An OMG® Systems Modeling Publication

Systems Modeling Application Programming Interface (API) and Services

Version 1.0 Beta 2.2
(Release 2024-09)

OMG Document Number: None

Date: October 2024

Standard document URL: https://www.omg.org/spec/SystemsModelingAPI/1.0/

Machine Readable File(s): https://www.omg.org/spec/SystemsModelingAPI/20240201/

Normative:

https://www.omg.org/spec/SystemsModelingAPI/20240201/Systems-Modeling-API.xmi
https://www.omg.org/spec/SystemsModelingAPI/20240201/OpenAPI.json
https://www.omg.org/spec/SystemsModelingAPI/20240201/Schemas.json
https://www.omg.org/spec/SystemsModelingAPI/20240201/OSLC-Domain-Model.zip

Non-normative:

https://www.omg.org/spec/SystemsModelingAPI/20240201/API-Cookbook.zip

---

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 companys 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

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR

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

List of Tables

List of Figures

0 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 https://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 Specifications, Report an Issue.

1 Scope

The purpose of this standard is to specify the Systems Modeling Application Programming Interface (API) and Services that provide standard services to access, navigate, and operate on KerML-based models [KerML], and, in particular, SysML models [SysML]. The standard services facilitate interoperability both across SysML modeling environments and between SysML modeling environments and other engineering tools and enterprise services.

The Systems Modeling API and Services specifies the types and details of the requests that can be made and responses that can be received by software applications that are consuming the services to software applications that are providing the services.

The Systems Modeling API and Services specification includes the Platform Independent Model (PIM) - see Clause 7 - and two Platform Specific Models (PSMs) - see Clause 8 : REST/HTTP PSM and OSLC PSM.

2 Conformance

This specification defines the Systems Modeling API and Services that provide standard services to access, navigate, and operate on KerML-based models [KerML] and, in particular, SysML models [SysML]. The specification comprises this document together with the content of the machine-readable files listed on the cover page. If there are any conflicts between this document and the machine-readable files, the machine-readable files take precedence.

A **Systems Modeling API and Services Provider** is a software application that provides the services defined in this specification.

A **Systems Modeling API and Services Consumer** is a software application that consumes the services defined in this specification and provided by the Service Provider.

Consumers send requests to Providers and receive responses with results, as illustrated in Fig. 1 below.

For brevity, this specification uses the phrase **Service Provider** for **Systems Modeling API and Services Provider**, and the term **Service Consumer** for **Systems Modeling API and Services Consumer**.

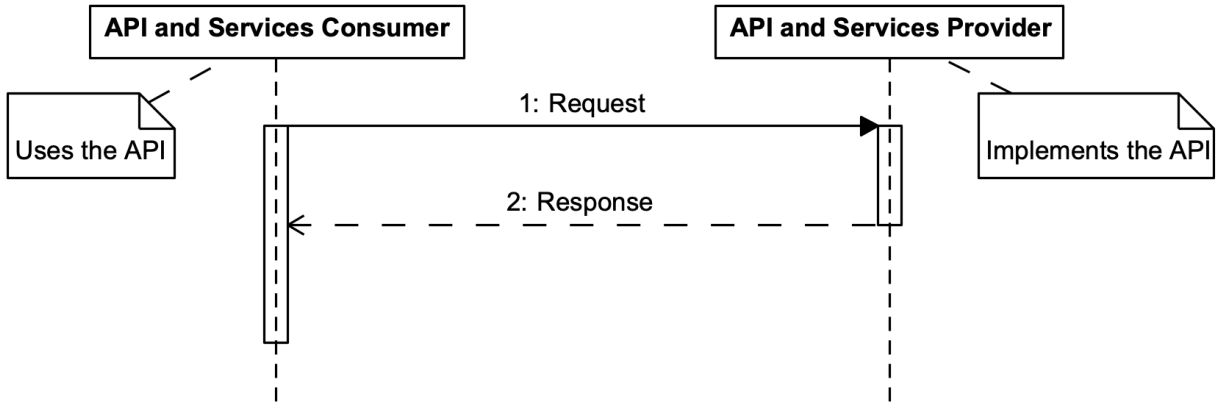

Figure 1. API and Services Provider and Consumer

A Service Provider can conform to this specification at the PSM or PIM level.

1. **PSM-level Conformance** - A Service Provider demonstrating PSM-level Conformance implements one or more of the Systems Modeling API and Services PSMs defined in this specification. For example, a Provider can implement the REST/HTTP PSM, the OSLC PSM, or both. PSM-level conformance of Service Providers ensures interoperability of Service Consumers using the PSM across the different Service Providers. See Clause 8 .
2. **PIM-level Conformance** - A Service Provider demonstrating PIM-level Conformance implements a PSM that is not defined in this specification but is based on Systems Modeling API and Services PIM defined in this specification. The Service Provider shall define the PSM and the mapping from PIM to PSM with the goal that the new PSM may become part of future versions of this specification. See Clause 7 .

A Service Provider tool must demonstrate conformance to one or more services, as described below.

1. **ProjectService Conformance** - A Service Provider must implement all the operations in the ProjectService, and demonstrate that the implementation successfully passes all the Project Service Conformance Test Cases (see A.1) and Cross-Cutting Conformance Test Cases (see A.7).
2. **ElementNavigationService Conformance** - A Service Provider must implement all the operations in the ElementNavigationService, and demonstrate that the implementation successfully passes all the Element

Navigation Service Conformance Test Cases (see A.2) and Cross-Cutting Conformance Test Cases (see A.7).

3. **ProjectDataVersioningService Conformance** - A Service Provider must implement all the operations in the ProjectDataVersioningService, and demonstrate that the implementation successfully passes all the Project and Data Versioning Service Conformance Test Cases (see A.3) and Cross-Cutting Conformance Test Cases (see A.7).
    1. **Derived Property Conformance** - A Service Provider conformant to the ProjectDataVersioningService may optionally demonstrate this additional conformance. The Derived Property Conformance is relevant for project commits where Element (KerML) data is created or updated. In order to demonstrate Derived Property Conformance, a service provider must do the following when ProjectDataVersioningService.createCommit operation is invoked.
        1. Compute or verify all derived properties for Element data that is created or updated in the commit
        2. Include the derived properties in the response, i.e. DataVersion.payload should include derived properties for Element data.
4. **QueryService Conformance** - A Service Provider must implement all the operations in the Query Service, and demonstrate that the implementation successfully passes all the Query Service Conformance Test Cases (see A.4) and Cross-Cutting Conformance Test Cases (see A.7).
5. **ExternalRelationshipService Conformance** - A Service Provider must implement all the operations in the ExternalRelationshipService, and demonstrate that the implementation successfully passes all the External Relationship Service Conformance Test Cases (see A.5) and Cross-Cutting Conformance Test Cases (see A.7).
6. **ProjectUsageService Conformance** - A Service Provider must implement all the operations in the ProjectUsageService, and demonstrate that the implementation successfully passes all the Project Usage Service Conformance Test Cases (see A.6) and Cross-Cutting Conformance Test Cases (see A.7).

3 Normative References

[GraphQL] GraphQL
https://graphql.org/

[Gremlin] Gremlin Graph Traversal Machine and Language
https://tinkerpop.apache.org/gremlin.html

[IRI] *Internationalized Resource Identifiers* (IRI)
https://www.w3.org/International/articles/idn-and-iri/

[KerML] *Kernel Modeling Language (KerML),* Version 1.0
https://www.omg.org/spec/KERML/1.0

[MOFVD] *MOF2 Versioning and Development Lifecycle (MOFVD™),* Version 2.0
https://www.omg.org/spec/MOFVD/2.0

[OpenAPI] *OpenAPI Specification*
https://www.openapis.org/

[OSLC] *Open Services for Lifecycle Collaboration (OSLC)*
http://open-services.net/

[QVT] *MOF Query/View/Transformation (QVT™)*, Version 1.3
https://www.omg.org/spec/QVT/1.3

[SEBoK] *Systems Engineering Body of Knowledge (SEBoK)*
www.sebokwiki.org

[SE Handbook] *INCOSE Systems Engineering Handbook*
https://www.incose.org/products-and-publications/se-handbook

[SMOF] *MOF Support for Semantic Structures (SMOF™)*, Version 1.0
https://www.omg.org/spec/SMOF/1.0

[SPARQL] *SPARQL Query Language for RDF*
https://www.w3.org/TR/rdf-sparql-query/

[SQL] *ISO/IEC 9075:2016, Information technology — Database languages — SQL*
https://www.iso.org/standard/63555.html

[STEP] *ISO 10303-233:2012 (STEP)*
https://www.iso.org/standard/55257.html

[SysML] *OMG Systems Modeling Language (SysML®), Version 2.0*
https://www.omg.org/spec/SYSML/2.0

[UML] *Unified Modeling Language (UML),* Version 2.5.1
https://www.omg.org/spec/UML/2.5.1

[UUID] *Universally Unique IDentifier (UUID) URN Namespace*
https://tools.ietf.org/html/rfc4122

[XMI] *XML Metadata Interchange (XMI®)*, Version 2.5.1
https://www.omg.org/spec/XMI/2.5.1

4 Terms and Definitions

Various terms and definitions are specified throughout the body of this specification.

5 Symbols

There are no special symbols defined in this specification.

6 Introduction

6.1 API and Services Architecture

The Systems Modeling API and Services includes the following.

(1) Platform-Independent Model (PIM) provides a service specification independent of any platform or technology. This specification defines each of the services and their operations with inputs and outputs. The PIM serves as the logical API model.

(2) Platform-Specific Models (PSMs) are bindings of the PIM using a particular technology, such as REST/HTTP, SOAP, Java, and .NET. Multiple platform-specific models can exist for a given PIM. Two PSMs are provided in this specification:

- REST/HTTP PSM - a binding of the PIM as a REST/HTTP API using OpenAPI specification.
- OSLC PSM - a binding of the PIM as services based on the OSLC standard.

For each PSM, a mapping is defined. This mapping is used to generate the PSM from the PIM.

Fig. 2 illustrates the PIM, PSMs, Service Providers that implement API PSMs, and Service Consumers that consume the API PSMs from multiple Providers.

