; the String is deconstructed into tokens by, and organized into a parse tree that reflects the structural patterns of the language's concrete syntax; eventually, the information content of the parse tree is extracted into an abstract syntax tree (AST). Semantic systems may further map the linguistic constructs of the AST to (an internal representation of) the concepts associated to the constructs, creating an abstract syntax graph (ASG) in the process.

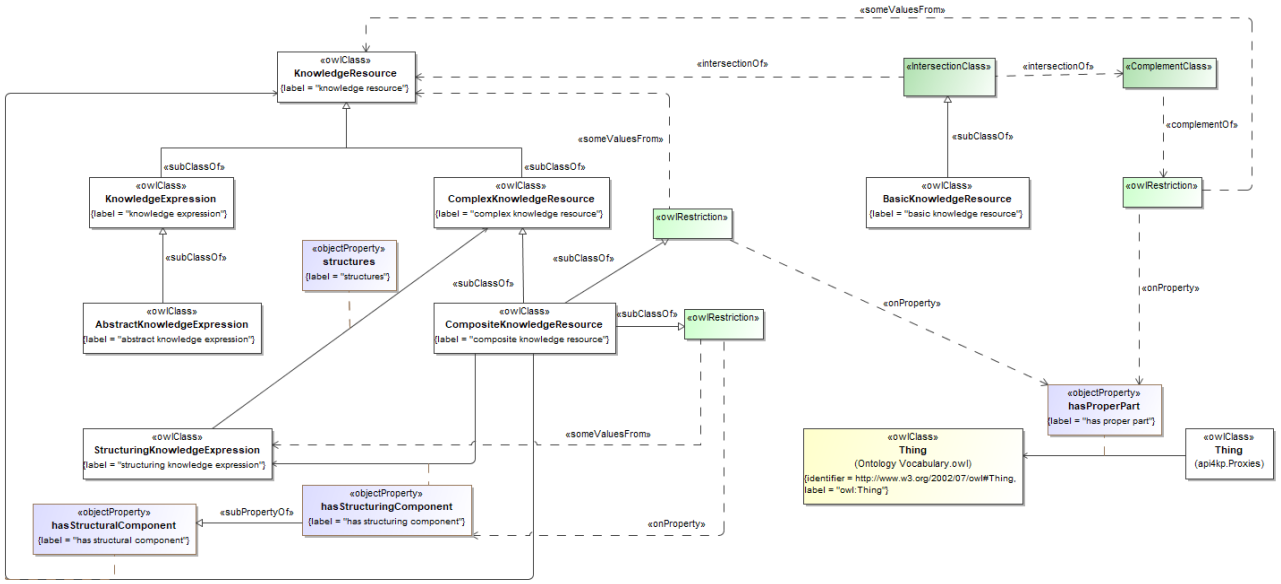

Figure 18: Basic and Complex Resources

Figure 18 shows the distinction between a basic and complex knowledge resource. A basic resource is atomic, i.e., one that does not have proper parts that are themselves individual knowledge resources. Basic resources can be further deconstructed into Fragments, usually corresponding to sentences or constructs in the resource's language, which can only be addressed in the context of the scoping resource.

In contrast, a complex knowledge resource is one that can possibly be decomposed into proper parts, i.e., parts that can be assigned identity and treated (recursively) as knowledge resources, typically at the same level of abstraction. A resource that is actually deconstructed is further considered a Composite knowledge resource. A Composite Resource always has a special component, which is a knowledge resource itself: a *Structuring* Component (often called 'Structure') that establishes the identity, types, roles, and relationships of the Composite with respect to its Components. In particular, Composites are *Homogeneous* (vs Heterogeneous) when all the components share the same representation, and Pure (vs Hybrid) if all the components share the same formal type. While Knowledge Artifacts processed individually are often Basic resources, non-trivial Knowledge Bases are usually Composite. Knowledge Base Construction APIs can be used to manipulate and assemble – as opposed to retrieve – Composite Knowledge Resources.

The additional ontologies included in the normative set of ontologies that comprise the specification build on these basic concepts.

API4KP Knowledge Asset Type Ontology (KAO)

The Knowledge Asset Type Ontology (KAO) provides a classification scheme for knowledge assets based on the logical and mathematical constructs used in the formalization of the assets themselves. The classification, in turn, determines (i) what (knowledge representation) languages are suitable to express the assets, and (ii) the kind of reasoning activities that can be performed using the assets. In other words, the classification scheme determine what kind of operations can be performed, and what kind of platform components are required in order to perform those operations.

Notice that this *formal* classification does not prevent, though may correlate, with other classification schemes such as ones based on domain-specific semantics. One should also consider that the classification is defined at the Knowledge Asset level, and thus does not depend on the choice of representation language, setting the basis for the application of iso-semantic and iso-pragmatic 'horizontal' API4KP operations.

A further corollary is that the choice of representation language may be contingent, and not sufficient, to determine the nature of the Asset carried by an Artifact. For example, an Artifact that embeds a well-formed OWL expression may not actually carry an Ontology, in the sense of a logically consistent, semantically correct, and pragmatically reasonable conceptualization of a domain of interest based on a first-order formalization in a description logic. However, one such Ontology could be expressed in OWL, but also in Common Logic, and the two variants could be translatable into each other.

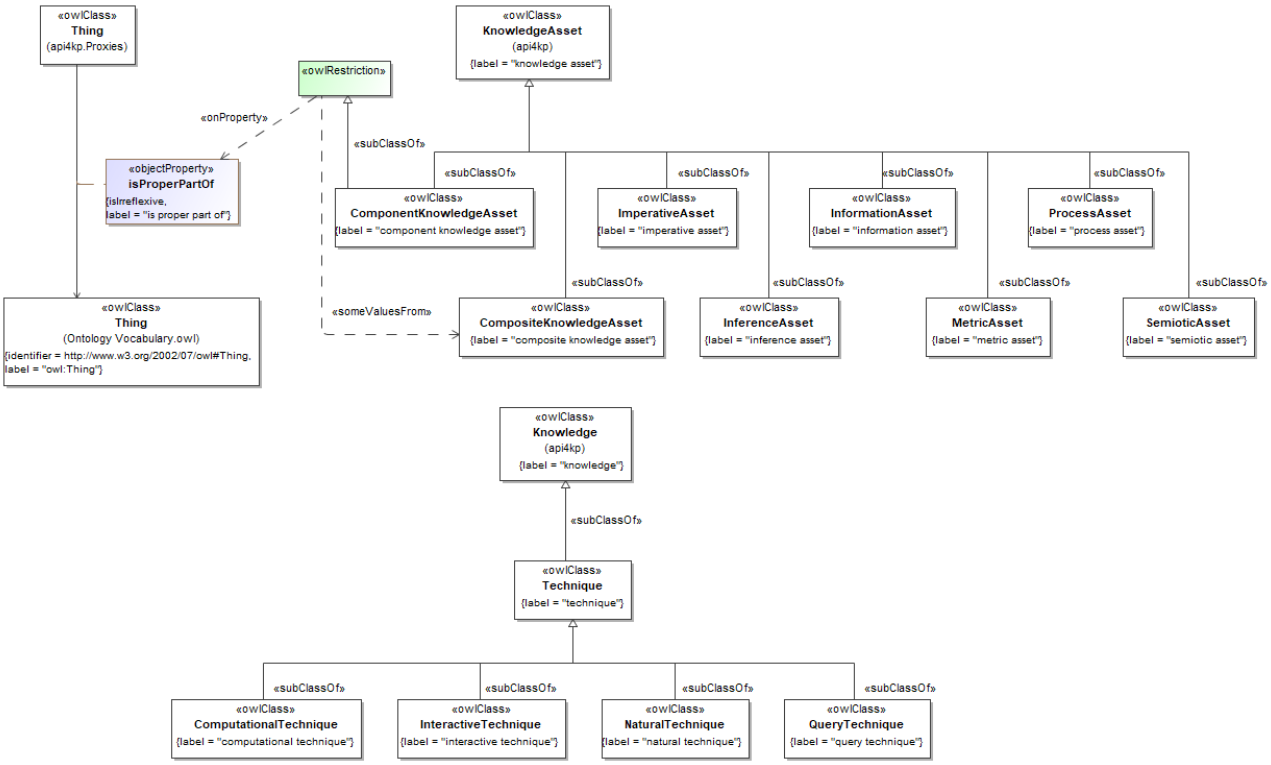

Figure 19: Knowledge Asset Type Class Hierarchy

API4KP Knowledge Platform (KP) Ontology

The Knowledge Platform (KP) Ontology extends the core ontology to specify the nature of a knowledge platform, i.e., a computing environment designed to host reasoners, rule engines and other knowledge processing capable applications, and consume knowledge artifacts, and enables specification of the services that such a platform can provide.

The KP ontology distinguishes between (Knowledge Processing) Software at rest, in the form of source code in some programming language, in transit (packaged for distribution), and in motion - deployed in a runtime environment.
As such, it draws a parallel between "Software Assets" and "Knowledge Assets", where the former can be considered a

narrow specialization of the latter, but also a ‘meta’ layer that uses knowledge *about* knowledge processing to create components that can execute the processing of knowledge.

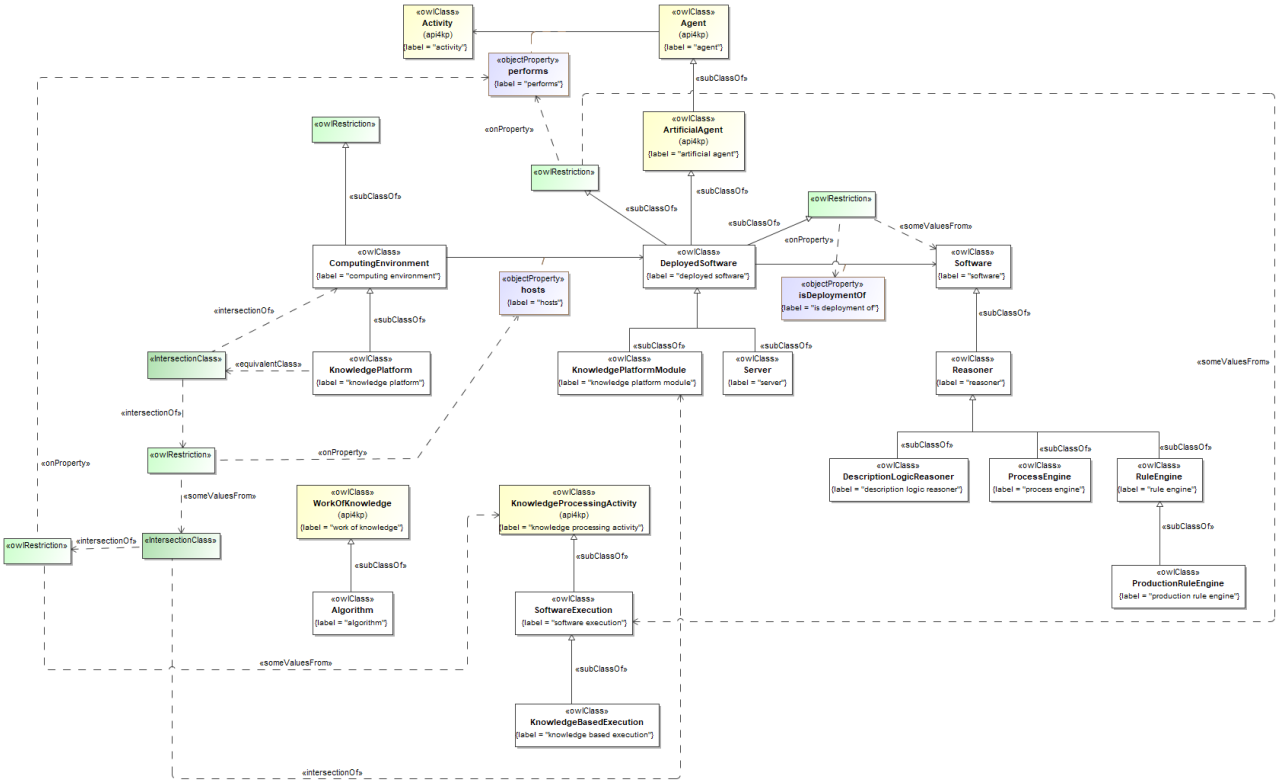

Figure 20: Knowledge Platform Class Hierarchy

API4KP Knowledge Representation and Reasoning (KRR) Ontology

The Knowledge Representation and Reasoning (KRR) Ontology specializes in the core concepts to support environments that provide formal semantics for the operations exposed via the APIs. This ontology builds on several concepts defined in the DOL-terms ontology, providing a tight integration point between the two standards.

In particular, the DOL notion of *institution* – a meta-framework that relates logics, languages, and mappings thereof – is used to scope a Knowledge Platform – a set of Components and Operations that support implementations of an Institution.

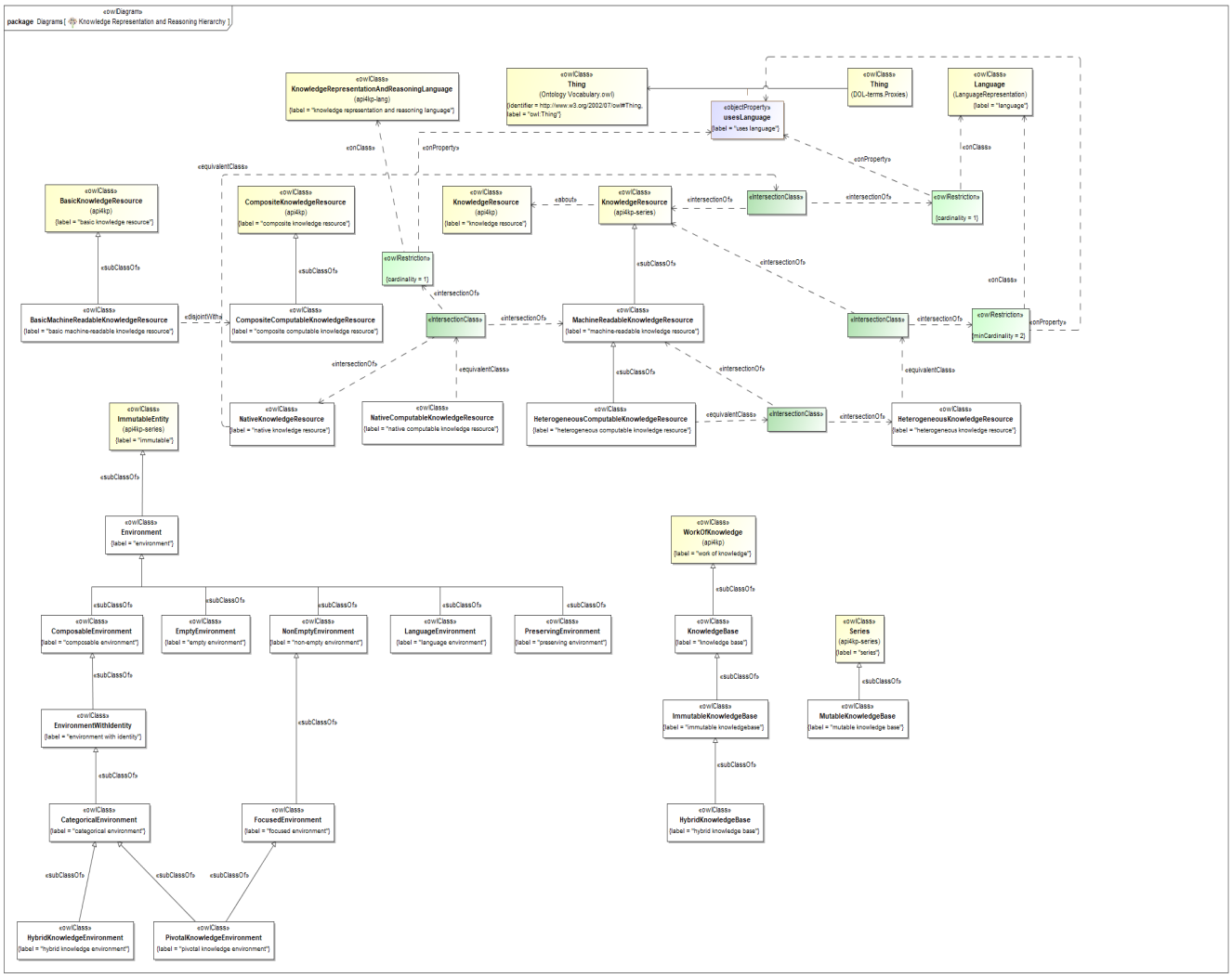

Figure 21: Knowledge Representation and Reasoning Hierarchy

API4KP Language (LANG) Ontology

The Languages (LANG) ontology relates natural languages to formal languages, including, but not limited to, those used for knowledge representation and reasoning. This ontology complements and extends the OMG's LanguageRepresentation (lcc-lr) ontology, from the Languages, Countries, and Codes (LCC) specification. **Error! Reference source not found.**, below, provides a view of how those relationships work.