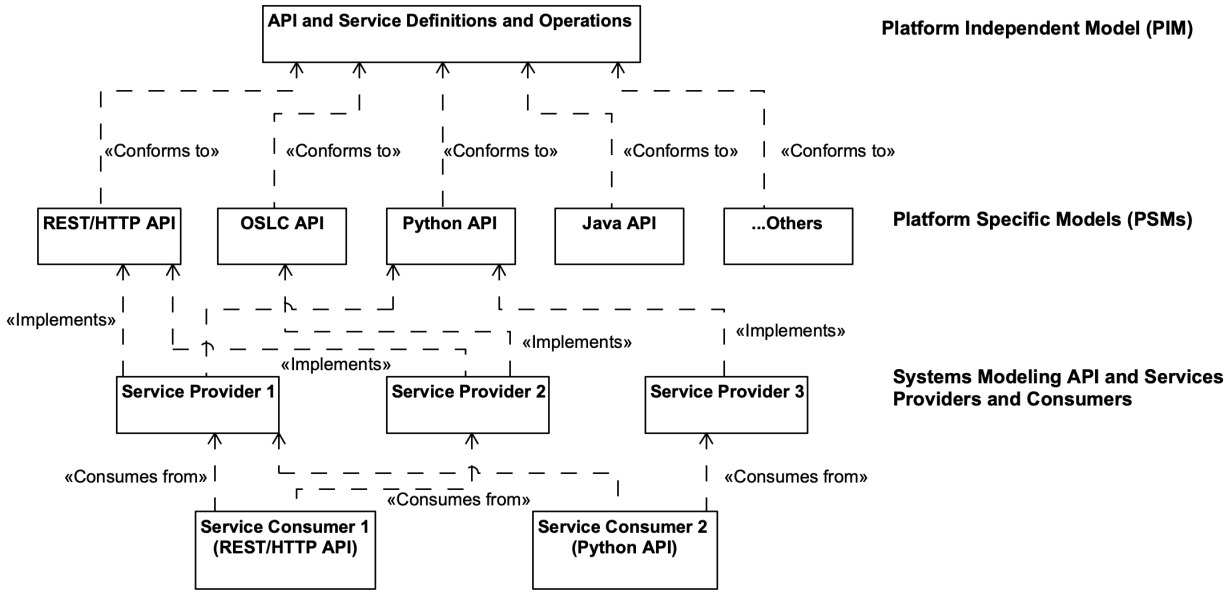

Figure 2. API and Services Architecture

Service specifications in the PIM do not prescribe or constrain the architecture of the Service Providers. For example, Service Providers with file-based, 3-tier application-based, or federated microservices-based architectures can all implement one or more PSMs derived from the same service specifications (PIM).

Service Consumers that use a specific PSM should be interoperable across multiple Service Providers that implement that PSM without requiring any modification in the consumer.

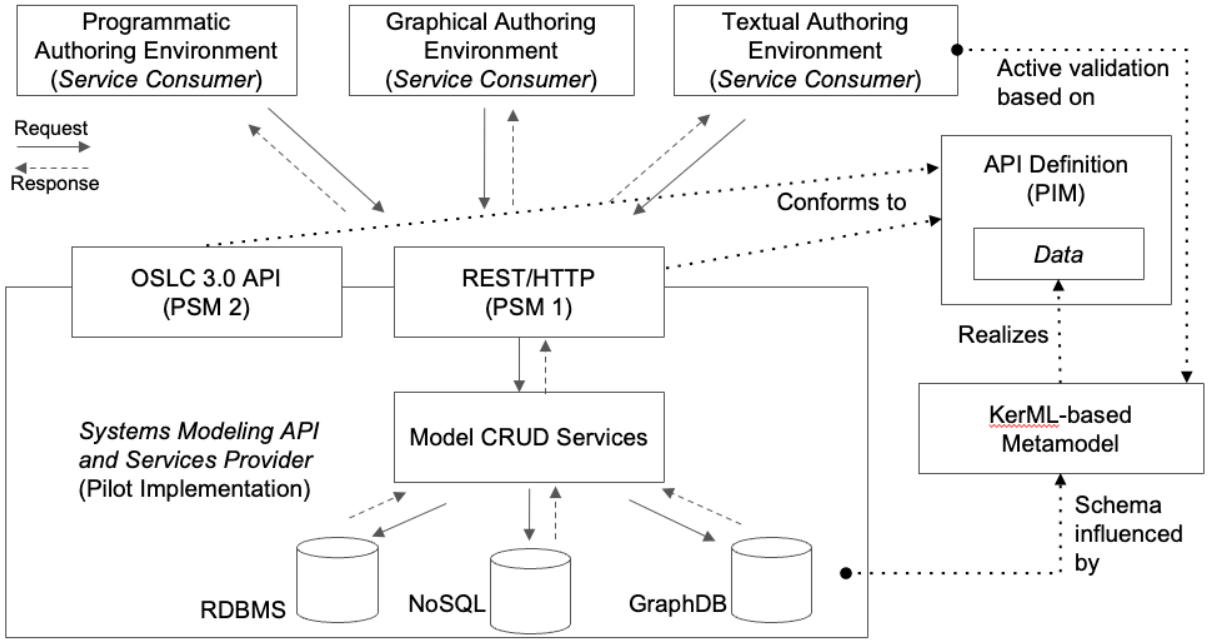

Figure 3. Use of PIM and PSMs by Providers and Consumers

Fig. 3 illustrates the role of PIM and PSMs in the context of Systems Modeling API and Services providers and consumers. The Systems Modeling API and Services, version 1.0, includes two PSMs, specifically the REST/HTTP PSM and OSLC 3.0 PSM.
A System Modeling API and Services provider implements either or both the PSMs using its native technology stack, such as databases and web-service frameworks. Service consumers, such as those used for programmatic, graphical, or textual authoring, navigation, and querying data use the PSMs (e.g. REST/HTTP API), agnostic of the native technology stack of the providers.
The choice of REST/HTTP PSM is key. Most modern programming languages provide libraries for consuming REST/HTTP APIs. Enterprise applications, written in any modern programming language, can consume the standard Systems Modeling API and Services, and interoperate with multiple providers.

6.2 Document Conventions

The following stylistic conventions are applied in the presentation of the Platform Independent Model (PIM) of the Systems Modeling API and Services.

Service definitions

1. Names of classes representing services start with an uppercase letter and use the camel case notation, such as ElementNavigationService.
2. Names of operations representing the API calls available for each service start with a lowercase letter and are italicized, such as *getElementById*

Input and output data

1. Names of classes representing data that is the input or output of services start with an uppercase letter, such as Project and Data
2. Names of attributes representing the details of the data that is the input or output of services start with a lowercase letter and are italicized, such as *identifier*

The services and operations in the PIM are presented using class diagrams and tables with descriptions of each operation.

The input and output data for services in the PIM are presented using class diagrams followed by detailed descriptions.

6.3 Document Organization

The rest of this document is organized into two major clauses.

- Clause 7 - Platform Independent Model (PIM) of the Systems Modeling API and Services
- Clause 8 - Platform Specific Models (PSMs) of the Systems Modeling API and Services
    - 8.1 - REST/HTTP PSM
    - 8.2 - OSLC PSM

These clauses are followed by two annexes.

- Annex A defines the suite of conformance tests that must be used to demonstrate the conformance of Service Providers to this specification - see Clause 2 .
- Annex B includes the following.
    - Examples of requests and responses for the REST/HTTP API endpoints, and
    - Cookbook with a collection of recipes, as Jupyter notebooks, demonstrating patterns and examples for using the Systems Modeling API and Services

6.4 Acknowledgements

The primary authors of this specification document, the PIM, and the REST/HTTP PSM are:

- Manas Bajaj, Intercax LLC
- Ivan Gomes, Twingineer LLC

The primary authors of the OSLC PSM are:

- David Honey, IBM
- Jad El-Khoury, KTH Royal Institute of Technology
- Jim Amsden, IBM

The specification was formally submitted for standardization by the following organizations:

- 88Solutions Corporation
- Dassault Systèmes
- GfSE e.V.
- IBM
- INCOSE
- Intercax LLC
- Lockheed Martin Corporation
- Model Driven Solutions, Inc.
- PTC
- Simula Research Laboratory AS

However, work on the specification was also supported by over 200 people in 80 organizations that participated in the SysML v2 Submission Team (SST). The following individuals had leadership roles in the SST:

- Manas Bajaj, Intercax LLC (API and services development lead)
- Yves Bernard, Airbus (v1 to v2 transformation co-lead)
- Bjorn Cole, Lockheed Martin Corporation (metamodel development co-lead)
- Sanford Friedenthal, SAF Consulting (SST co-lead, requirements V&V lead)
- Charles Galey, Lockheed Martin Corporation (metamodel development co-lead)

- Karen Ryan, Siemens (metamodel development co-lead)
- Ed Seidewitz, Model Driven Solutions (SST co-lead, pilot implementation lead)
- Tim Weilkiens, oose (v1 to v2 transformation co-lead)

The specification was prepared using CATIA No Magic modeling tools and the OpenMBEE system for model publication (http://www.openmbee.org), with the invaluable support of the following individuals:

- Tyler Anderson, No Magic/Dassault Systèmes
- Christopher Delp, Jet Propulsion Laboratory
- Ivan Gomes, Jet Propulsion Laboratory
- Robert Karban, Jet Propulsion Laboratory
- Christopher Klotz, No Magic/Dassault Systèmes
- John Watson, Lightstreet consulting

The following individuals made significant contributions to the API and Services pilot implementation developed by the SST in conjunction with the development of this specification:

- Manas Bajaj, Intercax LLC
- Ivan Gomes, Twingineer LLC
- Brian Miller, Intercax LLC

7 Platform Independent Model (PIM)

7.1 API Model

7.1.1 Record

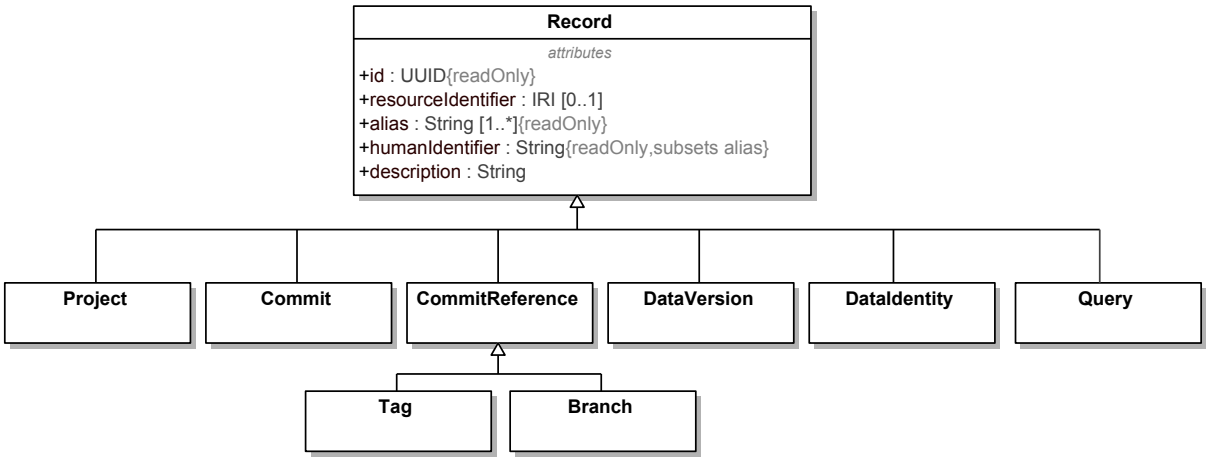

Figure 4. Types of Records

Record - A Record represents any data that is consumed (input) or produced (output) by the Systems Modeling API and Services. A Record is an abstract concept from which other concrete concepts inherit. A Record has the following attributes:

- *id* is the UUID assigned to the record
- *resourceIdentifier* is an IRI for the record
- *alias* is a collection of other identifiers for this record, especially if the record was created or represented in other software applications and systems
- *humanIdentifier* is a human-friendly unique identifier for this record
- *description* is a statement that provides details about the record.

7.1.2 Project Data Versioning

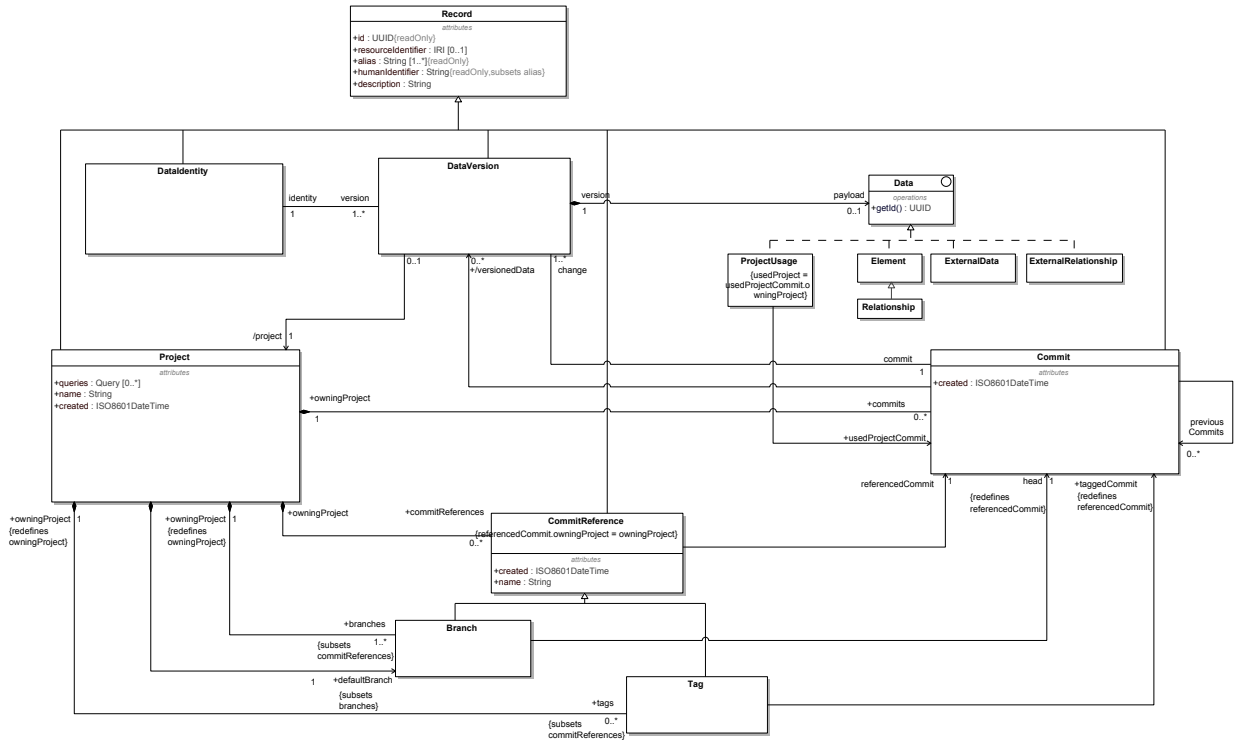

Figure 5. Project Data Versioning API Model

The class diagram above presents concepts related to Project and Data Versioning Service.

Data - Data represents any entity that can be created, updated, deleted, and queried by the Systems Modeling API and Services. In the PIM, Data is represented as an Interface that is realized by the following concepts in the scope of Systems Modeling API and Services.

- Element, root metaclass in the SysML v2 language metamodel
- External Data
- External Relationship
- Project Usage

Each realization of Data must implement the *getId()* operation that provides a valid UUID.

Data Identity - Data Identity is a subclass of Record that represents a unique, version-independent representation of Data through its lifecycle. A Data Identity is associated with 1 or more Data Version records that represent different versions of the same Data. A Data Identity record has the following additional attributes:

- *createdAt* is a derived attribute that references the Commit in a project in which the given Data was created
- *deletedAt* is a derived attribute that references the Commit in a project in which the given Data was deleted

Data Version - Data Version is a subclass of Record that represents Data at a specific version in its lifecycle. A Data Version record is associated with up to one (0..1) Data Identity record. Data Version serves as a wrapper for Data (*payload*) in the context of a Commit in a Project; associating the data identity with the state of the Data (payload) in the specific (range of) Commits, or no payload if no Data element with the given identifier is present at that Commit. Different versions of the same Data, identified by the same UUID values returned by *Data.getId()*, are represented in the following manner:

- One (1) Data Identity record is created for all versions of the same Data, where Data Identity.*id* returns the same UUID value as *Data.getId()*
- A Data Version record is created for each version of Data (and, if needed, also for a Commit where no Data exists for the given identity), where:
    - Data Version.*payload* is set to Data if exists in the commit, null otherwise.
    - Data Version.*identity* is set to the Data Identity common to all versions of the same Data.
    - Data Version.*id* is set to a new, randomly generated UUID for the specific Data Version record.

Data Version record has the following additional attributes:

- *commit*: project commit at which the wrapped data (*payload*) was created, modified, or deleted.
- */project*: derived attribute referencing the owning project

Project - Project is a subclass of Record that represents a container for other Records and an entry point for version management and data navigation. The Project record has the following attributes:

- *identifiedData* is a derived attribute that is the set of Data Identity records corresponding to the Data contained in the project
- *commit* is the set of all commits in the Project
- *commitReference* is the set of all commit references in the Project
- *branch* is the set of all the branches in the Project and a subset of *commitReference*
- *defaultBranch* is the default branch in the Project and a subset of *branch*
- *tag* is the set of all the tags in the Project and a subset of *commitReference*
- *usage* is the set of Project Usage records representing all other Projects being used by the given Project (*Project Usage.usedProject)*
- *queries* is the set of Query records owned by the project. Each Query record represents a saved query for the given project. See Query for details.
- *created* is the creation timestamp for the project, in ISO8601DateTIme format.

A project also represents a permission target at which access and authorization controls may be applied to teams associated with a project.

Project Usage - Project Usage is a subclass of Record that represents the use of a Project in the context of another Project. Project Usage is represented as a realization of Data, and has the following attributes:

- *usedProject* references the Project being used
- *usedProjectCommit* references the Commit of the Project being used

Commit - Commit is a subclass of Record that represents the changes made to a Project at a specific point in time in its lifecycle, such as the creation, update, or deletion of data in a Project. A Project has 0 or more Commits. A Commit has the following attributes:

- *timestamp* is the timestamp at which the Commit was created
- *owningProject* is the Project that owns the Commit.
- *previousCommit* is the set of immediately preceding Commits
- *change* is the set of Data Version records representing Data that is created, updated, or deleted in the Commit
- *versionedData* is the set of cumulative Data Version records in a Project at the Commit

Clarifications and Invariants:

- Commit.*versionedData* must indicate only the Data records that actually exist at the given Commit; all listed DataVersion records must have their payload attribute set to a non-null Data record.
- Commit.*change* indicates deletions by listing DataVersion records with their *payload* attribute set to null. This is only valid if at least one Commit in *previousCommits* contains a DataVersion with the same *identity* in its *versionedData* (i.e. only existing Data records may be deleted).

- DataVersion.*identity* is unique among records listed in Commit.*versionedData* and in Commit.*change*.
- A Commit must resolve all conflicts in its parent Commits: if the Commit C has two parent Commits C_a and C_b in C.*previousCommits*, where C_a.*versionedData* lists a DataVersion D_a but C.*versionedData* does not contain D_a (either the Data associated with D_a.*identity* is different, or not present at all), then C.*change* must list a DataVersion with D_a.*identity*

Commits are immutable. For a given Commit record, the value of Commit.*change* cannot be modified after a Commit has been created. If a modification is required, a new Commit record can be created with a different value of Commit.*change*.

Commits are not destructible[1]. A Commit record cannot be deleted during normal end-user operation. Commits represent the history and evolution of a Project. Deleting and mutating Commit records must be disabled for the normal end-user operations to preserve Project history.

[1]Note: A provider tool may provide administrative functions to repair the Commit graph of a Project but this is not considered a normal end-user operation.

Commit Reference - Commit Reference is an abstract subclass of Record that references a specific Commit (Commit Reference.*referencedCommit*). Project.*commit* is the set of all the Commit records for a given Project. Project.*commitRefererence* identifies specific Commit records in a Project that provide the context for navigating the Data in a Project. Two special types of Commit Reference are Branch and Tag, as described below.