While LCC-LR focuses on *Natural* Languages, the API4KP ontology focuses on *Constructed* Languages, and in particular on languages that can be described using a Formal Grammar. A Formal Grammar is a Knowledge Asset that has formal semantics – for example, based on constraints or production rules, which is used to define and recognize the sentences of a language. Formal Grammars are usually expressed using a Formal (meta)Language such as the Backus-Naur Format. Notice that being defined using a Formal Grammar is necessary for the language to be Machine-Readable. However, it is not sufficient to infer that any, as opposed to all, Expressions in the Language would have Formal Semantics itself (i.e., to make the Language Machine-Executable). In fact, to have Formal Semantics is a property of the Asset more than its Expression. For example, consider the elementary sentences "all men are mortal" or "2+2=4": both are (semantically) formal, despite not having been formalized. This distinction is primarily important from the perspective of the API4KP Knowledge Reasoning APIs. Because of the possible ambiguity, the use of the term "Formal Language" is discouraged unless absolutely clear from the context of use.

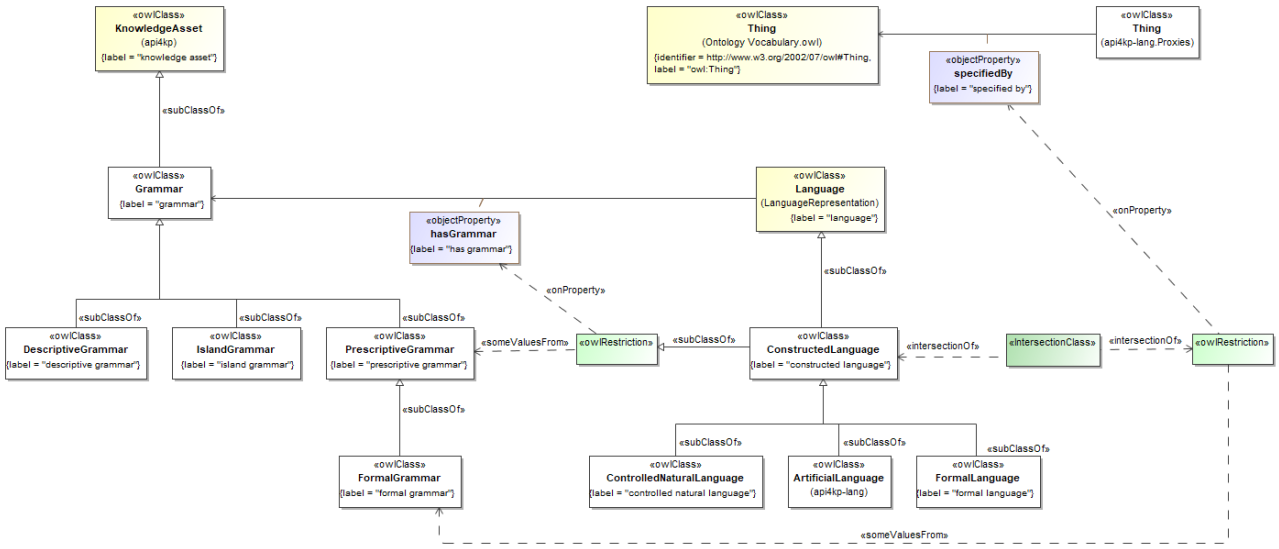

Figure 22: Expressions and Languages

API4KP Ontology of Operations (OPS)

The API4KP Operations (OPS) ontology formalizes the notion of a knowledge processing task, providing semantics for the operations that are exposed by means of the API4KP APIs, including but not limited to access and transformation services.

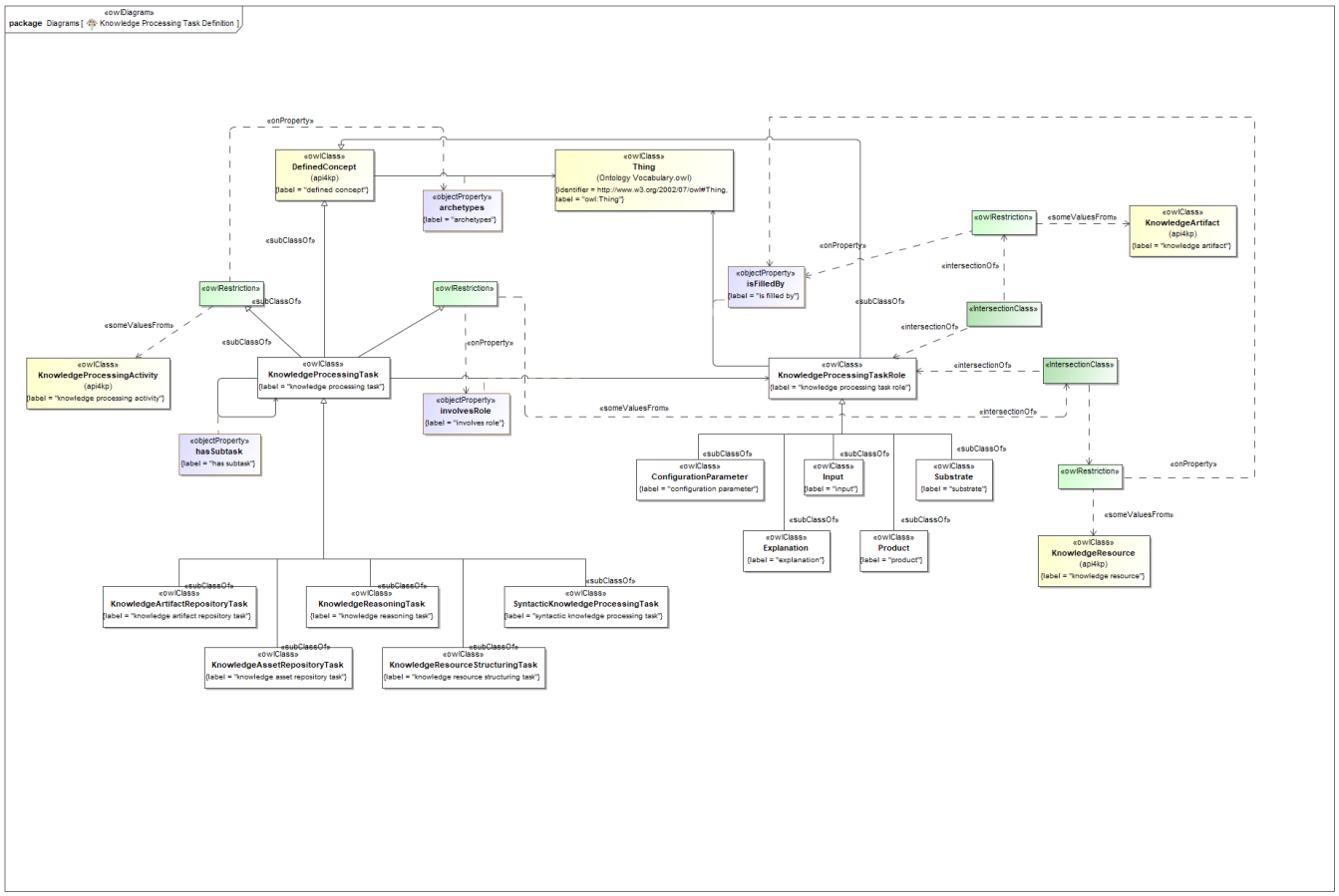

Figure 23: Knowledge Processing Task Definition

In API4KP, a “Task” is the abstract, conceptual counterpart of an Activity. Instances of the former are intensional and definitional, while instances of the latter are extensional occurrences that realize the former.

For example, the class of “Translation Tasks” correspond to the set of all Tasks that involve a mapping between two languages, such as ‘OWL-2 to Common Logic Translation Task (based on a specific mapping)’. The Task is then realized every time an Operator performs an (instance/occurrence of) Translation Activity, applying the mapping to a source Artifact e.g., in OWL-2 to generate a specific Artifact e.g., in Common Logic.

Error! Reference source not found., above, sets out the relationships between knowledge processing tasks and the roles that various resources play in those tasks. The conceptual hierarchy of processing tasks is shown below in **Error! Reference source not found.**.

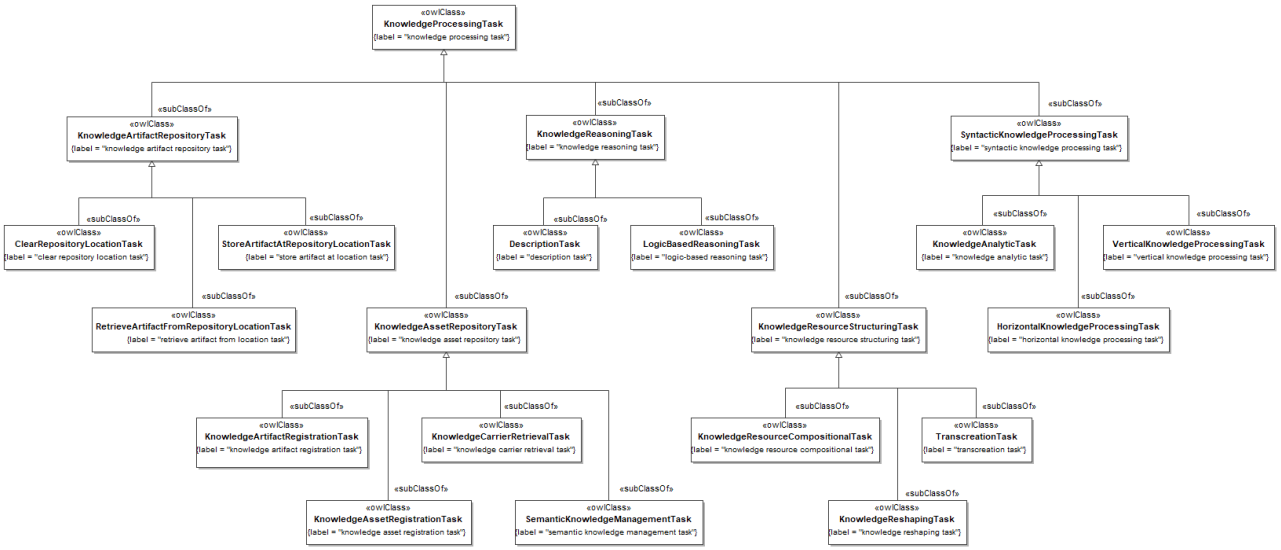

Figure 24: Knowledge Processing Task Hierarchy

The hierarchy is organized around the various API4KP Services – Repository, Syntactic Transrepresentation, Semantic Manipulation and Pragmatic Reasoning: in particular, the leaf classes are aligned with the API4KP operations described in Section 7.

API4KP Relations (REL) Ontology

The API4KP Relations (REL) ontology focuses on relationships between knowledge resources. These relationships are reused in a number of the subordinate ontologies and their usage is further exemplified in the set of informative ontologies that demonstrate how these ontologies can be used in an application environment. The ontology does NOT cover other relationships between knowledge resources and other Things such as concepts ('aboutness') or activities/agents that impacted the lifecycle of the resource. The upper concepts of the REL ontology are composition, derivation, variance, versioning, and dependency.

Versioning allows us to control mutability (and thus reproducibility) across chains of operations, even if (specific versions of) knowledge resources are considered immutable; Composition and Dependency impact the Knowledge Base Construction operations, defining tight and loose couplings between Components; Derivation and Variance are asserted as a consequence of Transrepresentation Operations, depending on an operation's characteristic of (not) preserving a Knowledge Asset while manipulating its Carrier.

API4KP Series (SERIES) Ontology

The API4KP Series (SERIES) ontology extends the core API4KP ontology to incorporate notions of snapshots and versions of knowledge artifacts as they change over time.

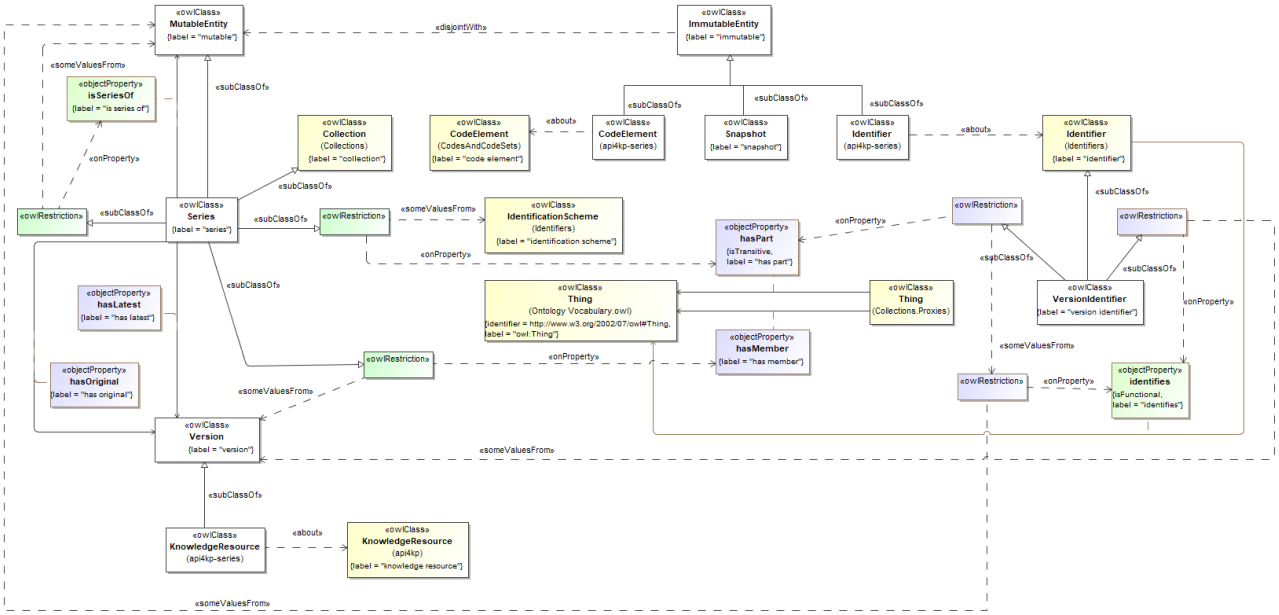

Figure 25: Series Ontology Overview

This page intentionally left blank.