Branch - Branch is an indirect subclass of Record (via CommitReference) that represents an independent line of development in a project. A Project can have 1 or more branches. When a Project is created, a default branch is also created. The default branch of a project can be changed, and a project can have only 1 default branch.

A Branch is a type of Commit Reference. A Branch is a pointer to a commit (*Branch.head*). The commit history of a Project on a given branch can be computed by recursively navigating *Commit.previousCommit*, starting from the head commit of the branch (*Branch.head*). A Branch has the following additional attributes:

- *creationTimestamp* is the timestamp at which the branch was created
- *deletionTimestamp* is the timestamp at which the branch was deleted
- *head* is the commit to which the branch is currently pointing. It represents the latest state of the project on the given branch.
- *owningProject* is the project that owns the given branch

Branches are immutable. Since a Branch is a pointer to a Commit, it can be updated to point to a different Commit. If a new Commit is created on a Project Branch, the value of *Branch.head* refers to that new Commit.

Branches are destructible under normal end-user operation. Branches can be deleted and merged with other branches.

Tag - Tag is an indirect subclass of Record (via Commit Reference) used for annotating specific commits-of-interest during Project development, such as for representing Project milestones, releases, baselines, or snapshots. A Project can have 0 or more tags.

A Tag is a type of Commit Reference. A Tag is a pointer to a commit (Tag.*taggedCommit*).

Tags are immutable. Tag.*taggedCommit* cannot be modified after a Tag record has been created. If Tag.*taggedCommit* needs to be modified to refer to a different Commit record, then the existing Tag can be deleted and a new Tag can be created with the same name and description.

Tags are destructible under normal end-user operation.

The table below summarizes the Mutability and Destruction semantics for Commit, Branch, and Tag.

Type of Record	Mutable	Destructible
Commit	No	No
Branch	Yes	Yes
Tag	No	Yes

7.1.3 ExternalData and ExternalRelationship

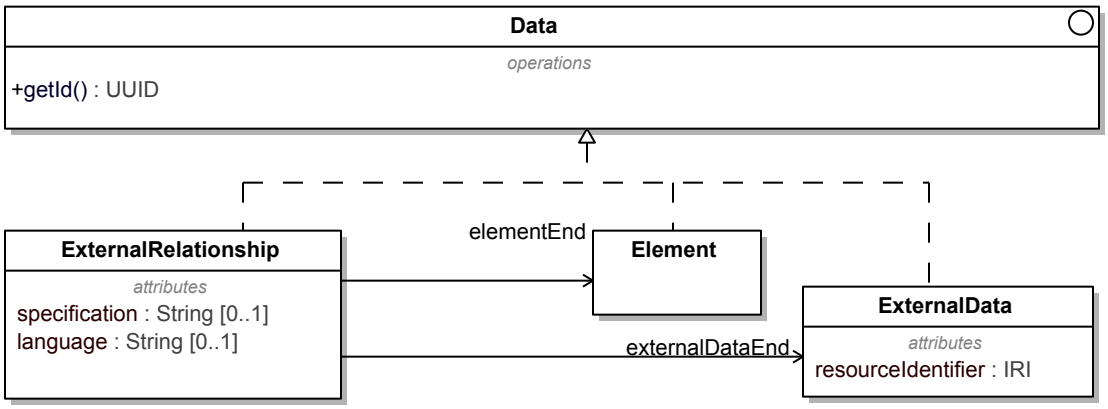

Figure 6. External Relationship API Model

The class diagram above presents concepts related to ExternalRelationship Service.

ExternalRelationship - ExternalRelationship is a realization of Data, and represents the relationship between a KerML Element [KerML] in a provider tool or repository to ExternalData in another tool or repository. The ExternalData may be a KerML Element or a non-KerML Element. A hyperlink between a KerML Element to a web resource is the most primitive example of an ExternalRelationship. An ExternalRelationship has the following attributes:

- *specification* is the formal representation of the semantics of the ExternalRelationship. The specification can be a collection of mathematical expressions. For example, an ExternalRelationship can be defined to map the attributes of a KerML Element to the attributes of an ExternalData. In this case, the specification would contain mathematical expressions, such as equations, representing the mapping. This is an optional attribute.
- *language* is the name of the expression language used for the specification. This is an optional attribute.

ExternalData - ExternalData is a realization of Data, and represents a resource external to a given tool or repository. ExternalData is defined only for the purpose of defining an ExternalRelationship. An ExternalData has the following additional attributes.

- *resourceIdentifier* is the IRI of the resource represented by the ExternalData

7.1.4 Query

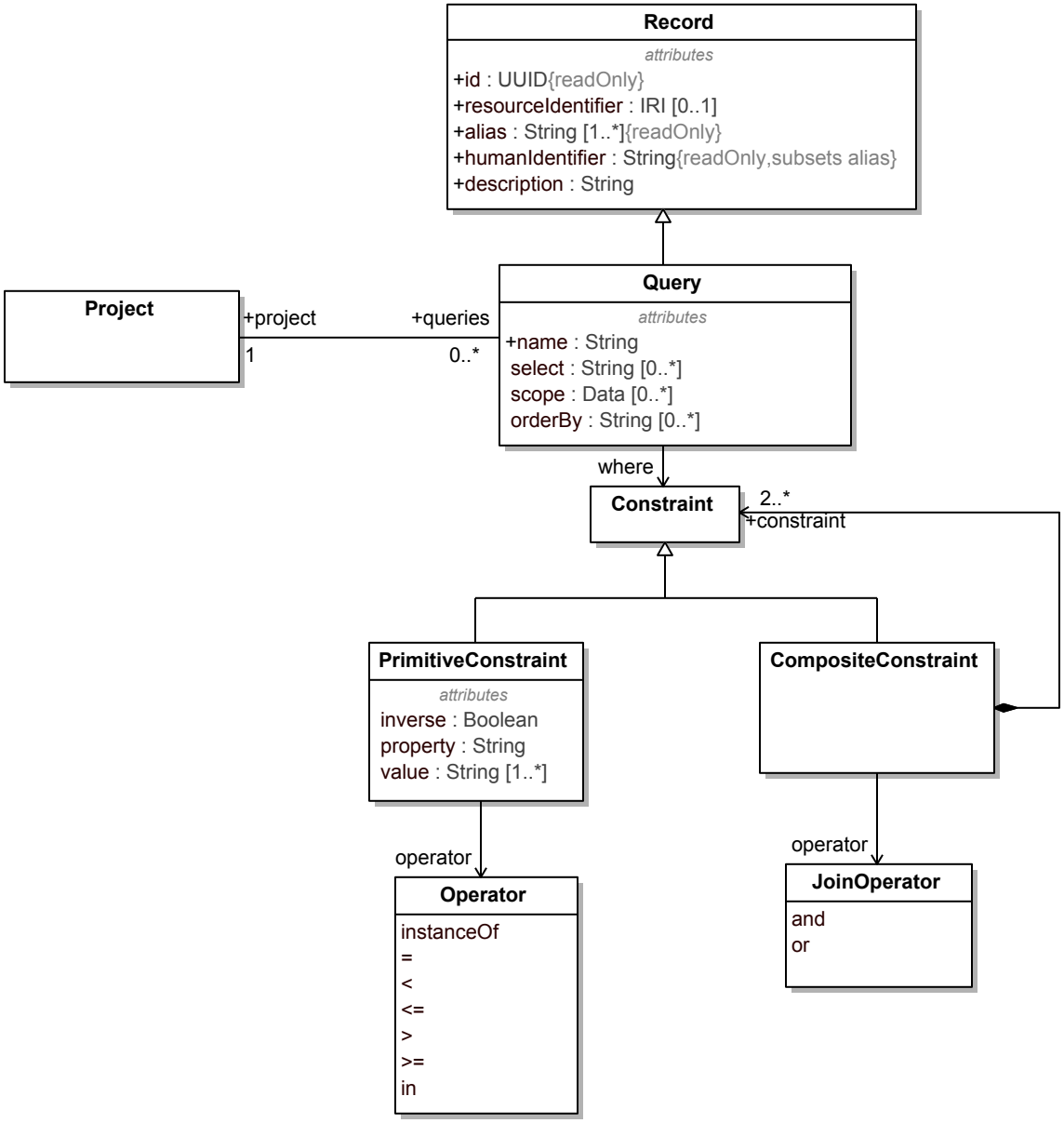

Figure 7. Query API Model

The class diagram above presents concepts related to the Query service.

Query - Query is a subclass of Record that represents a precise and language-independent request for information retrieval using the Systems Modeling API and Services. Query can be mapped to commonly used query languages, such as SQL, Gremlin, GraphQL, and SPARQL.

A Query record has the following attributes:

- *name* is the name of the Query

- *select* is a list of properties of Data (or its realizations) that will be included for each Data object in the query response. If no properties are specified, then all the properties will be included for each Data object in the query response.
- *scope* is a list of Data objects that define the scope context for query execution. The default scope of a Query is the owning Project.
- *where* is a Constraint that represents the conditions that Data objects in the query response must satisfy
- *orderBy* is a list of properties of Data (or its realizations) that are used for sorting the Data objects in the query response. The order of properties in the list governs the sorting order.

Constraint - Constraint is an abstract concept that represents conditions that must be satisfied by Data objects in the query response.

PrimitiveConstraint is a concrete subtype of Constraint that represents simple conditions that can be modeled using the *property-operator-value* tuple, e.g. *mass <= 4 kg.*, or *type instanceOf Generalization.* A PrimitiveConstraint has the following attributes:

- *property* is a property of Data (or its realizations) that is being constrained
- *operator* is of type Enumeration whose literals are mathematical operators, as shown in the figure above
- *value* is a list of primitive objects, such as String, Boolean, Integer, Double, and UUID
- *inverse* is of type Boolean. If true, a logical NOT operator is applied to the PrimitiveConstraint.