Annex B: API4KP Knowledge Architecture (informative)

This Annex provides background and insight on the practical applications of the concepts defined in the API4KP ontologies, providing insights on the relevance and scope of the related operations.

B.1 Knowledge Artifacts 'as Software'

Knowledge Artifacts are Carriers of Knowledge Assets, and Knowledge Reasoning APIs expose the behavior of Components that process those Knowledge Assets. In between the Acquisition – whether by Knowledge Representation or Machine Learning – of the Assets and its Execution, the Knowledge is Stored as a binary Artifact, and exchanged as an Expression.

This process is similar – and arguably a generalization – of the common workflows adopted in the development, distribution, deployment, and execution of traditional 'Software' – imperative algorithmic Knowledge expressed in a Programming Language. Knowledge Assets are usually expressed more declarative languages, and executed by means of dedicated virtual machines optimized around the formal nature of the Assets.

The analogy may drive the use, and possibly facilitate the adoption/integration, of API4KP compliant interfaces:

- Artifact Repository → Software Repository
- Asset Repository → Package Management System
- Transrepresentation → Validators, Compilers / Transpilers
- Knowledge Base Construction → Assemblers
- Reasoning → Interpreters

As a consequence, the implementation of API4KP components should leverage well known notions from the theory of compilers, in parallel or addition to the foundations of 'reasoning' algorithms such as RETE (for production rules) or Tableaux (for description logics). In particular, the former become a key element of an API4KP architecture when dealing with hybrid platforms, where there is a many-to-many relationships between Assets, Languages used for the Expression, and Components used for the Execution thereof.

The primary distinctive element of a Knowledge Artifact, besides its specific Asset content, is the Language used in the embodied expression. Knowledge Artifacts that are the product of explicit Knowledge Representation endeavors have formal semantics, and are expressed using formal languages, making the Artifact machine ***readable*** and ***executable***. The choice of language may correlate to the formal type of the Asset, thus determining what kind of operations can be performed with it.

Two additional dimensions involve ***portability*** and ***shareability***. Portability, the ability to execute a Knowledge Asset across different Platforms, depends on the ability of the Execution environment of providing valid, meaningful values for the Asset's open variables – inputs and parameters –, as well as the ability to resolve dependencies, both early and late bound. Dependencies are discussed in further detail in Appendix B.2

Shareability, the ability to *lift* a Knowledge Artifact and interpret its Asset content with equivalent results across different platforms, depends on whether the different environments share a common ontology, as well as a common language. Grammars (abstract and concrete) are necessary to recognize the patterns and the structure of the Expressions, enabling the mapping of the syntactic constructs to the proper concepts ("interpretation"). Formal concepts are usually expressed by *keywords* of the language, while domain concepts are expressed using terms from a Vocabulary which must be bound to the common ontology. Furthermore, to enable communication on a (digital) medium, the sentences that constitute the Expression must be serialized using characters from a known Alphabet and encoded – usually in binary form.

For this reason, mereologic syntactic categories (grammar), topologic patterns (serialization), terminology (vocabulary), symbols (alphabet) and encodings are all syntactic constructs that must be acknowledged in order to lift a Knowledge expression and interpret it as a Knowledge Asset. These elements constitute the 'core syntactic metadata' used in the KnowledgeCarrier wrappers used by the APIs and correspond to the 'vertical' parsing levels also stated in the KnowledgeCarrier and used by the Operations.

This perspective is further complicated when certain categories of Languages are involved:

- **Embedded Languages**
  Multi-lingual expressions are Expressions that use more than one Language for different Fragments. It is common for one Language to be the primary language, and the Expression would be parsed according to the Grammar of that Language. This main Grammar would then support 'Island' sub-languages in specific positions. Expressions in this sub-language become Embedded in another Expression in the primary language.
  Multi-lingual Artifacts are likely Carriers of Complex Assets and may have to be processed as Composites (see Section B.2).
- **Markup Languages**
  Fragments in a markup language wrap, rather than being embedded, Fragments expressed in a different language within the context of the same Artifact. Markup languages can be used to supplement another language, enriching its ability to provide structure (syntactic markup), semantics (annotation markup) or facilitate the recognition (presentation markup) of Expressions in the target language.
- **Meta-Formats**
  (Not) "Languages" such as JSON, XML, YAML, and possibly RDF (but not RDFS!) have grammars which exist *solely* at the concrete syntax level. For example, parsing XML produces a tree whose semantics is completely determined by the interpretation of the element/attribute names and values, which are part of a Vocabulary that is late bound to the specific Expression.
  Notice that *Schemas*, instead, count as Grammars.

B.2 Complex Knowledge Resources

In the simplest of use cases, Knowledge-Based System deal with individual Knowledge Expressions, that realize a single Intellectual Work, having one Piece of Knowledge as subject, serialized using one concrete syntax, and engraved into one Artifact that is an exemplar of a Native Carrier.

Example: a plain, single XML file that contains the XML serialization of one BPMN business process. As a Knowledge Artifact, it carries a Representation that expresses a model (the Intellectual Work) of how a loan approval process works (the Piece of Knowledge). The model is simple enough to be captured by that single BPMN expression carried by that file.

Real scenarios, however, involve more variety and complexity. For example, Knowledge Artifacts that are analogous to 'libraries', 'collections' or 'anthologies' may carry more than one Expression. The same Expression may require two or more Artifacts to be carried, each one carrying a Fragment of the original Expression. An Expression itself may be composite of multiple parts, which are themselves Expressions (or Fragments thereof), realizing some (Complex) Intellectual Work.

In general, there is a many-to-many relationship between Artifact and Expressions, a many-to-one relationship between Knowledge Expressions and (Atomic) Works of Knowledge, and a many-to-many relationship between Knowledge Expressions and Complex Works.

Structuring

API4KP 'Transrepresentation' and 'Knowledge Base' operations allow to (de)construct this complexity. The purpose of this section is to provide guidance on when and why to use the operations.

Within this document, the term ***'aggregation'*** will denote the combination of two or more entities of the same type into a collection thereof. The term ***'union'*** denotes the combination of a collection of two or more entities of the same type into a new, distinct single entity of that type.

The term ***'composition'*** denotes the extension of an entity by means of another entity, which does not need to be of the same type. A composition is ***'complex'*** when the different parts remain identifiable and play a specific role. ***'Injection'*** is the act of combining two composite entities into a single one: the injected entity becomes a part of the entity it is injected into, and the latter becomes an ***'extension'*** of its new component.

Composition is based on a parent/child part-of relationship, while aggregation defines an implicit container of which the elements are member-of, and siblings to each other. Aggregation and Composition are ***'structuring'*** operations: the elements remain separate, but their inter-relationships are described in an additional, separate ***'structure'*** entity, which can be conceptualized by means of a named, directed graph that states the actual relationships.

'Merging' or ***'Fusion'*** denotes the general operation of combining two or more entities into a single entity, in terms of union and/or injection. ***'Assembly'*** or ***'Flattening'*** is the act of merging entities based on the specification provided by a structure. Note that after an assembly is performed, it may or may not be possible to discern the original components.

For each one of these operations, an inverse can be defined. ***'Deconstructing'*** - either by ***'decomposition'*** or ***'disaggregation'*** **-** is the operation by which a structure is superimposed on an entity, to identify proper parts which can be ***separated***. In particular, a collection can be ***partitioned*** into its individual members, while a component can be ***extracted***, ***removing*** it from its composite parent.

'Disassembling' is the act of breaking an entity into a set of smaller entities, based on a destructuring.

Notice that the operations are defined on resources, atomic or not, and can be used recursively. These concepts apply to any Knowledge Endeavor. Because the API4KP specification focuses on Knowledge Expressions, the APIs expose these operations at the Expression level.

Formally[3]:

- **Construct**:
  - **Aggregation**(X_1, ..., X_n) => < { X_1, ..., X_n } >
  - **Composition**(X, Y) => < X[Y] >
- **Fusion**:
  - **Union**(X_1, ..., X_n) => X
  - **Injection**(X, Y) => X[Y]
- **Deconstruct**:
  - **Disaggregation**(X) => < { X_1 ..., X_n } >
  - **Decomposition**(X) => < X[Y] >
- **Separate**:
  - **Partition**({ X_1, ..., X_n }) => X_1, ..., X_n
  - **Extraction**(X[Y]) => X,Y
  - **Removal**(X[Y]) => X[]
- **Flatten**(X_1, …, Y_n,< X_1, ..., Y_n >) => Z
  - = Fusion(*, Structure(*))
- **Disassembly**(X, < X_1, ..., Y_n >) => X_1, ..., Y_n
  - = Separate(X, Deconstruct(X))

In order to understand the principles behind the 'structuring' operations, one has to consider three major elements: (i) tree- vs collection-orientation, (ii) analysis vs synthesis and (iii) inverse operations.

1. Expressions in a knowledge representation language are usually collections of sentences. Sets treat members as individual peers, which can be added or removed with limited syntactic burden. Each sentence, on the other hand, has an internal structure and can be modelled using an (abstract syntax)

[3] {} denotes set aggregation, [] is used for tree composition, and <> for structuring - combinations of trees and/or sets

tree. Because of the mutual role, elements of a composition must be compatible with each other, and even at the syntactic level, a component may only fit specific positions within the parent element.
2. A structure, which can include atomic elements as well as their set/tree sub-structures can be identified / defined without actually (dis)assembling a new Expression. However, the structure may be actually used, sooner or later, as a blueprint.
3. For each operation, an inverse operation is defined. Notice, however, that operations are not functions (e.g., an Expression can be Destructured in multiple ways).

The definition of the 'structuring' operations is transparent with respect to the actual syntax, semantics and pragmatics of the composites and components, but some constraints are imposed, nevertheless.
Set-oriented structures are used for Expressions where multiple 'sentences' (fragments) can be identified: e.g., OWL axioms in an OWL ontology, or RuleML rules in a RuleML rulebase. Likewise, Tree-oriented structures are used to manipulate individual sentences, typically by adding a term to a fragment, or expanding an existing one. Examples include replacing an occurrence of a named class in an OWL axiom with its equivalent class expression according to some ontology; composing a BPMN workflow model with one or more DMN decision models that specify some of the decision tasks in the workflow.

Dependencies

As previously noted, the purpose of a Work of Knowledge (WoK), atomic or complex, is to select and prepare a Piece of Knowledge (PoK) in a way that can be utilized, usually for representation (eventually aimed at communication) and/or reasoning. The atomic PoK is usually called 'concept', stressing it role as an abstraction that depends on some intelligent agent, or 'representational unit', emphasizing its potential role as the subject of a WoK.
Example: The minting of a new term for a Concept is possibly the simplest WoK. The creation of a Definition of a Concept is a WoK. The combination of the two acts is also a WoK.

While it may be possible to agree on an objective definition of atomic PoK, the notion of 'atomic WoK' is more arbitrary and depends on the intent of its creator. It is more interesting to analyze a WoK in terms of the boundary relationships between concepts included in the PoK that is the subject of the WoK, and the concepts that are not, but are ***pre-required*** to understand the WoK itself and internalize the Knowledge it conveys.

A Work of Knowledge is ***Plain*** if it captures a Piece of Knowledge that, in order to be understood by an Agent, pre-requires only Common Knowledge: i.e., knowledge that every agent possesses, and that every Agent can expect other Agents to possess. Otherwise, it is ***Profound***: it pre-requires additional non-trivial, possibly expert, knowledge to be understood completely. A Plain WoK can play the role of ***Elucidation*** if it ***elucidates*** a Profound WoK: that is, their composition is a Lucid Work. In this case, the elucidation creates a bridge between the advanced Concepts in the Cryptic WoK and simpler Concepts.

The 'Common Knowledge' that allows to distinguish between Lucid and Cryptic works can be absolute but is more likely to be scoped by a business domain, leading to the notion of ***Domain-Specific Common Knowledge***, proper of any (***Domain-***) ***Educated*** agent that is capable of conducting business in that domain. Agents that possess Knowledge about a Domain that is not Domain-Specific Common Knowledge are often called ***Subject Matter Experts*** - or SMEs.
Example: the distinction between 'procedures' and 'surgeries' is common knowledge for healthcare workers. A senior surgeon would possess Subject Matter Expertise on the topic.

Sometimes, a WoK is deliberately framed in a way that makes them Plain for SMEs, but Profound for anyone else. Educated Agents may require some kind of Elucidation, but the work may be simply too complex for anyone else without formal education on the subject.

Example: a lecture in an advanced course, that has some prerequisite courses.

This kind of ***Expert Work of Knowledge***, i.e., a WoK that is targeted to SMEs, requiring additional knowledge that is contained not in the WoK itself, is often designed to be more ***portable***, since it leaves more degrees of freedom of interpretation, but may do so at the expense of shareability, since it is generally understandable by a much smaller audience.

Example: a recipe of the traditional Bolognese Lasagna, which takes for granted the recipe of the Bolognese Sauce. A chef versed in the traditional cuisine would be able to follow it, possibly using their own interpretation of the sauce - e.g., using a variation without tomato for people with allergies. A chef that is not an SME would need to find the recipe of the Sauce first. Other agents would go to a 'rosticceria' and buy a tray.

Given the arbitrary nature of the boundaries of a WoK, and that complex multi-part works can be assembled into a single one, we will initially discuss the relationships between a single WoK, its requirements in terms of other PoKs, and the Expressions that realize it.

For every atomic WoK, there is at least one Expression of that WoK in the (Natural) Language of its author. This Expression, whose original Carrier is named '*Manifestation Singleton'* in the FRBR-OO terminology, is generated as an outcome of the initial creative Work. That WoK is based on Concepts (and thus knowledge) that the authors possess, new concepts the author may have defined in terms of more primitive ones, and the overall creative organization that results from the Work. That expression is a Knowledge Resource that can be translated (interpreted and expressed again) into other languages as needed. In particular, the Expression can be destructured and then disassembled into parts, and each part translated into different languages, as long as the structure is preserved. Each part is either an Expression of its own - if it can be considered the expression of some WoK (regardless of the explicit intention of the author) - or a Fragment thereof.

Example: destructuring a narrative, identifying the description of a landscape. Later, the narrative is disassembled, the description is replaced by a drawing (written language -> pictorial language translation), and the resulting composition is reassembled.

Example: decomposing a narrative that expresses the combination of an ECA rule that triggers a business process, and the specification of that business process. The rule is then translated into RuleML, the business process specification into BPMN, and the two resulting Knowledge Expressions are recomposed.

When such a WoK is decomposed and its constituent parts are extracted, it is usually still desirable to maintain the structure explicitly, to preserve the overall expression of WoK itself. A (composite) *Expression* A ***'imports'*** another (component) *Expression* B that resulted from such a separation. More specifically, an *Expression* A ***'includes (by reference)'*** another Resource B if B is explicitly expected to be injected into A. The separation allows for reuse of *Expressions*, so that an author does not to have to express the same knowledge again when it is shared across Works. Import relationships can be distributed and inlined explicitly in the Expressions, but also expressed in a separate 'structure' Resource.

Example: an OWL ontology A that imports a different OWL ontology B.

Example: an empty OWL ontology C whose only purpose is to import the OWL ontologies A and B, which are otherwise unrelated.

Example: a clinical decision support (ECA) rule that imports a cohort definition to limit its applicability to a specific class of patients.

Imports/inclusions preserve the integrity of a WoK even when its Expression is decomposed. In the case of Cryptic WoK, however, the integrity is not guaranteed. For simplicity, we define *Cryptic Expression* a Knowledge Expression that expresses a Cryptic Work of Knowledge. A structure that defines the disassembled Expression of a Work of Knowledge without fully specifying the imports between the components results, when reassembled, into a Cryptic Expression[4]. This scenario can arise when a WoK is originally devised for an expert audience, and the author deliberately chose not to communicate parts of the Work which are taken for granted, or when some parts of the Expression of a Lucid Work are removed.

It is important to remark the distinction between a Representation the author chose to express Knowledge that is part of their original WoK, any other representation that expresses equivalent Knowledge (or the same Knowledge in an

[4]Since the author's WoK is not fully represented, it is not trivial for a consumer Agent to understand the entirety of what the author wanted to communicate, for the simple reason that they would not know when there was anything else to understand.

equivalent way), and an expression that the recipient would be able to incorporate to better understand the Work but is not part of the original Work. In the first scenario, a Representation C would include a named Representation B. In the second, Representation C would import either named Representation C, or any member of a class of Representation B* that meet some criteria. In the third, a Representation C would be used to elucidate C. A Representation C ***requires (depends on)*** either a named Representation B, or any one member of a class thereof, for the purpose of fully expressing the author's WoK. This relationship is defined to support constrained, and/or late-binding compositions. When an actual Representation B is chosen to fill the dependency, C ***builds-upon*** *B.* C *safely* builds-upon B if the actual composition of C and B is coherent and consistent. Notice that, by definition, builds-upon further generalizes the notion of import, which is used to assert static dependencies.

Any Representation that is a candidate to fill a dependency of another Representation C (i.e., a Representation B that is of a type that C and such that C can safely build upon B), or any Representation that can elucidate C is ***compatible*** with that Representation C. Compatible representations used that are structured with C become ***supplements*** or ***complements***. Complementary resources improve the accessibility when available, supplemental resources reduce the accessibility when not available.

Example: A clinical decision support rule recommends immunization against pneumococcal infections for asplenic patients. Based on the scope of the author's WoK, that rule requires some (unspecified) definition of 'asplenic' patient but is compatible with any business process for the choice, scheduling, and administration of an actual immunization procedure. When the particular cohort definition and the process specification are chosen, the rule will build upon them.

Expressions, including Representation Fragments and Structures, are carried by Knowledge Artifacts. An Artifact may carry one or more expressions. If an Artifact carries all the Representations that compose a Structure, the implicit Representation that results from its assembly is *self-contained* within the Artifact. Conversely, if an Artifact A does not carry an imported Representation, which is instead carried by a different Artifact B, then A ***depends*** on B.

More generally, an Artifact A depends on a (singleton) class of Artifacts B when (i) A carries an Expression R that needs a named Expression S carried by B, or (ii) A carries an Expression R that needs am Expression that expresses some (Work of) Knowledge that is required by the (Work of) Knowledge realized by A.
Conversely, an Artifact B is ***linked*** to an Artifact A when B carries an Expression that is used as a supplement or as a complement for some Expression carried by A.

Structures

Structuring is the act of identifying the components of a (Complex) Resource, and their mutual, functional relationships. **Structures** can be either ***synthetic*** or ***analytical***. Analytical structures are identified on pre-existing Resources, while synthetic structures are created in the process of assembling complex Resources. Analytical structures are further classified into ***explicit*** or ***emergent***: the latter are superimposed on a Flat(tened) Resource by means of a de-structuring operation, while the former can be immediately identified as part of a Structured Resource.

Regardless of their origin, Structures are Knowledge Expressions themselves: they can be identified, versioned, and need a language equipped with a concrete syntax in order to be expressed.

One should distinguish between ***descriptive (****aka* ***assertional, extensional)*** and ***constructive (****aka* ***operational, intensional)*** expressions: the former defines what a structure is, while the latter describe how to create one. There is a dual relationship between the two approaches: the assertion of a relationship between two Resources in a descriptive structure can be considered the product of the execution of an operation prescribed by a constructive representation of the same structure.

Example: A descriptive representation of a structure, conceptualized as a graph of dependencies between resources, can be expressed using RDF, leveraging some of the relationships in the API4KP FRKR ontology.

Example: A constructive representation of a structure can be expressed using the OMG DOL language.

Any language used to express a Structure must, at a minimum, support (versioned) identifiers and either relationships or relator operations. A Structure is (i) ***closed*** if every component Resource can be identified and resolved univocally, and (ii) ***deterministic*** if it is closed and the assembling of those components is functional, resulting in one and only one Resource.

In particular, openness may arise from:

- *Openness by version*: A structure that identifies the components, but not their specific version. Instead, versions are either not specified, or specified in terms of a version interval, bounded or not.
  Example: An OWL ontology o_1 that imports the latest version of the OWL ontology http://omg.org/spec/api4kp/FRBR-KR
- *Openness by resolution:* A structure that denotes particular components by means of references that are not identifiers, such as names.
  Example: An OWL ontology o_1 that imports the API4KP FRKR ontology.
- *Openness by definition:* A structure where relationships/operations are not referencing particular Resources, but classes thereof, defined intensionally
  Example: An OWL ontology o_1 that imports any other one ontology O_2, such that "O_2 defines the notion of Knowledge Resource", of which the API4KP FRKR ontology is a fitting candidate.

B.3 Identification and Versioning

Identification

Identifiers

The API4KP specification uses URIs as identifiers. URIs are commonly supported even beyond the scope of the (Semantic) Web, and most identifier schemes can be cast into some URI forms. Implementation might substitute other identifiers as long as they support the following properties. That is, an Identifier *MUST* support

- **Universal Scope**: Identifiers must be globally unique.
- **Namespace support**: Identifiers must be decomposable into the combination of a namespace and a locally scoped identifier.
- **Support for Fragments**: Fragments are used as 'anchor points' to identify parts of an identified entity that are not entities themselves, and as such are not independently identifiable.
  Example: Knowledge Fragments within an Expression carried by an Artifact.
- **Decomposable Versioning**: Identifiers must support the identification of an entity within a series. (See Versioning)

An Identifier *SHOULD* also support the following:

- **Uniqueness:** Each entity should have one canonical Identifier, even if an entity is allowed to have multiple Identifiers. An Identifier *MUST* still denote at most one entity and *SHOULD* denote exactly one entity.
- **Transparency:** URIs can be transparent or opaque, even if evidence recommends the use of at least one opaque URIs [add refs].
  (A transparent Identifier is such that information about the denoted entity can be inferred from the structure of the Identifier. Example: ex: person-123 likely denotes a person.)
- **Persistence**: Identifiers that have been assigned to an entity should never be retired.
  Identifiers of 'social' entities, such as a Knowledge Asset, *MUST* be persistent.

Identifiables

Every entity in API4KP is identified by means of a URI. In particular, Intellectual Works, Expressions (abstract and concrete) and Carrier Artifacts are each assigned a persistent identifier. Even if the identifier of an abstract entity (e.g., a Piece of Knowledge) cannot be dereferenced to the entity itself, it can be associated to the *concept* of that entity,

metadata about that entity, or used to establish relationships to/with other entities. For example, the URI of a Piece of Work is used to correlate different Expressions of the same Work, and to reference semantic metadata that describes the subject of the Work.

It is important to remark that even abstract entities such as Intellectual Works must be identified. The distinction is important, for example, to be able to distinguish between an algorithm and its implementations (e.g., in Java) across copies of the source code, or to express the fact that an OWL ontology and a Common Logic theory are different formalizations of the *same* ontology.

Identification

Each entity type shall be equipped with a principle of identity and, optionally, a principle of equality.

(The actual choice of principles is beyond the scope of this specification. It is noted that the matter is a subject for the debate in philosophy as much as in computer and information science. As such, the following definitions should be considered non-normative.)

In general, each entity (type) possesses essential ('genotypical') qualities that remain unchanged from the moment the entity is created until its destruction, and contingent ('phenotypical') qualities that may change over time, resulting in a *new version* of the *same entity*. Furthermore, qualities can be *observable* if it possible to determine their value ('quale') for a given individual entity. The *type* of an entity is associated to a set of essential qualities (and behaviors) that an entity must possess. The exact relationship between type and qualities varies: for example, 'duck typing' goes as far as stating that the type is *defined* by a set of qualities.

An entity comes into existence at a given point in space and time, in time remains the ***sameAs*** itself across all the time its essential qualities - including its type - remain unaltered and is destroyed as soon as any one of them changes. An entity is (perceived to be) ***identical*** to another entity if they share the same type and all its (observable) qualities have the same values. An entity is identical to the same version of itself, and two distinct entities, i.e., two individuals that are not the same, can still be identical for as long as they share all the same property values, but can occupy different portions of space and time. Consider for example, twins, or copies of the same book. Furthermore, two entities are ***equal***, according to some criteria, is a certain common subset of their (observable) qualities share the same values. Two entities are ***equivalent*** (to a degree) if there exist a (non-boolean) criterion that allows to determine whether two entities are equivalent or not, and that criterion holds true for that pair of entities. Notice that equivalence is not necessarily based only on the qualities of those entities. Finally, two entities are considered ***equipollent*** if they can be substituted (*i.e.*, they can be used interchangeably) in some activity to yield the same effect.

Example: based on these principles, a file that carries an OWL/TTL ontology remains the same until it is deleted and is (byte-wise) identical to itself all its copies until it is changed and saved onto itself, resulting in a new version. It is equal to any file with the same content that uses a different encoding, it is equivalent to any other concrete expression of the same ontology (e.g., an OWL/RDF file that results from a transcription) and is equipollent to an exact translation of the same logical theory in a different language (e.g., Common Logic) for the purpose of performing inferences.

Analogous definitions could be provided focusing on Knowledge Assets (using the AST as a focus), or even a Piece of Work, even if the criteria tend to be less objective.

Versioning and Series

Any Knowledge Endeavor in API4KP - including the software that implements the components of a Knowledge Platform - is expected to be an entity that is identified and versioned. Given identity and equality principles, the relationship ***next*** (and its inverse ***previous***) will be used to denote the relationship between the *same* entity between two states such that the entity is no longer *identical* to itself. More generally, the relationship ***later*** (vs ***earlier***) will denote two non-contiguous states. The ***original*** version of an entity is such that there is no previous version, and the ***latest*** is such that there is no next version. A ***snapshot*** is the version of an entity as of a particular point in time, as opposed to being defined. All the versions of the same entity, over time, form a ***Series***.

Based on these relationships, versioning shall follow the Memento pattern [http://mementoweb.org/guide/howto].

- The same entity, across its version, will be attributed a generic URI
- Each version shall have a URI that can be uniquely mapped to the generic URI.
    - The version URI will introduce a single component that uniquely denotes the version, in a predictable position in the URI structure: the generic URI can then be obtained by removing that component from the version URI.
- The version-specific part of the identifier can be implemented using a variety of strategies: incremental numbers, semantic versioning, timestamp-based, etc.
    - Version identifiers v() for the same entity should respect a linear ordering > defined on the space of identifiers. That is, if B is the next (version of) A, it should be the case that v(B) > v(A)
    - Snapshots should use timestamp-based version identifier.

Versioning within Structures

Structures define how atomic Knowledge Representations are aggregated and composed together. A structure is a defined in terms of a typed, directed graph that asserts the specific dependencies between the Representations. Specifically:

- Each node in the graph is labelled with either
    - the URI of a Representation (as a whole)
    - the URI of a specific version of a Representation
    - the URI of a fragment within a specific version of a Representation
- Each edge in the graph is labelled with the URI that denotes the specific semantic relationship between (two fragments of) two (specific versions of two) Representations.

As a Resource itself, the Structure has a version URI. Every time a structure is modified, e.g., because of the application of a structuring operation that is used for the incremental construction/modification of a complex Resource, the version of the Structure will be incremented.
This approach is required to decouple the evolution of complex Expressions from the evolution of its components. The fact that a certain version of a Resource could be combined with others does not guarantee in general that a newer version of that Resource can safely be combined as well. Instead, it is likely that the other components of a resource would need to be revised (and possibly updated).

B.4 Derivation

Derivation is a general relationship that holds between two entities and, in particular, two knowledge endeavors. An entity ***derives*** from another entity if the former is the output of a (creative) activity that has the former as one of its inputs.

Versioning almost always implies derivation: a newer version is usually somewhat influenced by the previous one. However, versioning emphasizes the act of retiring an endeavor, and providing a new(er) one that should be used in place of its predecessor. Derivation, instead, focuses on the kind of activity that led to the generation of the new endeavor, regardless of whether it is intended to replace another endeavor or not. More importantly, a next version of an entity has one prior entity but can be the derivative of several other entities.

In the FRBR conceptual model that inspired the API4KP concepts, Derivation is further categorized according to two criteria: level of abstraction (work vs expression) and preservation (or not), leading to three categories:

1. R2R derivation of a new Expression of the same WoK
2. W2R derivation of a new Expression of a new WoK from the Expression of a different WoK
3. W2W derivation of a new WoK from an existing WoK

In API4KP, a slightly different categorization is followed.

Intra-institution trans-representations (R2R, WoK preserving)

An important sub-category is composed by derivations induced by operations that leverage (almost) exact mappings between languages within the same *Institution*. Depending on what level the mapping is applied to, a Resource ***is transliteration of, is transcription of,*** or ***is translation of*** another Resource if the change affects, respectively, the tokens, the parse tree, or the AST.