CompositeConstraint is a concrete subtype of Constraint that represents complex conditions composed of two or more Constraints using logical AND or OR operator. CompositeConstraint has the following attributes:

- *constraint* is the set of Constraints being composed
- *operator* is the logical operator for composing the Constraints

7.2 API Services

7.2.1 ProjectService

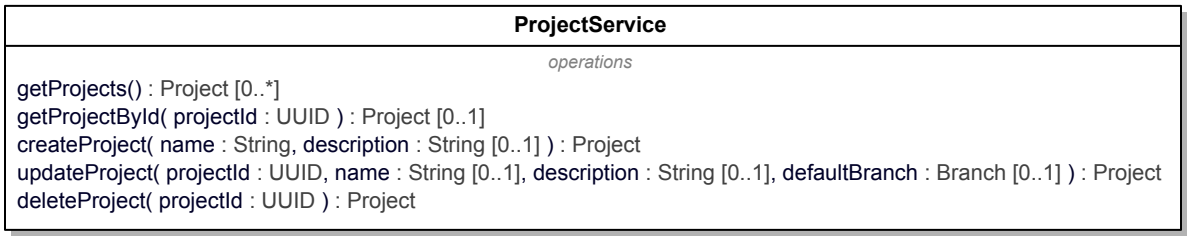

Figure 8. ProjectService Operations

Table 1. Operations

Name	Documentation
createProject	Create a new project with the given name and description (optional).
getProjectById	Get project with the given id (projectId).
updateProject	Update the project with the given id (projectId).
deleteProject	Delete the project with the given id (projectId).
getProjects	Get all projects.

7.2.2 ElementNavigationService

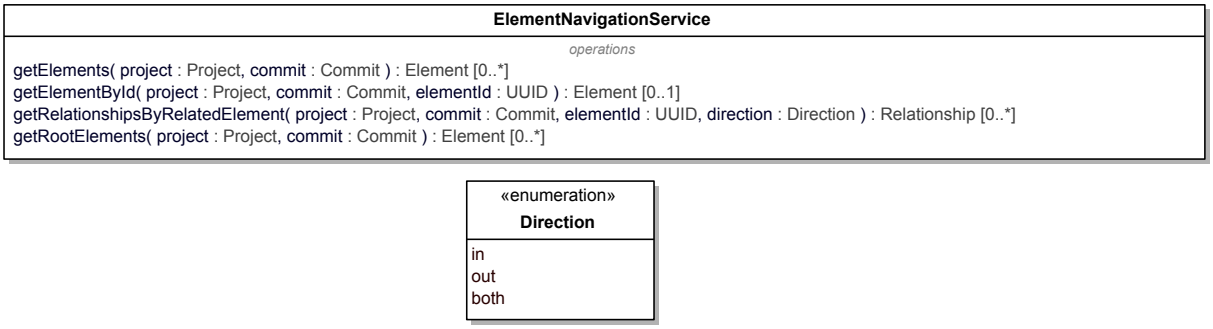

Figure 9. ElementNavigationService Operations

Element is the root metaclass in the KerML abstract syntax [KerML]. Relationship is a subtype of Element. Both Element and Relationship realize the Data interface defined in the API Model (refer to 7.1.2 - Project Data Versioning).

Table 2. Operations

Name	Documentation
getElementById	Get element with the given id (elementId) in the given project at the given commit.
getRootElements	Get all the root elements in the given project at the given commit.
getElements	Get all the elements in a given project at the given commit.
getRelationshipsByRelatedElement	Get relationships that are incoming, outgoing, or both relative to the given related element.

7.2.3 ProjectDataVersioningService

SYSMOAS_-78: DataVersion.identity multiplicity
SYSMOAS_-20: Description of deletion in createCommit row seems wrong

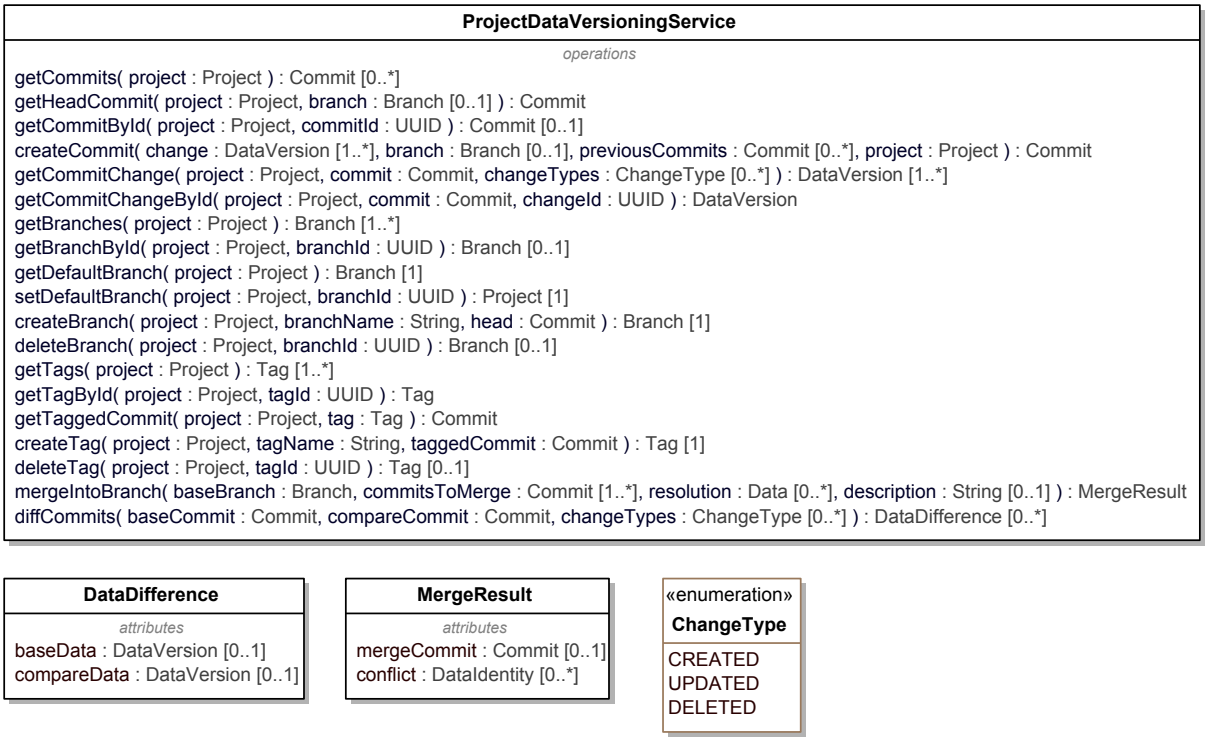

Figure 10. ProjectDataVersioningService Operations

Table 5. Operations

Name	Documentation
getBranches	Get all the branches in the given project.
getTaggedCommit	Get the tagged commit of the given tag in the given project.
getHeadCommit	Get the head commit of the given branch in the given project. If the branch is not specified, the default branch of the project is used.
getDefaultBranch	Get the default branch of the given project.
getCommitChangeById	Get the change with the given id (changeId) in the given commit of the given project. The changeId is the id of the DataVersion that changed in the commit.
getCommitById	Get the commit with the given id (commitId) in the given project.
setDefaultBranch	Set the branch with the given branchId as the default branch of the given project.
deleteTag	Delete the tag with the given id (tadId) in the given project.
getCommits	Get all the commits in the given project.

Name	Documentation
getCommitChange	Get the change in the given commit of the given project. The operation getCommitChange in ProjectDataVersioningService has an optional argument changeTypes that is a collection typed by the enumeration ChangeType with three literals (CREATED, UPDATED, DELETED). If the argument changeTypes is passed, then only the changes of the given type will be returned by the operation as DataVersion records. Some examples to elaborate this behavior are included below. If changeTypes = [], i.e. the argument is not specified, then the DataVersion records for all the data that was created, updated, or deleted in the given commit will be returned. If changeTypes = ['DELETED'], then the DataVersion records for all the data that was deleted in the given commit will be returned. If changeTypes = ['CREATED', 'UPDATED'], then the DataVersion records for all the data that was created or updated in the given commit will be returned.
getBranchById	Get the branch with the given id (branchId) in the given project.

Name	Documentation
createCommit	Create a new commit with the given change (collection of DataVersion records) in the given branch of the project. If the branch is not specified, the default branch of the project is used. Commit.change should include the following for each Data object that needs to be created, updated, or deleted in the new commit. (1) Creating Data - Commit.change should include a DataVersion record with DataVersion.payload populated with the Data being created. DataVersion.identity is not provided, either left empty, in which case a new DataIdentity needs to be created by the Service and assigned to DataVersion.identity in the new commit; or provided a brand new value (one that does not already exist in any of the *previousCommits*) by the client and accepted by the Service as is. (2) Updating Data - Commit.change should include a DataVersion record with DataVersion.payload populated with the updated Data. DataVersion.identity should be populated with the DataIdentity for which a new DataVersion record will be created in the new commit. (3) Deleting Data - Commit.change should include a DataVersion record with DataVersion.payload not provided, thereby indicating deletion of DataIdentity in the new commit. DataVersion.identity should be populated with the DataIdentity that will be deleted in the new commit. When a DataIdentity is deleted in a commit, all its versions (DataVersion) are also deleted, and any references from other DataIdentity are also removed to maintain data integrity. In addition, for Element Data (KerML), deletion of an Element must also result in deletion of incoming Relationships. When Element Data (KerML) is created or updated, derived properties must be computed or verified if the API provider claims Derived Property Conformance. The deleted element - DataIdentity and its DataVersion records - will be accessible in previous commits.
createBranch	Create a new branch with the given name (branchName) in the given project, and set the head of the new branch as the given commit (head).
deleteBranch	Delete the branch with the given id (branchId) in the given project.
getTags	Get all the tags in the given project.

Name	Documentation
mergeIntoBranch	Merge the given commits (commitsToMerge) in the given branch (baseBranch). The commits included in commitsToMerge may be commits referenced by a CommitReference, such as Branch.head or Tag.taggedCommit, or any other commit in the owning project (Project.commits). This operation returns a MergeResult which will include either of the following: (1) commit after the merge operation if successful, or (2) a set of DataIdentity records representing the merge conflicts if the merge operation is unsuccessful. Two optional inputs may be provided: (1) resolution as a set of Data that will resolve the merge conflicts, and (2) description of the merged commit if this operation is successful.

Name	Documentation
diffCommits	Get the difference between two commits - compareCommit and baseCommit. The set of all DataVersion records in a project at a given commit is accessible as Commit.versionedData. From a set theoretic perspective, this operation gets compareCommit.versionedData - baseCommit.versionedData and returns a DataDifference object with baseData and compareData for each difference. If any data is present in the compareCommit but absent in the baseCommit, DataDifference.compareData will include the corresponding DataVersion and DataDifference.baseData will be empty. If any data is absent in the compareCommit but present in the baseCommit, DataDifference.compareData will be empty and DataDifference.baseData will include the corresponding DataVersion. If any data is present in both but different in the compareCommit and baseCommit, DataDifference.compareData and DataDifference.baseData will include the corresponding DataVersion records. The operation diffCommits in ProjectDataVersioningService has an optional argument changeTypes that is a collection typed by the enumeration ChangeType with three literals (CREATED, UPDATED, DELETED). If the argument changeTypes is passed, then only the changes of the given type will be returned by the operation as DataDifference objects. Some examples to elaborate this behavior are included below. If changeTypes = [], i.e. the argument is not specified, then the DataDifference objects for all the data that was created, updated, or deleted in the compareCommit versus the baseCommit will be returned. If changeTypes = ['DELETED'], then the DataDifference objects for all the data that was deleted in the compareCommit versus the baseCommit will be returned. If changeTypes = ['CREATED', 'UPDATED'], then the DataDifference objects for all the data that was created or updated in the compareCommit versus the baseCommit will be returned.
createTag	Create a new tag with the given name (tagName) in the given project, and set the taggedCommit of the new tag as the given commit (taggedCommit).
getTagById	Get the tag with the given id (tagId) in the given project.

7.2.4 QueryService

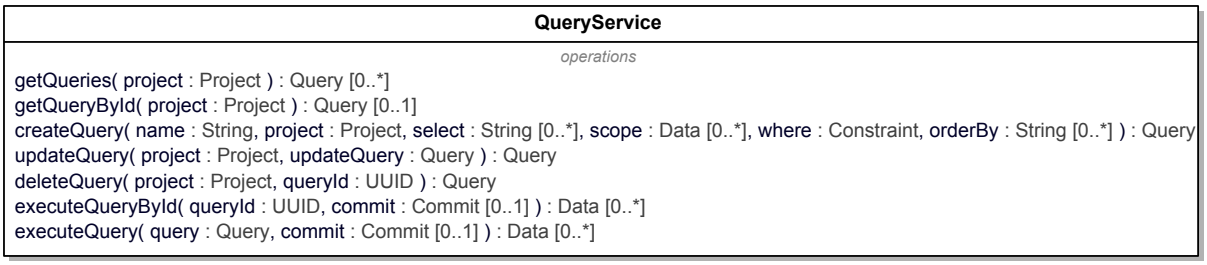

Figure 11. QueryService Operations

Table 6. Operations

Name	Documentation
updateQuery	Update the given query (updateQuery) in the given project.
getQueryById	Get the query with the given id (queryId) in the given project.
deleteQuery	Delete the query with the given id (queryId) in the given project.
executeQuery	Execute the given query in the owning project (Query.project) at the given commit. If the commit is not specified, then the head commit of the default branch of the project will be used.
executeQueryById	Execute the query with the given id in the owning project (Query.project) at the given commit. If the commit is not specified, then the head commit of the default branch of the project will be used.
getQueries	Get all the queries in the given project.
createQuery	Create a query in the given project with the given inputs.

7.2.5 ExternalRelationshipService

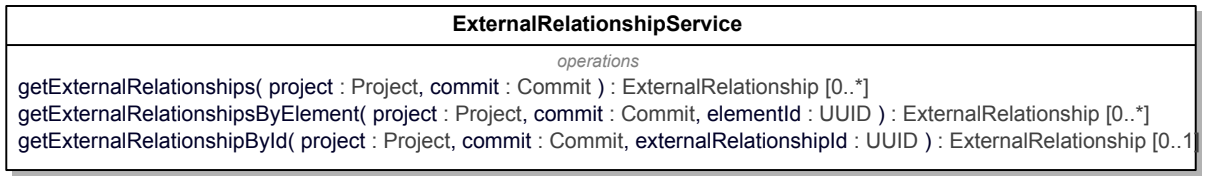

Figure 12. ExternalRelationshipService Operations

Table 7. Operations

Name	Documentation
getExternalRelationshipsByElement	Get all the external relationships in the given project at the given commit, where the id of elementEnd of the external relationship is the given elementId.

Name	Documentation
getExternalRelationshipById	Get the external relationship with the given id (externalRelationshipId).
getExternalRelationships	Get all the external relationships in a given project at a given commit.

7.2.6 ProjectUsageService

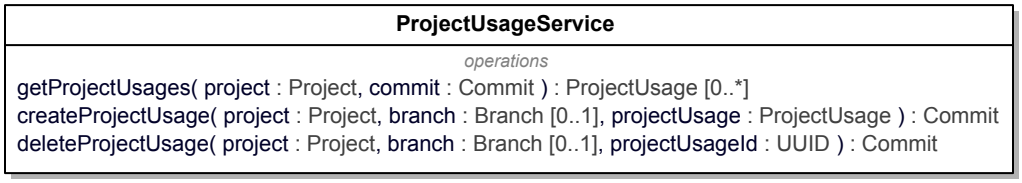

Figure 13. ProjectUsageService Operations

Table 8. Operations

Name	Documentation
deleteProjectUsage	Deletes the project usage with the given id (projectUsageId) from the given project at the head commit of the given branch. This operation returns a new commit where the given project usage does not exist, and sets the head of the given branch to the new commit. If a project branch is not given, then the default branch of the project will be used.
getProjectUsages	Get all the project usages in the given project at the given commit.
createProjectUsage	Create a new project usage in the given project at the head commit of the given branch. This operation returns a new commit that includes the new project usage, and sets the head of the given branch to the new commit. If a project branch is not given, then the default branch of the project will be used.

8 Platform Specific Models (PSMs)

8.1 REST/HTTP PSM

8.1.1 Overview

The REST/HTTP Platform-Specific Model (PSM) for the Systems Modeling API and Services is described using OpenAPI Specification (OAS) 3.1 and is included with this specification. The REST/HTTP PSM is described in the following sections.

- **PIM API Model - REST/HTTP PSM Model Mapping**: This section presents the mapping from the PIM API Model concepts to the JSON Models in the REST/HTTP PSM (OpenAPI specification).
- **PIM API Services - REST/HTTP PSM Endpoints Mapping**: This section presents the mapping from the PIM API Service definitions and operations to the API endpoints in the REST/HTTP PSM (OpenAPI specification).

8.1.2 PIM API Model - REST/HTTP PSM Model Mapping

The table below presents the mapping from the PIM API Model concepts to the JSON Models in the REST/HTTP PSM (OpenAPI specification).

Table 9. PIM API Model - REST/HTTP PSM Model Mapping Table

PIM Concept	REST/HTTP PSM Model (JSON)
Project	Project
Commit	Commit
Tag	Tag
Branch	Branch
Data	Data
DataIdentity	DataIdentity
DataVersion	DataVersion
Element	Element
Relationship	Relationship
ExternalData	ExternalData
ExternalRelationship	ExternalRelationship
ProjectUsage	ProjectUsage
Query	Query
PrimitiveConstraint	PrimitiveConstraint
CompositeConstraint	CompositeConstraint
DataDifference	DataDifference
MergeResult.mergeCommit	Commit
MergeResult.conflict	DataIdentity [0..*]

8.1.3 PIM API Services - REST/HTTP PSM Endpoints Mapping

SYSMOAS_-78: DataVersion.identity multiplicity

The table below presents the mapping between the PIM Services to the REST/HTTP PSM Endpoints. This is followed by a detailed description of the pagination strategy used by the REST/HTTP PSM.

Table 10. PIM to REST / HTTP PSM Mapping

PIM Service	REST / HTTP PSM Endpoint
ProjectService	
createProject	POST /projects
getProjects	GET /projects
getProjectById	GET /projects/{projectId}
updateProject	PUT /projects/{projectId}
deleteProject	DELETE /projects/{projectId}
ElementNavigationService	
getElements	GET/projects/{projectId}/commits/{commitId}/elements
getElementById	GET /projects/{projectId}/commits/{commitId}/elements/{elementId}
getRelationshipsByRelatedElement	GET /projects/{projectId}/commits/{commitId}/elements/{relatedElementId}/relationships • *direction* query parameter with allowable values {in, out, both}
getRootElements	GET /projects/{projectId}/commits/{commitId}/roots
ProjectDataVersioningService	
getCommits	GET /projects/{projectId}/commits
getHeadCommit	• GET /projects/{projectId}/branches/{branchId} returns the branch with the given branch ID. If the branch ID is not provided, then the ID of the default branch of the project is used. Use the following steps to get the ID of the default branch. ◦ GET /projects/{projectId} return the project with the given project ID ◦ Project.defaultBranch provides the ID of the default branch of the project • Branch.head provides the ID of the head commit of the branch • GET /projects/{projectId}/commits/{commitId} returns the head commit with the the given commit ID
getCommitById	GET /projects/{projectId}/commits/{commitId}

PIM Service	REST / HTTP PSM Endpoint
createCommit	POST /projects/{projectId}/commit • The body of the POST request is a CommitRequest. • The content of CommitRequest.change for creating, updating, and deleting Data maps directly to the corresponding PIM operation (createCommit), and is described below. ◦ For creating new Data, CommitRequest.*change* should include a DataVersion where DataVersion.*payload* includes the Data being created and DataVersion.*identity* is either not specified or set to a new DataIdentity that does not already exist in any of the *previousCommits*. ◦ For updating existing Data, CommitRequest.*change* should include a DataVersion where DataVersion.*payload* includes the updated Data, and DataVersion.*identity* is the DataIdentity for which a new DataVersion will be created in the commit. ◦ For deleting existing Data, CommitRequest.*change* should include a DataVersion where DataVerision.*payload* is not specified, and DataVersion.*identity* is the DataIdentity that will be deleted in the commit.
getCommitChange	GET /projects/{projectId}/commits/{commitId}/changes
getCommitChangeById	GET /projects/{projectId}/commits/{commitId}/changes/{changeId}
getBranches	GET /projects/{projectId}/branches
getBranchById	GET /projects/{projectId}/branches/{branchId}
getDefaultBranch	• GET /projects/{projectId} returns a Project with the given ID (projectId) • Project.*defaultBranch* provides the ID of the default branch of the project
setDefaultBranch	• PUT /projects/{projectId} • The body of the PUT request is a ProjectRequest. Set the ID of the new default branch as ProjectRequest.*defaultBranch* in the body.
createBranch	POST /projects/{projectId}/branches
deleteBranch	DELETE /projects/{projectId}/branches/{branchId}
getTags	GET /projects/{projectId}/tags
getTagById	GET /projects/{projectId}/tags/{tagId}