Linguistic manipulations (R2R, WoK preserving)

However, not all WoK-preserving derivations need to be based on a mapping between languages, or elements thereof.

A Resource ***is revision of*** another Resource if any element of its parse tree is altered in a way that (is intended to) reduce the cost and/or the likelihood of errors in the act of abstracting its WoK content. Revised expressions are often released as next versions of a resource.

A Resource R_2 ***is abridgement of*** another Resource R_1 if they have equivalent ASTs (i.e., parse(R_1) = parse(R_2)), but the parse tree of R_2 is a subtree of the parse tree of R_1.
OR
if the AST of R_2 is a subtree of the AST of R_1, but both can be abstracted to equivalent WoKs: abstract(R_1) = abstract(R_2).

A Resource R_2, ***is (re)arrangement of*** a Resource R_1 if they have different structures, but can be flattened to yield equivalent Expressions: flatten(R_1) = flatten(R_2)

Content manipulations (non-WoK preserving)

The last category of interest involves derivations that do not preserve the underlying WoK. This kind of derivation implies the abstraction of the original resource, the generation of a new WoK, and the expression of this resulting WoK into a new, derived Resource. As such, when these relationships are asserted between Resources, they reflect underlying relationships between the respective WoKs that the conceptualize the two expressions.

An (expression of a) WoK W_2 ***is summarization of*** an (expression of) a WoK W_1 if W_2 is a subgraph of W_1, but W_2 entails W_1, so that W_1 can be reconstructed by inference.

An (expression of a) WoK W_2 ***is paraphrase of*** (an expresison of) a WoK W_1 if the two graphs have different node (concepts) and edges (relationships thereof), but W_1 entails W_2 and vice versa. In particular, W_2 ***is linguistic adaptation of*** W_1 if the particular combination of concepts used by W_2 facilitates its expression in some target language, e.g., because the language would not have symbols to express some of the concepts used in W_1.

A WoK W_2 ***is inspired by*** a WoK W_1 if their respective graphs are similar enough according to some criteria.
If the similarity exceeds some threshold, W_2 may be considered an ***imitation of*** W_1.

Finally, a WoK W_2 ***is transcreation of*** a WoK W_1 if W_2 is inspired by W_1, and the boundary concepts of W_2's graph are close(r) to concepts that can be considered background knowledge for expressions of W_2, whereas W_1 would not. Transcreation is another kind of ***adaptation***.

B.5 Examples

B.5.1 Composite Asset with Semantic Versioning

Consider the following scenario. A healthcare SME devises a rule to help manage patients on anticoagulant therapy. Based on an estimate of a patient's probability of suffering from a stroke, as opposed to bleeding, criteria based on a risk/benefit analysis are used to make recommendations on how to adjust the dosing of the drugs.

A Knowledge Engineer working with the SME observes that this Work of Knowledge can be decomposed into multiple parts: the rule's precondition uses a patient cohort definition ('patient on anticoagulants'), two predictive models ('risk of stroke', 'risk of bleed'), a decision model ('what is the most effective dose?'), drug-related knowledge ('how anticoagulants work'), and everything relies on a common domain terminology and its underlying ontology.
A (written) natural language expression of the WoK would look like : *'If a patient is on an anticoagulation therapy, and their risk of bleeding is greater than their risk of stroking, then reduce the dose of anticoagulant as appropriate'*.

The Knowledge Engineer establishes that four Resources should be created: An OMG PRR expression with OCL as a constraint language, to express the rule and the cohort definition, treated as a Fragment; two DMG PMML scorecard predictive models; one OMG DMN decision model with FEEL fragments. The terminology, provided by a SKOS concept set based on an underlying OWL ontology, is taken as background knowledge. Because of this decision[5], the Rule is Profound until the terminology/ontology is referenced as a dependency or injected into the expression.
The Structure is a heterogeneous composition: Rule/prr/ocl[[PM/pmml], [PM/pmml], [DM/dmn/feel]].

Each element is assigned an identifier. Identifiers are minted using a variation of the Semantic versioning strategy [http://semver.org/]. Major version numbers are used to identify the WoK: the number is then incremented every time the SME revises the work in a way that requires the Resources to be revised. The increments in the minor and patch version numbers, instead, reflect the effort of the Knowledge Engineer evolving, improving, and fixing the resources while trying to create more effective and faithful representations of the SME's WoK.

Assuming the rule is a new work, and every Resource is the first attempt to express that work, every Resource - the rule, the two predictive models and the decision model - is assigned version number "1.0".
In particular, the identifiers of each resource are minted to be: ex:rule:ac/1.0, ex:pm:stroke/1.0, ex:pm:bleed/1.0 and ex:dm:ac_dose/1.0
The series IDs associated to the Resource series across their versions can be identified deterministically by removing the version number component from the URI.

The nature of the rule is such that the predictive models and the decision model are injected into specific points. The structure graph would contain edges such as: <ex:rule:ac/1.0#bleed imports ex:pm:bleed/1.0>
Depending on the expressivity of the languages, the structure itself could be expressed explicitly as a RDF graph (and get its own versioned identifier ex:struct:ac/1.0), or be implicitly determined by 'import'-like fragments in the individual expressions.

At some point, the drug dosing recommendations are updated by some professional society. The decision model - the part of the work that deals with drug dosing - is revised by an SME. Changes are significant enough to mandate a new representation, which is assigned id ex:dm:ac_dose/2.0
Around the same time, the knowledge engineer decides to fix a few minor bugs in the 'bleed' predictive model, creating version ex:pm:bleed/1.0.1
The currently released version of the complex is still ex:struct:ac/1.0, relying on version 1.0 of each component.
A joint effort by the SME and the knowledge engineer establishes that the new dosing algorithm and predictive model still fit the intent, and serve the purpose of making anticoagulant recommendations, but improve its effectiveness. As such, the composite as a whole cannot be considered a new WoK, but a revision of the existing one.
Hence, the (expression of) the complex WoK is revised to version ex:struct:ac/1.1, as opposed to version 2.0. This version 1.1 of the complex expression is based on the original rule, ex:rule:ac/1.0, but now it is pointing to ex:pm:stroke/1.0, ex:pm:bleed/1.0.1 and ex:dm:ac_dose/2.0.
Users will now be able to choose between ex:struct:ac/1.0 and ex:struct:ac/1.1
The version agnostic generic URI ex:struct:ac will resolve to ex:struct:ac/1.1.

[5] In a real-world example, terms are usually expressed by means of URIs or QNames that are resolvable into well-known ontologies.

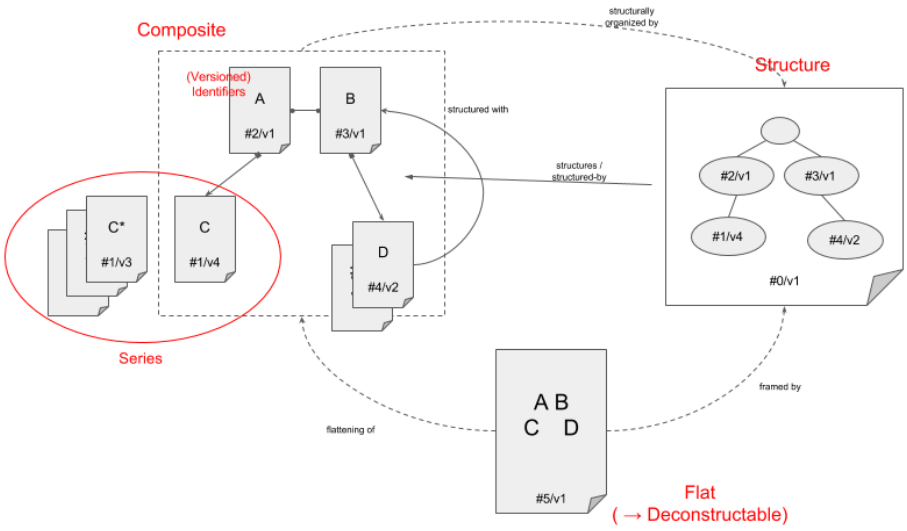

Figure 26: Composite Knowledge Assets

B.5.2 Semantic Decomposition and Classification

Example: The HL7 'Documentation Template' is a profile of the HL7 KNART notation - a UML-based notation comparable to OMG PRR with an XML-based concrete syntax defined by an XSD grammar.
It allows to express sets of 'documentation items', works of knowledge that conceptualize the notion of a 'question' used to elicit some useful piece of information. Optionally, a documentation item can also specify the admissible answers, and how they should be expressed. Additionally, a documentation item can include some business logic (rules) to predict and/or validate the answer in a given context.
Formally, the documentation template can be decomposed as an aggregate of items, and an item is a resource that can be decomposed into its question/answer primary resource, and its optional rule fragments (rules are tightly coupled to the question, so they are not usually considered resources themselves). From a logic perspective, a documentation item can be formalized using a combination of erotetic logic (for the question/answer component) and production rules and/or constraints (for the validation and/or prediction component).
KNART is a flat model which does not have formal semantics, but a decomposition could be superimposed, and a given documentation template with rules could look as follows:

S_1 : Template[{ Item1[{ Rule1a, Rule1b }], Item2[] }, Rule0]

In fact, a future version of the KNART notation might become structured explicitly and allow the use of 'pluggable' sub-languages for the expression of business rules within a documentation item, or even within a template.
Regardless of an explicit decomposition, one may define a general class of KNART Documentation Items, and a subclass of 'Smart' KNART Documentation Items that are explicitly known to contain business rules. Knowing whether a Documentation Template contains business rules or not is critical because of the different nature of the underlying logic aspects, which possibly require different types of knowledge bases and/or reasoners.

An explicit structure allows for a deterministic classification, whereas a flat model requires a decomposition step which may be non-deterministic. In particular, the decomposition may be able to identify the presence of certain components, but not the exact relationships between the parts. However, this non-determinism may be pragmatically irrelevant: in other words, it could be possible to define 'paraphrasing' operators that map one possible decomposition to another possible decomposition while still remaining within a class of equivalent (or at least equipollent) expressions. In the mentioned example, an equipollent decomposition could look as follows:

S_2 : Template[{ Item1, Item2 }, { Rule0, Rule1a, Rule1b }]

If rules can be attributed to items explicitly, the transformation between the two structures is obvious, and is biunivocal. if not, there is a *-to-one relationship between S_1 and S_2. If the class of 'Smart Templates' is defined on S_2 , the presence of a rule - regardless of its role and the Item it affects - is sufficient. If S_1 can be superimposed, one could define 'Smart Items', and Smart Templates as aggregates that contain at least one Smart Item.

Summarizing, strong definitions based on an explicit structure would be:

- Template(R) <= *struct*(R) ~ { Item+ }
- Smart Template(R) <= *struct*(R) ~ { Item+, Rule+ }
- Smart Template(R) <= *struct*(R) ~ { Smart Item+ }
    - Smart Item(I) <= *struct*(I) ~ Item[Rule+]

The weak definitions would be:

- Smart Template(R) <=
  exists M : *map*(*decompose*(R), M) ~ { Item+, Rule+ }

This page intentionally left blank.

Annex C: Use Cases (informative)

C.1 Generic Criminal Legal System

with input from (http://en.wikipedia.org/wiki/Italian_Criminal_Procedure#Parties)

Actors Parties may have agents acting on their behalf, and these agents may be restricted in their access to the KB to some subset of the authority of the Party. Parties include: Judges (may be different depending on stage of proceedings), Suspect, Defendant, Prosecutor, Police, Injured party, Civilly liable party (to pay damages and/or fines), Counsel / Lawyers for suspect/defendant, Witnesses, Experts, Court, Jury, Legislature.

Actions can be roughly categorized based on CRUD (Create/Read/Update/Delete). Note that the difference between Create and Update is a function of the modularity of the KB. In a highly modular architecture, a new knowledgebase-module may be created when a law is passed, when an investigation is opened, etc.. In a less modular architecture, these actions may be Updates rather than Creates.

1. (*CRU) pass, modify and annul *laws* - Legislature
2. (*R) query (including semantic query) to legal KB for details of the legal code - General Public
3. (*CRU) maintain records of investigations - Prosecutor, Judges, Defendant, Counsel, Police
4. (CRU) initiate proceedings - Prosecutor
5. (CRU) call a hearing - Judge
6. (*CRU) file requests (authorization to conduct investigations, such as wiretapping) - Prosecutor
7. (*CRU) issue an order (e.g., authorizing investigations), *with explanation* - Judge
8. (CRU) appeal an order - Prosecutor, Counsel
9. (RU) drop charges - Prosecutor
10. (CRU) proceed to trial - Judge, Prosecutor
11. (*CRU) file a brief - Counsel
12. (CRU) summon witness or expert - Prosecutor, Counsel
13. (*CRU) provide testimony or expert judgement - Witness, Expert, Defendant, Injured Party
14. (*RU) convict/acquit, *with explanation* - Judge, Jury
15. (*CRU) issue a sentence - Judge
16. (CRU) appeal a conviction - Prosecutor, Counsel
17. (*RU) reverse, amend or quash a decision, *with explanation* - Judge
18. (UD) delete/expunge records - Court
19. (CRU) manage records on payment of penalties, imprisonment, etc - Court

Those items with * are the services that go beyond the capability of an ordinary database, requiring encoding of natural language texts into a knowledge representation language and performing specialized actions, such as parsing or (semantic) querying, on that encoding.

C.2 Connected Patient

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) 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, . . .). 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 physical distributed infrastructure allows many variations (partitioning, replication, caching and proxying, 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.

C.3 Semantic Workflow Models

Business Process Specifications Languages (e.g., BPMN, CMMN) are commonly used to model workflows with different degrees of imperativeness and prescriptiveness. In the case of knowledge-intensive domains, such as healthcare or finance, decision modelling languages (e.g., DMN) are used to represent those sub-activities that are cognitive in nature, as opposed to being manual tasks or nested sub-processes. Decision modelling generalizes, and wraps, the use of decision support knowledge represented e.g., in the forms of business rules or predictive models, for which dedicated languages exist (e.g., RuleML, PMML). With the exception of RuleML, none of the cited languages is formal in the mathematical sense of the term. Such additional semantics can be conveyed externally, e.g., providing a first order logic theory that equips the tasks in a process with action semantics. All the knowledge resources also require an underlying common ontology that defines and relates all the non-logical terms.

C.4 Knowledge Management and Delivery Platform

The core of a Knowledge Management and Delivery Platform is a Semantic Repository that facilitates the organization, retrieval, and delivery of one or more collections of knowledge resources. Semantic platforms of this class extend the notion of 'Semantic Content Management System' in several ways. Each catalogued asset is accompanied by a Knowledge Surrogate, represented by an instance of a semantic information model - i.e., a schema such that each element (class, attribute, relationship, admissible values) is defined in terms of an corresponding ontology. At a minimum, the 'surrogate' will contain information about the content of an asset, its representation, and its relationship to other assets, but are likely to also include 'meta-knowledge' such as versioning, workflow, provenance/pedigree, applicability, usage and/or rights information. The surrogates form an assertional knowledge base that can be queried or used for other inferential tasks. Workflow models can be superimposed to manage the lifecycle of the assets. Business rules and other forms of analytics can be used to assist the work of knowledge engineers maintaining the knowledge assets. In terms of delivery, various facades can be created for different user (roles), such that translations and/or structuring operations are executed on the fly to deliver 'precision' representations on demand, even when what the client perceives does not correspond to what is actually materialized in the repository.