PIM Service	REST / HTTP PSM Endpoint
getTaggedCommit	• GET /projects/{projectId}/tags/{tagId} returns the tag with the given ID (tagId) • Tag.taggedCommit provides the ID of the tagged commit • GET /projects/{projectId}/commits/{commitId} returns the tagged commit given its ID (see the previous step)
createTag	POST /projects/{projectId}/tags
deleteTag	DELETE /projects/{projectId}/tags/{tagId}
mergeIntoBranch	POST /projects/{projectId}/branches/{targetBranchId}/merge
diffCommits	GET /projects/{projectId}/commits/{compareCommitId}/diff
QueryService	
getQueries	GET /projects/{projectId}/queries
getQueryById	GET /projects/{projectId}/queries/{queryId}
createQuery	POST /projects/{projectId}/queries
updateQuery	PUT /projects/{projectId}/queries/{queryId}
deleteQuery	DELETE /projects/{projectId}/queries/{queryId}
executeQueryById	GET /projects/{projectId}/queries/{queryId}/results
executeQuery	GET /projects/{projectId}/query-results POST /projects/{projectId}/query-results Either the GET or the POST endpoint may be used. The POST endpoint is provided for compatibility with clients that don't support GET requests with a body
ExternalRelationshipService	

PIM Service	REST / HTTP PSM Endpoint

getExternalRelationships	• POST /projects/{projectId}/queries with QueryRequest JSON model
	  ◦ Query.*where* is set to a PrimitiveConstraint
	  ◦ PrimitiveConstraint.*property* = @type
	  ◦ PrimitiveConstraint.*value* = 'ExternalRelationship'
	  ◦ PrimitiveConstraint.*operator* = '='
	• Execute the query with the following request
	  ◦ GET /projects/{projectId}/queries/{queryId}/results?commitId={commitId}, where {projectId} and {commitId} are the ids of the given Project and Commit

getExternalRelationshipsByElement	• POST /projects/{projectId}/queries with QueryRequest JSON model
	  ◦ Query.*where* is set to a CompositeConstraint
	  ◦ CompositeConstraint.*constraints* includes the following 2 instances of PrimitiveConstraint with the *and* operator
	  ▪ PrimitiveConstraint 1
	  ▪ PrimitiveConstraint.*property* = @type
	  ▪ PrimitiveConstraint.*value* = 'ExternalRelationship'
	  ▪ PrimitiveConstraint.*operator* = '='
	  ▪ PrimitiveConstraint 2
	  ▪ PrimitiveConstraint.*property* = elementEnd
	  ▪ PrimitiveConstraint.*value* = {elementId}
	  ▪ PrimitiveConstraint.*operator* = '='
	• Execute the query with the following request
	  ◦ GET /projects/{projectId}/queries/{queryId}/results?commitId={commitId}, where {projectId} and {commitId} are the ids of the given Project and Commit

PIM Service	REST / HTTP PSM Endpoint
getExternalRelationshipsById	• POST /projects/{projectId}/queries with Query JSON model ◦ Query.*where* is set to a CompositeConstraint ◦ CompositeConstraint.*constraints* includes the following 2 instances of PrimitiveConstraint with the *and* operator ▪ PrimitiveConstraint 1 ▪ PrimitiveConstraint.*property* = @type ▪ PrimitiveConstraint.*value* = 'ExternalRelationship' ▪ PrimitiveConstraint.*operator* = '=' ▪ PrimitiveConstraint 2 ▪ PrimitiveConstraint.*property* = @id ▪ PrimitiveConstraint.*value* = {externalRelationshipId} ▪ PrimitiveConstraint.*operator* = '=' • Execute the query with the following request ◦ GET /projects/{projectId}/queries/{queryId}/results?commitId={commitId}, where {projectId} and {commitId} are the ids of the given Project and Commit
Project Usage Service	

PIM Service	REST / HTTP PSM Endpoint
getProjectUsages	• POST /projects/{projectId}/queries with QueryRequest JSON model ◦ Query.*where* is set to a PrimitiveConstraint ◦ PrimitiveConstraint.*property* = @type ◦ PrimitiveConstraint.*value* = 'ProjectUsage' ◦ PrimitiveConstraint.*operator* = '=' • Execute the query with the following request ◦ GET /projects/{projectId}/queries/{queryId}/results?commitId={commitId}, where {projectId} and {commitId} are the ids of the given Project and Commit
createProjectUsage	• POST /projects/{projectId}/commits?branchId={branchId} with CommitRequest JSON model, such that: ◦ CommitRequest.*change* = DataVersion with the following inputs ▪ DataVersion.*payload* = ProjectUsage with the following inputs ▪ ProjectUsage.*usedProjectCommit* = {commitId} of the Project and Commit being used.
deleteProjectUsage	• POST /projects/{projectId}/commits?branchId={branchId} with CommitRequest JSON model, such that: ◦ CommitRequest.*change* = DataVersion with the following inputs ▪ DataVersion.*identity* = DataIdentity with the following inputs ▪ DataIdentity.*id* = {projectUsageId} ▪ DataVersion.*payload* = null

Pagination

The REST/HTTP PSM uses a Cursor-based pagination strategy for the responses received from the GET requests. The following 3 query parameters can be specified in any GET request that returns a collection of records.

1. *page[size]* specifies the maximum number of records that will be returned per page in the response
2. *page[before]* specifies the URL of the page succeeding the page being requested
3. *and page[after]* specifies the URL of a page preceding the page being requested

If neither *page[before]* nor *page[after]* is specified, the first page is returned with the same number of records as specified in the *page[size]* query parameter. If the *page[size]* parameter is not specified, then a default page size is used, which can be set by the API provider.

The *Link* header in the response includes links (URLs) to the previous page and the next page, if any, for the given page in the response. The specification of these links is conformant to the IETF Web Linking standard. As an example, the value of the *Link* response header is shown below. The *rel* value associated with each page link specifies the type of relationship the linked page has with the page returned in the response. Page link specified with

rel value as *next* is the link for the next (or succeeding) page to the page returned in the response, and the page link specified with *rel* value as *prev* is the link for the previous (or preceding) page to the page returned in the response.

```
<http://sysml2-api-host:9000/projects?
    page[after]=MTYxODg2MTQ5NjYzMnwyMDEwOWY0MC00ODI1LTQxNmEtODZmNi03NTA4YWM0MmEwMjE&
    page[size]=3>; rel="next",
<http://sysml2-api-host:9000/projects?
    page[before]=MTYxODg2MTQ5NjYzMnwxMDg2MDFjMS1iNzk1LTRkMGEtYTFiYy1lZjEyYmMwNTU5ZjI&
    page[size]=3>; rel="prev"
```

Example

An example demonstrating the Cursor-based paginated responses received from GET requests to the */projects* endpoint is presented here. The term "User" in the example scenario presented below refers to an API user that could be a human user or a software program.

Step 1 - User makes a GET request to the */projects* endpoint with *page[size]* query parameter set to 3. If successful, this request will return the first page with a maximum of 3 project records. The URL for this GET request is shown below.

```
http://sysml2-api-host:9000/projects?page[size]=3
```

Step 2 - If there are more than 3 projects in the provider repository, the *Link* header in the response will provide the URL for the next page with *rel* value equal to *next.* The User gathers the link to the next page.

```
<http://sysml2-api-host:9000/projects?
    page[after]=MTYxODg2MjE2NTMxNXwwOGY0MzNkYi1iNmQ0LTQxYjgtOTAyMC1lODIwZWJjNDE3YmU&
    page[size]=3>; rel="next"
```

Step 3 - User makes a GET request to the URL for the next page gathered from Step 2. The *Link* header in the response will provide the URL for the next page with *rel* value equal to *next*. Additionally, the *Link* header will include the URL for the previous page with *rel* value equal to *prev*.

```
<http://sysml2-dev.intercax.com:9000/projects?
    page[after]=MTYxODg2MjY4OTYxNHwyMDEwOWY0MC00ODI1LTQxNmEtODZmNi03NTA4YWM0MmEwMjE&
    page[size]=3>; rel="next",
<http://sysml2-dev.intercax.com:9000/projects?
    page[before]=MTYxODg2MjY4OTYxNHwxMDg2MDFjMS1iNzk1LTRkMGEtYTFiYy1lZjEyYmMwNTU5ZjI&
    page[size]=3>; rel="prev"
```

Step 4 - User continues Step 3 until the *Link* header in the response does not include the URL for the next page (*rel* value as *next*).

8.2 OSLC 3.0 PSM

8.2.1 Overview

The OSLC Platform-Specific Model (PSM) for the Systems Modeling API and Services is described using OpenAPI Specification (OAS) 2.0 and is included with this specification.

Note that the URLs listed in the OpenAPI Specification are provided as examples only. With OSLC, all URLs are implementation-specific. A OSLC client typically relies on the OSLC discovery mechanism (https://docs.oasis-open-projects.org/oslc-op/core/v3.0/ps01/discovery.html), to determine what services are provided by an OSLC server, as well as the necessary information (such as a service URL) to be able to consume any such service. At the least, an OSLC client requires a discovery URL to bootstrap this discovery for any particular server. The various approaches for bootstrapping and discovery are further detailed in the OSLC standard.

- 8.2.2 presents a brief introduction to OSLC and its nomenclature.

- 8.2.3 presents the mapping of PIM concepts to OSLC resource types
- 8.2.4 presents the mapping of PIM services and operations to OSLC services

An OSLC implementation may typically need to realize a broader set of services than those defined by the PIM for full integration with other OSLC-compliant systems. Services such as Delegated UI for Selection and Creation, resource UI Preview, authentication, and support for arbitrary queries (beyond those defined in the PIM).