C.5 Ontology-Driven Terminology Systems

Terminology servers (e.g., CTS-2, FHIR compliant ones) expose operations that can be formalized in terms of knowledge base operations on a specific class of knowledge resources. In fact, it is not uncommon to see server implementations based on RDF / OWL / SPARQL approaches. A terminology system is an assertional Knowledge

Base, usually formalized using an OWL A-box based on the SKOS ontology, an OWL T-box, or a combination of both based on the more recent OntoLex model. Searching and browsing a terminology system, and retrieving detail information about a particular concept, are query operations on such a knowledge base. (Intensional) 'Valuesets', likewise, are named queries promoted to the status of knowledge asset so that they can be managed and shared. Valueset expansion and membership check operations rely on the ability to expand the query and/or check the consistency of the assertion that a certain concept belongs to a broader class. Finally, testing two concepts for subsumption is a special case of classification.

C.6 'Discovery' Platform

'Machine Learning' is a term used to encompass a number of techniques that allow a machine to build Knowledge out of Data. As such, Machine Learning activities produce Knowledge Resources that can be constructed into Knowledge Bases and used to make inferences. More specifically, such models are often used to for pattern recognition, classification and/or prediction tasks. PMML is one of the standard languages to express several kinds of such 'learnable' (aka 'discoverable') Resources, and it emphasizes the analytical/statistical/quantitative nature of the models. Models can be assimilated to functions that compute an Output inferred set of features Y based on (i) an Input set of features X, an optional internal state S, and a set of Parameters P. The input represents the contingent facts, while the parameters represent the long-term Knowledge. The State, which is only typical of recurrent (aka 'closed loop') models represent some accumulated additional knowledge that depends on the execution history.

In API4KP, such models are assertional Knowledge Resources, and common functionalities can be interpreted in terms of API4KP operations. In most implementation, 'training' and 'production' phases are distinct, but hybrid paradigms where a model is trained on the fly ('continuous learning'), or where different models are run and trained in parallel, should also be supported.

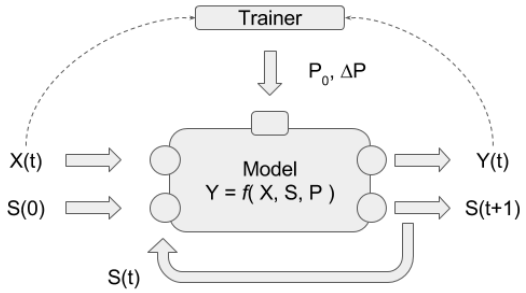

Figure 27: Training / Incremental KB Construction

(Re)Deployment
A Knowledge Base is initialized, and the Resource that expresses the Model - in terms of Parameters P and a model structure (e.g., neural network, SVM, etc…) P that determines the evaluation rules - is added to that Knowledge Base. Once a KB with a given model exists, new versions of that model may be generated. The existing KB may be updated (i.e., a new version of the KB is created) with the new version of the model; or a brand-new KB may be initialized, leaving the two models to run, possibly in parallel.

Initialization / Reset
For recurrent models, the internal state is asserted, or replaced by a full update.

Evaluation
Given a KB that contains a model and a current state S, the current set of input features X is asserted or updated (if previously asserted), partially or completely. An inference is then run on a snapshot of the KB, inferring Y and the new value of S. The output Y is returned or queried asynchronously.

Training
Given a current snapshot of the model's KB, the model is evaluated on a set of input features X. The resulting output Y is consumed by a trainer, to determine the new, adjusted values of the model's parameters.
In *Supervised Training* models, a reference output Y* is used for this computation: $\Delta P = g(Y-Y^*)$; in *Unsupervised* models, different approaches such reinforcement signals are used. In *Incremental Training* models, the new value of the model's Parameters is determined based on the current value; in *Batch Training* the new parameters are computed and then used to replace the current values.

In *Offline* training, training and evaluation are run separately, in *Online* training, the two activities run in parallel. Online training is particularly useful when the environment is subject to changes: incremental online training is used to handle *Concept Drifts*, while full redeployments of a (new version of a) model is used to handle *Concept Generation*. Of particular interest to API4KP are Knowledge-Based trainers, such that the training logic is also specified as a Knowledge Base, and decisions on how/when to update a model are driven by Knowledge Resources.

C.7 'SME to Screen' hybrid pipelines

Healthcare is a domain where the collective corpus of knowledge is vast and evolves quickly. In a "Learning Health System", Knowledge is be delivered at the point of care, monitored, and used to acquire new Knowledge in a continuous feedback loop. Situation-aware, Cognitive Support Applications have been proven to effectively support clinicians in their workflow.
One of the (many) challenges in the development of clinical applications is the elicitation / representation / implementation process, which can be simplified adopting knowledge-driven solutions.
Business oriented, visual knowledge modeling languages such as CMMN and DMN can help mediate between subject matter experts and knowledge engineers. The availability of mature tooling further improves the elicitation-representation cycles, even if business models must be augmented with domain semantics that comes from well-established clinical and medical ontologies. However, business models are more suitable for backend automation than client-side cognitive support – not many engines are designed *to prevent* complete automation, and the XML-based concrete syntax is less than optimal for use in modern, client-side web applications. On the other hand, HL7 has designed a suite of standards, FHIR, which is designed for that exact purpose, and has much better integration with clinical data. However, FHIR models have ambiguous semantics and, to this date, are severely lacking in terms of tooling. A hybrid approach could leverage the best of both standard platforms: elicitation and reasoning using OMG standards, integration and delivery using FHIR standards.

API4KP can be used to mediate both the authoring/publication and the delivery stages. Authoring environments can implement the same APIs as a Knowledge Repository to support ETL processes which involve the 'weaving' of ontology fragments into the models, as well as the generation of metadata.

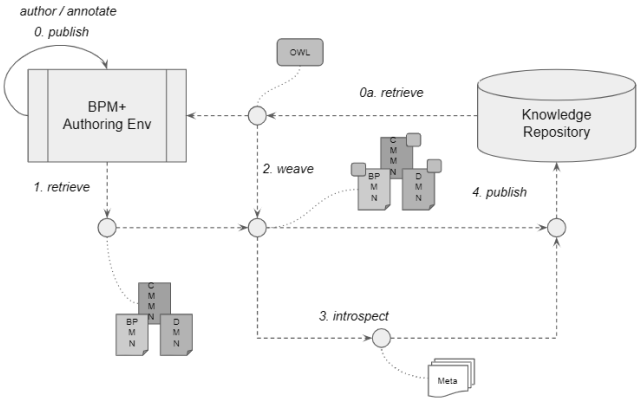

Figure 28: Knowledge Representation + Publication

On the delivery side, clinical Care (Process) Models are Composite Assets/Artifacts that must be assembled, translated, and flattened on demand. Throughout the process, identifiers, versions, and other metadata information must be propagated consistently. In particular, provenance and traceability plays an important role due to the regulatory constraints mandated on clinical applications.

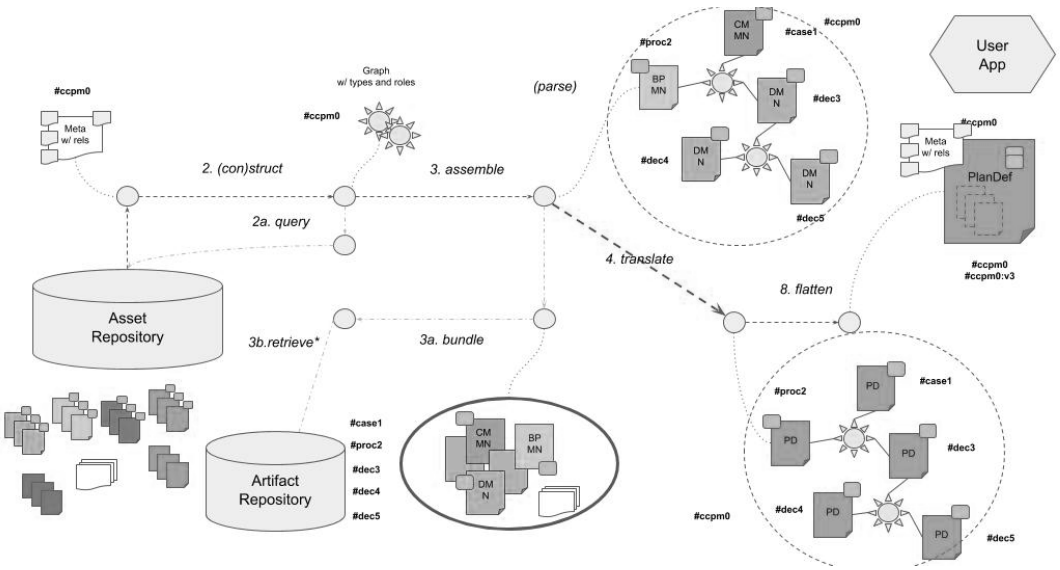

Figure 29: Chaining Operations for Delivery

This page intentionally left blank.

Annex D: Architectural Styles (informative)

D.1 Integration Styles

Interface-mediated object-oriented libraries, RMI/RPC and Web Services are just a few of the possible architectural paradigms that can be used to connect an (API4KP) client and server. The following list describes a number of additional options, differentiated in terms of the degree of coupling between the client and the server. The alternatives can be classified along a number of 'dimensions' which defined the main qualities of a given style.

- **Direct Strongly-Coupled API4KB Access**

Strong coupling with the local client requiring direct knowledge of the (downloaded) API4KB Artifacts or direct knowledge about how the inter-process interaction and access with the remote API4KB works in ad-hoc network programming (e.g., via socket programming).
Example: OntoMaven and RuleMaven.

- **Loosely-coupled Remote Invocation via API4KB Interfaces**

Wide range of techniques based on a loosely-coupled two-way exchange via an interface between communicating entities.

- **Request-Reply Protocols**

Protocols involve pairwise exchange of messages from client to server and from server back to client with the first message containing an encoding of operation to be executed at the server, the second message contains the result (encoded as an array of bytes). Paradigm is rather primitive (in contrast to RPC/RMI) and typically only used for e.g., embedded systems where performance is very important. Approach is also used by e.g., the http protocol.

- **Remote Procedure Calls**

Examples, e.g., Web Services, stateless REST Web Services, Enterprise Service Bus, …

- **Remote Method Invocation**

Resembles RPC in the world of distributed objects. Distributed Object Middleware, e.g., Java RMI (which is restricted to Java). OMG CORBA is a multi-language solution with a declarative Interface Description Language (IDL). Usually, developers choose to invoke CORBA methods through a static interface, which is obtained by using an automated tool to translate the IDL into the chosen implementation language. However, it is also possible to formulate a CORBA message using the facilities of the Dynamic Invocation Interface (DII).

- **Distributed Components**

A unit of composition with contractually specified interfaces and explicit content dependencies only. Component is specified in terms of a contract which includes a set of provided interfaces (interfaces that the component offers as a service to other components) and required interfaces (dependencies that this component has on other components). A container provides managed server-side hosting environment for components and deals with the distributed systems and middleware issues. Examples, e.g., Java Beans, Corba Component Model, OntoMaven Aspect-Oriented Component Model, …

- **Decoupled Indirect Communication**

Indirect decoupled techniques where sender and receiver are time and space uncoupled via an intermediary. Indirect communication between entities in a distributed system through an intermediary with no direct coupling between the sender and the receiver(s), e.g., event routing in publish-subscribe middleware (based e.g., on peer-to-peer), streaming to a cloud, ….

- **Publish Subscribe and Distributed Event Based Systems**

Publish-subscribe with event-based communication through propagation of events (via an underlying overlay network, e.g., structured, and unstructured peer-to-peer or other broker overlay). Publishers publish structured events to an event service (responsible for event routing and matching) and subscribers express interest in particular events through subscriptions. Distributed Event Based Systems and Event Streaming with Complex Event Processing (CEP) in Event Processing Agents (EPAs) deployed in Event Processing Networks (EPNs).

- **Group communication**

Broadcast and multicast.

- **Shared Resources**

Examples, e.g., Tuple Spaces, Distributed Shared Memory, Message Queues (e.g., JMS, Active MQ, …)

- **Asynchronous Messaging Libraries**

Libraries exist which combine and can be used according to a number of the different, above mentioned messaging patterns, e.g., 0MQ (ZeroMQ) is a lightweight messaging system specially designed for high throughput and low latency scenarios as in IoT, low-level event streaming etc. It provides "sockets" (a many-to-many connection generalizing the concept of network socket) which each operate according to a specific messaging pattern (e.g., RPC, pub-sub, …), which (in contrast to e.g., more advanced messaging queue servers and enterprise service bus middlewares) need to be manually implement by combining various pieces of the framework (see ad-hoc network programming with sockets and devices in strong coupling category).
higher-level message queue middleware such as Erlang (RabbitMQ), C (beanstalkd), Ruby (Starling or Sparrow), Scala (Kestrel, Kafka) or Java (ActiveMQ), and Enterprise Service Bus middleware such as OpenESB, Mule ESB, …

Summary of Dimensions

- **Time Uncoupling** - The sender and the receiver(s) can have independent lifetimes
- **Space Uncoupling** - The sender does not know or need to know the identity of the receiver(s), and vice versa
- **Synchronous** vs. **Asynchronous Communication** - sender sends a message and then continues without blocking
- **Centralized** vs. **Distributed b Hybrid** Architecture
- **Structured** vs. **Unstructured Topology**
- **Placement**: Multiple Servers, Proxy/Cache, Mobile Code
- **Architecture Styles**: Client-Server, Broker Overlay, Peer-to-Peer
- **Multiplicity of Message Recipients:** Group Communication, Brokered/Brokerless
- **Directionality** of Communication (Uni-/Bi-directional)
- **Communication Entities**: Processes, Objects, Components, Services / Agents
- Classification of **Endpoints** (Single-/Multi-Sorted)

D.2 Knowledge Based System Patterns

A generic Knowledge Based System (KBS) can be defined in terms of a few key components:

- **D - 'Data' or 'Facts'**
  true statements which are expected to be processed using the Knowledge.
- **K - 'Knowledge' - including 'Ontologies', 'Models' and/or 'Specifications'**
  insights that allow to extract Information out of Data by means of 'Reasoning'.
- **E - 'Engine' aka 'Reasoner'**
  virtual machine (interpreter or compiler) that allows to apply the Knowledge to the Data.
- **C - 'Context' - including 'Parameters'**
  additional information controlling the inference, and/or describing the client's environment. May include identifying information, ways to resolve references, and/or auth*ion parameters, as well as temporal/versioning information, and/or control parameters.
- **A - 'Answer'**
  the (part of the) computation result which satisfies the needs of a client. Ranges from anything as simple as a yes/no boolean to a full theory that is the result of an inference.
  May or may not include additional information (see the API4KP notion of 'Explanation')

The existence of a 'client' needing the Answer, which justifies the effort in the first place, and the availability of a computing environment that allows to run the Engine, deploy the knowledge, and feed the data is instead taken for granted.

API4KPs are generic enough to enable the construction of any Knowledge Based System from its very foundation. However, not every KBS needs to be constructed dynamically at runtime. The purpose of this Annex is to describe more traditional KBS implementation patterns and show how they can be mapped to an API4KP specification, whether conceptually or for actual implementation reasons, possibly to create a facade on an API4KP compliant architecture.

In general, a KBS system can be modelled in functional terms:

Answer ← Data, Knowledge, Engine, Context

Different KBS implementation patterns[6] fix some of the variables in advance, as opposed to allowing the client to specify the values in each individual call. In theory, up to 32 (2^5) combinations are possible, but some are more interesting and/or common than others.

Query vs Test

Answer A, as the result of a computation, is always returned to the client by the server, regardless of the arrangement of the other parameters. This can be considered a 'query' to the
`A ← f*(*)`
A few systems, instead, allow the client to submit a candidate 'test' answer, and respond whether that answer is correct or not:
`bool ← f*( A, * )`
In practice, these systems are quite rare: the 'Answer' is generally an output.

Input Context

For similar pragmatic reasons, the 'Context' is almost always an input - or handled at a different level of abstraction (e.g. authorization). In distributed architectures, the 'Context' is often materialized at the *envelope* level, whereas inputs and outputs are considered *payload*.

When discussing common scenarios, then, it is reasonable to start from
`A ← f*( C, * )`
and discuss different combinations of the use of D, K and E.

Inversion of Control

Whenever any of the constituents (D, K and/or E) is under the responsibility of the Server, two options are possible:

- *static configuration*: the entity that created the Server is responsible for initializing the component with predefined implementations.
- *dynamic deployment*: the Server, at runtime, will be able to acquire and deploy the Data, Knowledge and/or Engine - e.g. respectively from a Data Source, a Knowledge Repository and/or a Software Container.
  The choice of the specific element may be predetermined, or *hinted* to by the Client, as part of the Context C.
  The emphasis on 'hints' is important because, even when Clients pass references to constituents (e.g. a the ID of an entity for which Data has to be retrieved), the Server implementation may still have some degrees of freedom on how to resolve the references.

Main scenarios:

In theory, non-trivial cases reduce to 8 (2^3). However, not all are meaningful.

Platform aaS (PaaS)

`A ← f( D, K, E | C )`

In this scenario, a provider offers a computational platform. A client is allowed to deploy their own reasoning Engines (e.g., installing a Virtual Machine image, or deploying a container), set up their Knowledge Base and feed Data to compute Answers.

[6] It is important to qualify these as 'patterns', because each individual implementation will scope the kind of knowledge, data, engines, information, and context that are supported (and meaningful)

Reasoner aaS
$A \leftarrow f_{E}(D, K \mid C)$
In many cases, the platform already provides components, including Engines. A client is allowed to create 'Session' where an Engine is instantiated, and a Knowledge Base can be set up. The Data is then passed to the Reasoner for computation.

Knowledge aaS
$A \leftarrow f_{K, E}(D \mid C)$
One of the most common patterns, a predefined Knowledge Base running on a given platform is exposed as a Service. Clients can submit their Data and receive Answers in return. The Client may or may not be able to retrieve the Knowledge used to process the data - or a distilled version thereof, as part of the provenance of the Answer.

Data(set) aaS
$A \leftarrow f_{D, E}(K \mid C)$
In some cases, the Server holds the data and the computational power, while allowing Clients to submit a variety of 'jobs' (e.g. 'queries') to be applied to the data. This pattern is common when data sets are too large to be transmitted over the network (e.g. a patient's genetic information), or when the data is subject to legal and/or security restrictions.

Microservice
$A \leftarrow f_{D, K, E}(\mid C)$
In this conceptualization, then, a (Knowledge-Based) Microservice is a service that is built on a specific Platform, leverages existing Engines with pre-defined Knowledge Bases, and is able to possess/acquire the Data needed by a client. The service solves a specific problem, and the client is simply required to pass minimal context (e.g., auth*ion) in order to retrieve the Answer.

(Other)

$A \leftarrow f_{*}(E, * \mid C)$
The remaining scenarios, where the Client is actually able to pass the Engine as a parameter, are quite uncommon, possibly with the exception of some scenarios based on code (agent) mobility.

Not-a-service
For completeness, we mention but do not discuss the 'void' scenario, which is not a service at all:
$\emptyset \leftarrow f_{A,D,C,K,E}(\emptyset)$

Layering:

The various patterns are partially ordered in terms of generality, with PaaS being the most general, and Microservices being the most specific. Any time two (candidate) interfaces can be related by generality/specificity, it is possible to define mappings, which in turn can be implemented as facades/adapters. Two cases are possible:

General → Specific
$f(X, Y \mid Z) \Rightarrow f_{X}(Y \mid Z)$

In this case, an API allows a client to specify X, but implementations exist for specific (classes of) admissible values for X. A *router* is needed to map the general call to the appropriate specific implementation, i.e.

$f(x, y, z) \leftarrow$ case (x) {
$a : f_{X=a}(y, z)$
…
}

Specific → General
$f_{X}(Y \mid Z) \Rightarrow f(X, Y \mid Z)$

In this scenario, it is a responsibility of the adapter to acquire the values for the ‘missing’ parameter, i.e.:

```
f(y,z) ← f( getX( z ), y, z )
```

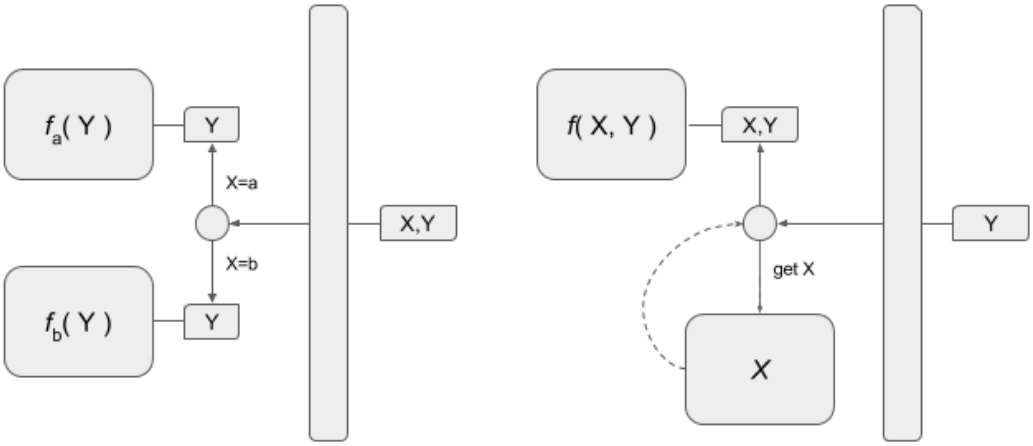

Figure 30: Reasoner Integration Patterns

This page intentionally left blank.

Annex E: Examples

The complete source code for the examples is available at https://github.com/API4KBs/api4kp-examples

1) Initialization

A Monad-style API is used to bind an Artifact – in Binary, String, Parse or AST format, to the API framework using the KnowledgeCarrier wrapper.

```
KnowledgeCarrier kc = of(
        "<html xmlns=\"http://www.w3.org/1999/xhtml\"> "
            + "<head> "
            + "  <title>My Title</title> "
            + "</head> "
            + "<body> "
            + "  My Content "
            + "</body> "
            + "</html> ")
        .withRepresentation(XHTML, XML, UTF_8)
        .withAssetId("urn:myAsset:000")
        .withArtifactId("urn:artifact123:0.0.1");
```

This operation requires not only to specify the Artifact, but also allows to provide identity and syntactic information. The client is able to pass argument values such as “XHTML” or “XML” that are controlled terms derived from an API4KP ontology.

2) Lift / Lower

```
KnowledgeCarrier<OWLOntology> ontology =
       parser.applyLift(
              of( /* fetch from URL */ ).withRepresentation(
                     OWL_2, RDF_XML_Syntax, XML, defaultCharset())),
      Abstract_Knowledge_Expression)

KnowledgeCarrier<String> serialized =
       ontology.flatMap(o ->
              parser.applyLower(
                     o, Serialized_Knowledge_Expression,
                            rep(OWL_2, TTL_Syntax, TXT, defaultCharset()))
```

The manipulation of Knowledge Artifacts often involves the parsing and/or serialization of the Expressions. The ‘lift’ and ‘lower’ APIs are agnostic of the specific syntactic representation, which are provided as parameters.
This approach not only provides a uniform interface but allows for broker-oriented implementations where multiple components can be exposed through the same endpoint.

This specific example delegates the parsing and serialization of the artifact, which is declared to be an OWL ontology, to the OWL APIs. Alternative implementations based on Semantic Web frameworks such as Jena are possible and would not impact the client.

3) Introspection

```
KnowledgeCarrier artifact = /* read binary */
KnowledgeCarrier manifest = /* read … */

KnowledgeCarrier<KnowledgeAsset> =
         introspector.initKnowledgeBase(artifact)
                  .flatMap(p -> x.populateKnowledgeBase(p.getUuid(),p.getVersionTag(), manifest))
                  .flatMap(kb -> x.introspect(kb.getUuid(), kb.getVersionTag()));
```

Metadata Surrogates are used to provide more comprehensive information about Assets and Artifacts. Surrogates are used for publication in a Knowledge Asset Repository and drive the composition of Assets.
Surrogates can be asserted, but can also be inferred, typically from the target Artifact itself and one or more context objects, which are also considered Artifacts.

Since more than one object is involved, it is necessary to leverage an incremental Knowledge Base. The KB is initialized with the root object, then additional components are added. Finally, an 'introspection' operator is involved: that operator is expected to be able to parse and interpret both Artifacts in the KB and use the information to instantiate a canonical metadata Surrogate, which could be returned or further processed, e.g., to satisfy a client's content negotiation preferences.

4) Publication / Surrogate + Carrier

```
KnowledgeCarrier<KnowledgeAsset> surrogate = /* … */
KnowledgeCarrier<OWLOntology> artifact = /* … */

KnowledgeCarrier<byte[]> binary = parser.applyLower(artifact, … );

surrogate.flatmap( metadata ->
        assetRepo.setKnowledgeAssetVersion(
                meta.assetId.uuid, meta.assetId,versionTag, meta);
binary.flatmap( bytes ->
        surrogate.flatMap( meta ->
                assetRepo. setKnowledgeAssetCarrierVersion(
                        meta.assetId.uuid, meta.assetId,versionTag,
                        binary.assetId.uuid, binary.assetId.versionTag,
                        bytes);
```

Once Artifacts and their metadata have been acquired, they can be published to an Asset Repository from where they can be discovered and retrieved by other clients. The Asset Repository API looks at artifacts from the Asset perspective, as Surrogates and Carriers.

An Asset Repository implementation should consider the metadata stored therein as a Knowledge Graph, and could be implemented, for example, on a Triple Store or Graph database. The Asset Repository then embeds a Knowledge Artifact Repository client, and delegates to this component the persistence of the actual Artifacts. Implementations of a Knowledge Artifact Repository can be provided by most Database Management Systems, and/or Content Management Systems.

5) Inference

```
Answer<KnowledgeAsset> asset = assetRepo.listKnowledgeAssets(Decision_Model).map(l -> l[0])
          .flatMap(ptr -> assetRepo.getKnowledgeAsset(ptr.uuid);

Bindings input = new Bindings<>();
map.put("x", true);
map.put("y", 42);
Answer<Bindings> output = asset.flatMap( model ->
          reasoner.evaluate( model.uuid, model.versionTag, input ));
```

The basic Inference operation requires the client to bind the input variables. The client submits the bindings to a specific Knowledge Base runtime, using the ID of the (root) Knowledge Asset that has been used to initialize the KB. In the example, the Asset is a Decision Model that – in a way that is irrelevant to the client – has been expressed in DMN and deployed in an open source, DMN compliant rule engine.

The Server itself is implemented using API4KP. At some point before serving the client's request, the server will also consult the Asset Repository to acquire metadata about the model, and/or the model itself. Unlike the client, the server may not have to discover the models, but rather have a known list, or simply react to the client's request.

```
Answer<KnowledgeAsset> asset = assetRepo.getKnowledgeAsset(model.uuid, model.versionTag);
```

```
Answer<KnowledgeCarrier<DMNDefinitions>> dmn =
         assetRepo.getKnowledgeAssetCarrier(model.uuid, model.versionTag)
                  .flatMap(binary -> parser.applyLift(binary, DMN))

Function<Bindings, Answer<Bindings>> evaluate = input ->
         kbManager.initKnowledgeBase(dmn)
                  .flatMap(kb -> DMNEngineFactory.newEngine(kb.getManifestation()))
                  .flatMap(engine -> engine.nativeEval(input))
                  .map(output -> new Bindings(output));
```

The Server uses implementation-specific methods to instantiate the engine using the acquired model and adapts back the results into an API4KP wrapped Bindings object, to be returned to the client.

An operation similar to *evaluate* is *infer*. Instead of exchanging bindings of input / output variables, infer returns the entire inferred Knowledge Base obtained from the client provided asserted Knowledge Base. Like in the previous example, the API bridges the standard interfaces with the native implementation of the reasoner – an integration that is further mediated by the OWL-API.

```
Function<KnowledgeCarrier, Answer<KnowledgeCarrier>> infer = asserted ->
         kbManager.initKnowledgeBase(asserted)
                  .flatMap(kb -> OWLReasoner.newEngine(kb.getManifestation()))
                  .flatMap(engine -> engine.nativeInfer(input))
                  .map(output -> parser.applyLower(inferred, …);
```

6) Composite Transformation Pipeline

Despite the conceptual complexity, Use Case C7 can be abstracted and simplified significantly using properly implemented operators. As a precondition, all the necessary BPM+ models have been published in the same Asset Repository using the APIs as shown in Example 4.

```
KnowledgeCarrier planDefinitionComposite =
        constructor.getKnowledgeBaseStructure(getRootAssetID(), getRootAssetVersion())
            .flatMap(kc -> assembler.assembleCompositeArtifact(kc))
            .flatMap(kc -> parser.applyLift(kc, Abstract_Knowledge_Expression))
            .flatMap(ckc ->
                translator.applyTransrepresent(
                    ckc,
                    rep(FHIR_STU3, SNOMED_CT, PCV)))
            .orElseGet(Assertions::fail);

    KnowledgeCarrier flatPlanDef = flattener
        .flattenArtifact(planDefinitionComposite, getRootAssetID())
        .orElseGet(Assertions::fail);
```

First, a Constructor implementation connects to the Asset Repository, and leverages the client's provided asset Id and the Asset-to-Asset 'Import' relationships to query the Asset Repository's Knowledge Graph and generate the Structure of the composite Care Process Model. This structure is provided to an Assembler, which is responsible for retrieving the Artifacts – the various CMMN, BPMN and DMN models – and combining them into a single CompositeKnowledgeCarrier.
The Artifacts are lifted to the object level, and passed to a trans-representation operator which, for each BPM+ model, generated a corresponding, derivative, FHIR PlanDefinition. The FHIR resources are reduced back to a single CompositeKnowledgeCarrier. Finally, a Flattener is responsible for taking the homogeneous CompositeKnowledgeCarrier, extracting the PlanDefinition resources, and generating a single PlanDefinition.

7) Terminology API

Terminology servers allow, among others, to resolve concept identifiers to descriptions of the identified concepts. It is common for the identifier to be a URI, and the description to be an RDF graph that is the result of a SPARQL query.

Along the same lines, a terminology server could allow a client to resolve the identifier of a vocabulary and return all the (descriptions of the concepts evoked by the) terms contained in the vocabulary itself.
API4KP could be used to implement the operations of a classic terminology server using an architecture that involves a SPARQL engine, a SPARQL endpoint backed by a triple store, and a (SKOS) Graph/Ontology.

```
listTermsForVocabulary(KnowledgeAsset vocabularySurrogate, String labelFilter) {
        return termsKBManager.initKnowledgeBase(wrap(vocabularySurrogate))
                .flatMap(kBaseId ->
                        getQuery(vocabularySurrogate, labelFilter)
                                .flatMap(boundQuery -> askQuery(kBaseId, boundQuery)));
}
```

In response to a client's request involving the asset ID of a vocabulary, the server will use a Knowledge Asset Repository to fetch the Vocabulary's Surrogate and use it to initialize a KnowledgeBase. In particular, the Surrogate's KnowledgeArtifact component contains the URL where the Vocabulary is accessible via a SPARQL endpoint.
The server will then use that information, along with other context metadata, to (i) discover, (ii) get, (iii) lift, (iv) bind a SPARQL query that realizes the client's request, and finally submits the query to the SPARQL endpoint.
More specifically, the server invokes an internal component that implements the API4KP *askQuery* operation, which in turn is implemented as a proxy for the remote SPARQL endpoint.

The remote SPARQL server may or may not be API4KP compliant itself: the Knowledge Graph could have been acquired as a Knowledge Artifact and wrapped in a KnowledgeBase. Likewise, instead of exposing a native SPARQL endpoint, the server could have exposed the API4KP *askQuery* operation itself and mapped that internally to a SPARQL engine.

8) GraphQL (with SPARQL)

From an API4KP perspective, GraphQL architectures consist in the combination of a Schema Language, a Query Language, a functional Query engine and a generic API.
As such, GraphQL endpoints can be integrated using the *askQuery* operation, submitting queries in the GraphQL query language, not unlike demonstrated with SPARQL in example 7.

```
Answer<KnowledgeCarrier> gqlQuery = Answer.of(GQL_QUERY)
        .map(s -> of(s).withRepresentation(rep(GraphQL_Queries, TXT, defaultCharset())));

List<Bindings> response =
        gqlQuery.flatMap( query ->
                gqlKBPointer.flatMap( ptr ->
                        gqlQueryEngine.askQuery( ptr.getUuid(), ptr.getVersionTag(), query )))
```

Some GraphQL engine implementations leverage the notion of 'resolver function' to associate back-end queries to the fields in the GraphQL schema that are subject to a Query.
API4KP could enable an architecture where the Resolvers are (i) knowledge-based and (ii) bound dynamically to the engine.

To this end, one could:

- Model each resolver as a formal query in a query language, e.g., SPARQL
- Assert a Dependency relationship between the query and the GraphQL schema
- As the GraphQL engine is instantiated, resolve the Dependency, and get the Query, lift it and inspect it to obtain the name of the query's output variable
- Lift and inspect the GraphQL schema to validate that the variable name maps to an actual field in the schema
- Bind the query to the field's resolver function. In particular, the function will bind the query, and use it in combination with the *askQuery* operation. Notice that, while invoked and directed at different contexts, the main GraphQL and the secondary SPARQL query are invoked in the exact same way.
- The SPARQL query's result bindings are bound back into the GraphQL schema instance and returned to the initial client.
- Any/all these interactions could be mediated by a Knowledge Asset Repository.

```
Answer<KnowledgeCarrier> sparqlQuery = Answer.of(SQL_QUERY)
        .map(s -> of(s).withRepresentation(rep(SPARQL, TXT, defaultCharset())));

List<Bindings> response =
        sparqlQuery.flatMap( query ->
                gqlKBPointer.flatMap( ptr ->
                        gqlQueryEngine.askQuery( ptr.getUuid(), ptr.getVersionTag(), query ))
```

This page intentionally left blank.

Annex F: Informative API4KP Machine-Readable Files

A number of derivative files are included by reference in this specification: a set of .yaml vocabulary files, generated from the UML datatype models that are a normative component of the specification, a set of XML schema datatype files (.xsd), also generated from the UML datatype models that are a normative component of the specification, and a set of SKOS vocabularies (.rdf), that are derived (MIREOT-ed) from the normative ontologies and are further flattened for use with the APIs as discussed in section 7.2.1.

The list of files and their URLs are provided herein for reference purposes.

F.1 API4KP OpenAPI .yaml Vocabulary Files

The files listed below comprise the informative set of .yaml vocabularies, generated from the UML datatype models that are a normative part of this specification.

Table E-1. Filenames and URLs for the informative OpenAPI .yaml Vocabulary Files comprising Application Programming Interfaces for Knowledge Platforms (API4KP)

Filename	URL
api4kp.yaml	https://www.omg.org/spec/API4KP/20210201/api4kp.yaml
datatypes.yaml	https://www.omg.org/spec/API4KP/20210201/datatypes/datatypes.yaml
id.yaml	https://www.omg.org/spec/API4KP/20210201/id/id.yaml
inference.yaml	https://www.omg.org/spec/API4KP/20210201/services/inference/inference.yaml
repository.yaml	https://www.omg.org/spec/API4KP/20210201/services/repository/repository.yaml
transrepresentation.yaml	https://www.omg.org/spec/API4KP/20210201/services/transrepresentation/transrepresentation.yaml
services.yaml	https://www.omg.org/spec/API4KP/20210201/services/services.yaml
surrogate.yaml	https://www.omg.org/spec/API4KP/20210201/surrogate/surrogate.yaml
DependencyType.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/DependencyRelType/DependencyType.series.yaml
DependencyType.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/DependencyRelType/DependencyType.yaml
DerivationType.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/DerivationRelType/DerivationType.series.yaml
DerivationType.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/DerivationRelType/DerivationType.yaml
Language.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/ISO639-2-LanguageCode/Language.series.yaml
Language.yaml	https://www.omg.org/spec/API4KP/20190201/taxonomy/ISO639-2-LanguageCode/Language.yaml

KnowledgeArtifactCategory.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeArtifactCategory/KnowledgeArtifactCategory.series.yaml
KnowledgeArtifactCategory.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeArtifactCategory/KnowledgeArtifactCategory.yaml
KnowledgeAssetCategory.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeAssetCategory/KnowledgeAssetCategory.series.yaml
KnowledgeAssetCategory.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeAssetCategory/KnowledgeAssetCategory.yaml
KnowledgeAssetType.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeAssetType/KnowledgeAssetType.series.yaml
KnowledgeAssetType.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeAssetType/KnowledgeAssetType.yaml
KnowledgeProcessingOperation.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeProcessingOperations/KnowledgeProcessingOperation.series.yaml
KnowledgeProcessingOperation.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeProcessingOperations/KnowledgeProcessingOperation.yaml
KnowledgeProcessingTechnique.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeProcessingTechnique/KnowledgeProcessingTechnique.series.yaml
KnowledgeProcessingTechnique.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeProcessingTechnique/KnowledgeProcessingTechnique.yaml
KnowledgeRepresentationLanguage.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguage/KnowledgeRepresentationLanguage.series.yaml
KnowledgeRepresentationLanguage.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguage/KnowledgeRepresentationLanguage.yaml
KnowledgeRepresentationLanguageProfile.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageProfile/KnowledgeRepresentationLanguageProfile.series.yaml
KnowledgeRepresentationLanguageProfile.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KRProfile/KnowledgeRepresentationLanguageProfile.yaml
KnowledgeRepresentationLanguageRole.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageRole/KnowledgeRepresentationLanguageRole.series.yaml
KnowledgeRepresentationLanguageRole.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageRole/KnowledgeRepresentationLanguageRole.yaml
KnowledgeRepresentationLanguageSerialization.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageSerialization/KnowledgeRepresentationLanguageSerialization.series.yaml
KnowledgeRepresentationLanguageSerialization.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageSerialization/KnowledgeRepresentationLanguageSerialization.yaml
Lexicon.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/Lexicon/Lexicon.series.yaml

Lexicon.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/Lexicon/Lexicon.yaml
ParsingLevel.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/ParsingLevel/ParsingLevel.series.yaml
ParsingLevel.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/ParsingLevel/ParsingLevel.yaml
PublicationStatus.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/PublicationStatus/PublicationStatus.series.yaml
PublicationStatus.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/PublicationStatus/PublicationStatus.yaml
RelatedVersionType.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/RelatedVersionType/RelatedVersionType.series.yaml
RelatedVersionType.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/RelatedVersionType/RelatedVersionType.yaml
SerializationFormat.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/SerializationFormat/SerializationFormat.series.yaml
SerializationFormat.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/SerializationFormat/SerializationFormat.yaml
StructuralPartType.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/StructuralRelType/StructuralPartType.series.yaml
StructuralPartType.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/StructuralRelType/StructuralPartType.yaml
SummarizationType.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/SummaryRelType/SummarizationType.series.yaml
SummarizationType.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/SummaryRelType/SummarizationType.yaml
VariantType.series.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/VariantRelType/VariantType.series.yaml
VariantType.yaml	https://www.omg.org/spec/API4KP/20210201/taxonomy/VariantRelType/VariantType.yaml

F.2 API4KP XML Schemas derived from the UML model files

The files listed below comprise the informative set of .xsd schemas generated from the UML datatype models that are a normative part of this specification.

Table E-2. Filenames and URLs for the informative .xsd XML Schema Vocabulary Files comprising Application Programming Interfaces for Knowledge Platforms (API4KP)

Filename	URL
api4kp.xsd	https://www.omg.org/spec/API4KP/20210201/api4kp.xsd
datatypes.xsd	https://www.omg.org/spec/API4KP/20210201/datatypes/datatypes.xsd
id.xsd	https://www.omg.org/spec/API4KP/20210201/id/id.xsd
inference.xsd	https://www.omg.org/spec/API4KP/20210201/services/inference/inference.xsd
repository.xsd	https://www.omg.org/spec/API4KP/20210201/services/repository/repository.xsd

transrepresentation.xsd	https://www.omg.org/spec/API4KP/20210201/services/transrepresentation/transrepresentation.xsd
services.xsd	https://www.omg.org/spec/API4KP/20210201/services.xsd
surrogate.xsd	https://www.omg.org/spec/API4KP/20210201/surrogate/surrogate.xsd
DependencyType.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/DependencyRelType/DependencyType.series.xsd
DependencyType.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/DependencyRelType/DependencyType.xsd
DerivationType.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/DerivationRelType/DerivationType.series.xsd
DerivationType.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/DerivationRelType/DerivationType.xsd
Language.series.xsd	https://www.omg.org/spec/API4KP/20190201/taxonomy/ISO639-2-LanguageCode/Language.series.xsd
Language.xsd	https://www.omg.org/spec/API4KP/20190201/taxonomy/ISO639-2-LanguageCode/Language.xsd
KnowledgeArtifactCategory.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeArtifactCategory/KnowledgeArtifactCategory.series.xsd
KnowledgeArtifactCategory.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeArtifactCategory/KnowledgeArtifactCategory.xsd
KnowledgeAssetCategory.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeAssetCategory/KnowledgeAssetCategory.series.xsd
KnowledgeAssetCategory.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeAssetCategory/KnowledgeAssetCategory.xsd
KnowledgeAssetType.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeAssetType/KnowledgeAssetType.series.xsd
KnowledgeAssetType.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeAssetType/KnowledgeAssetType.xsd
KnowledgeProcessingOperation.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeProcessingOperations/KnowledgeProcessingOperation.series.xsd
KnowledgeProcessingOperation.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeProcessingOperations/KnowledgeProcessingOperation.xsd
KnowledgeProcessingTechnique.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeProcessingTechnique/KnowledgeProcessingTechnique.series.xsd
KnowledgeProcessingTechnique.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeProcessingTechnique/KnowledgeProcessingTechnique.xsd
KnowledgeRepresentationLanguage.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguage/KnowledgeRepresentationLanguage.series.xsd
KnowledgeRepresentationLanguage.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguage/KnowledgeRepresentationLanguage.xsd

KnowledgeRepresentationLanguageProfile.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageProfile/KnowledgeRepresentationLanguageProfile.series.xsd
KnowledgeRepresentationLanguageProfile.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageProfile/KnowledgeRepresentationLanguageProfile.xsd
KnowledgeRepresentationLanguageRole.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageRole/KnowledgeRepresentationLanguageRole.series.xsd
KnowledgeRepresentationLanguageRole.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageRole/KnowledgeRepresentationLanguageRole.xsd
KnowledgeRepresentationLanguageSerialization.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationLanguageSerialization/KnowledgeRepresentationLanguageSerialization.series.xsd
KnowledgeRepresentationLanguageSerialization.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/KnowledgeRepresentationSerialization/KnowledgeRepresentationLanguageSerialization.xsd
Lexicon.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/Lexicon/Lexicon.series.xsd
Lexicon.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/Lexicon/Lexicon.xsd
ParsingLevel.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/ParsingLevel/ParsingLevel.series.xsd
ParsingLevel.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/ParsingLevel/ParsingLevel.xsd
PublicationStatus.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/PublicationStatus/PublicationStatus.series.xsd
PublicationStatus.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/ParsingLevel/PublicationStatus.xsd
RelatedVersionType.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/RelatedVersionType/RelatedVersionType.series.xsd
RelatedVersionType.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/RelatedVersionType/RelatedVersionType.xsd
SerializationFormat.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/SerializationFormat/SerializationFormat.series.xsd
SerializationFormat.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/SerializationFormat/SerializationFormat.xsd
StructuralPartType.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/StructuralRelType/StructuralPartType.series.xsd
StructuralPartType.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/StructuralRelType/StructuralPartType.xsd
SummarizationType.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/SummaryRelType/SummarizationType.series.xsd
SummarizationType.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/SummaryRelType/SummarizationType.xsd

VariantType.series.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/VariantRelType/VariantType.series.xsd
VariantType.xsd	https://www.omg.org/spec/API4KP/20210201/taxonomy/VariantRelType/VariantType.xsd