8.2.2 OSLC Nomenclature

What is OSLC?

Open Services for Lifecycle Collaboration (OSLC) is an open community creating specifications for integrating tools. OSLC specifications allow conforming independent software and product lifecycle tools to integrate their data and workflows in support of end-to-end lifecycle processes. OSLC is based on the W3C Linked Data and the use of RDF to represent artifacts using common vocabularies, and HTTP to discover, create, read, update, and delete such artifacts. For a more comprehensive introduction, see the OSLC Primer and the OSLC specifications at https://open-services.net/specifications/.

OSLC servers may support any or all of the following:

1. *Creation factories* for creating a resource of some RDF type associated with the factory. For example, a client might create a new change request by an HTTP POST including the RDF representation of the change request to be created to the URI of a change request creation factory.
2. REST services to read, update, and/or delete resources at the resource's URI.
3. *Query capabilities* that allow OSLC clients to query for resources of an RDF type associated with the query capability. For example, a client might query for change requests by an HTTP GET or POST to the query base URI of a query capability for change requests. See OSLC Query Version 3.0 for further details.
4. *Creation dialog* that allows some other application to embed it in an iFrame of an application dialog that allows a user to fill in information and create a resource of some type associated with the creation dialog.
5. *Selection dialog* that allows an application to display it to select a resource of some type associated with the dialog in order to create and persist a link to that resource in the application.
6. *Resource preview*, such as a pop-up display, that is shown as a rich hover when a user hovers over a link to a resource managed by the OSLC server.

OSLC Discovery

OSLC clients and other servers discover the OSLC capabilities offered by a server through OSLC discovery. This allows clients to discover the URIs of creation factories, query capabilities, creation dialogs, and selection dialogs without the need for hard coding or constructing URIs for them. Discovery starts with a known URI for an OSLC *Service Provider Catalog*. That service provider catalog may reference zero, one, or many *service providers*. Servers that support the concept of project as a container of resources, often for access control, often define a service provider for each such container. A service provider may declare one or many services, each of which may define the creation factories, query capabilities, creation dialogs, and selection dialogs supported by that service. For more details, see OSLC Core Version 3.0. Part 2: Discovery.

A creation factory for a specific RDF type is discovered by:

1. Start with a known OSLC service provider catalog URI, perform HTTP GET.
2. Get the URIs of service providers from the response.
3. For each service provider, perform HTTP GET.
4. From the response, look for services described in the response data, that declare a creation factory for the RDF type.
5. Get the URI of the creation factory.

A query capability for a specific RDF type is discovered by:

1. Start with a known OSLC service provider catalog URI, perform HTTP GET.
2. Get the URIs of service providers from the response.
3. For each service provider, perform HTTP GET.
4. From the response, look for services described in the response data, that declare a query capability for the RDF type.
5. Get the query base URI of the query capability.

OSLC Resource Shapes

Resource shapes specify a standard way (using RDF) of describing resources of specific RDF types and their properties. For more details, see OSLC Core Version 3.0. Part 6: Resource Shape. Resource shapes may be discovered in a number of ways:

1. The definition of a creation factory may reference a resource shape that describes the properties that might be included in the RDF content POSTed to that creation factory.
2. The definition of a query capability may reference a resource shape that describes the properties of the query results and references a shape that describes the properties that might be queried as a condition, or selected to be included in the results.
3. A resource may reference an instance shape that describes the properties of that resource.

Linked Data Platform Containers

Linked Data Platform Containers (LDPC) are a way of representing containers as an RDF resource. OSLC specifications specify LPDCs as a container representation. See W3C Linked Data Platform 1.0 for further information.

OSLC Service Providers

A "global" service provider will contain one or more services that cross all systems modeling projects. A service provider will be declared for each systems modeling project that provides capabilities specific to that project.

RDF Media Types

OSLC recommends that servers support RDF/XML (application/rdf+xml), Turtle (text/turtle, application/x-turtle), and JSON-LD (application/ld+json).

8.2.3 PIM API Model – OSLC PSM Resource Mapping

The mapping from the PIM API Model to the OSLC PSM resource types includes the following.

1. Mapping from the KerML and SysML abstract syntax (Element and subtypes) to OSLC resource shapes and vocabulary. The package containing the resulting OSLC resource shapes and vocabulary is included with this specification - see ***OSLC_Systems_Modeling_Resource_Shapes_and_Vocabulary.zip***.
2. Mapping from the API Model concepts to resource types in other OSLC specifications, such as OSLC Configuration Management specification and OSLC Query specification. This mapping is shown in the table below. References to the OSLC specifications mentioned in the table below are as follows.
    1. OSLC Configuration Management refers to OSLC Configuration Management Version 1.0 available at https://oslc-op.github.io/oslc-specs/specs/config/oslc-config-mgt.html.
    2. OSLC Query refers to OSLC Query Version 3.0 available at https://docs.oasis-open-projects.org/oslc-op/query/v3.0/os/oslc-query.html

Table 11. PIM Concept to OSLC Resource type Mapping

PIM Concept	OSLC Resource Type (OSLC Specification)	`dcterms:identifier`
Project	Component (oslc_config:Component) *(OSLC Configuration Management*)	PIM project id
Branch	Stream (oslc_config:Stream) *(OSLC Configuration Management*)	PIM branch id
Tag	Baseline (oslc_config:Baseline) *(OSLC Configuration Management*)	PIM tag id
Commit	Not supported by OSLC Configuration Management.	PIM commit id
DataIdentity	Concept resource *(OSLC Configuration Management*)	PIM Data Identity id
DataVersion	Version resource (oslc_config:VersionResource) *(OSLC Configuration Management*)	PIM Data Version id
Query	OSLC Query (*OSLC Query*)	
PrimitiveConstraint	*oslc.where* in OSLC Query (*OSLC Query*)	
CompositeConstraint	*oslc.where* in OSLC Query (*OSLC Query*)	
ProjectUsage	Not Available. PIM Project concept is mapped to OSLC Component but OSLC does not define Component Usage.	
DataDifference	Not Available	
MergeResult	Not Available	

8.2.4 PIM API Services – OSLC PSM Service Mapping

The table below presents the mapping between the PIM Services to the OSLC 3.0 PSM.

Table 12. PIM API Services - OSLC Services Mapping

PIM Services	OSLC PSM
ProjectService	

getProjectById	In OSLC, a Project is identified by its URL. So simply perform a GET on the Project URL.
	To search for a project with a given dcterms:identifier (or any other property that is deemed to return a single query result):
	
	1. Discover a query capability for components (oslc_config:Component). See *OSLC Discovery* section.
	2. Perform a GET on the query base, specifying a query for the dcterms:identifier in a *oslc.where* query parameter, and specifying which properties, if any, of the component should be returned in the query result using an *oslc.select* query parameter.
	
	Example: Query for components with identifier 123, returning all properties.
	
	`GET queryBaseUri?`
	`oslc.where=dcterms%3Aidentifier%3D%22123%22&`
	`oslc.select=*`

createProject	1. Discover a creation factory for components (oslc_config:Component). See *OSLC Discovery* section. 2. POST the RDF content describing the component (with a Content-Type header set to an RDF media type supported by the server) to the creation factory. The RDF content should be compliant with the resource shape specified for the creation factory, and that resource shape must be compatible with the resource shape for Component as described in the OSLC specification (see https://docs.oasis-open-projects.org/oslc-op/config/v1.0/ps01/config-resources.html#ComponentShape). Example: `POST creationFactoryUri`

getProjects	1. Discover a query capability for components (oslc_config:Component). See *OSLC Discovery* section.
	2. Perform a GET on the query base, specifying which properties, if any, of the component should be returned in the query result using an *oslc.select* query parameter.
	
	Example: Query for components returning their name (dcterms:title) and id (dcterms:identifier)
	
	`GET queryBaseUri?`
	`oslc.select=dcterms%3Atitle%2Cdcterms%3Aidentifier`

updateProject	Execute an HTTP PUT with the revised content of the component. The RDF content should be compliant with the instance resource shape of the Component, and this should be compatible with the resource shape for Component as described in the OSLC specification (see https://docs.oasis-open-projects.org/oslc-op/config/v1.0/ps01/config-resources.html#ComponentShape)

PIM Services	OSLC PSM
deleteProject	Not supported directly. Deletion operation is not supported for Components in the OSLC Configuration Management specification. See https://oslc-op.github.io/oslc-specs/specs/config/config-resources.html#componentoperations. However, providers may implement deletion operation for Components.
ElementNavigation Service	

getElements	1. Discover a query capability for elements (an API-specific RDF type) for the specified project. See *OSLC Discovery* section.
	2. Perform a GET on the query base, specifying which properties, if any, of the commit should be returned in the query result using an *oslc.select* query parameter.
	
	Example: Query for elements returning name (dcterms:title) and identifier (dcterms:identifier).
	
	`GET queryBaseUri?`
	`oslc.select=dcterms%3Atitle%2Cdcterms%3Aidentifier`

getElementById	In OSLC, an Element is identified by its URL. So simply perform a GET on the Element URL.
	To search for an Element with a given dcterms:identifier (or any other property that is deemed to return a single query result):
	
	1. Discover a query capability for elements (an API-specific RDF type) for the specified project. See *OSLC Discovery* section.
	2. Perform a GET on the query base, specifying a query for the dcterms:identifier in a *oslc.where* query parameter, and specifying which properties, if any, of the commit should be returned in the query result using an *oslc.select* query parameter.
	
	Example: Query for elements with identifier 123, returning all properties.
	
	`GET queryBaseUri?`
	`oslc.where=dcterms%3Aidentifier%3D%22123%22&`
	`oslc.select=*`

PIM Services	OSLC PSM

getRelationshipsByRelatedElement	1. Discover the query capability for Elements for the particular project (and its specific commit). See *OSLC Discovery* section.
	2. Perform a GET on the query base, setting the *oslc.where* query parameter to:
	1. *sysml:out* = URI of the subject element to get all the outgoing Relationships from that element
	2. *sysml:in* = URI of the subject element to get all the incoming Relationships to the element
	
	Example 1: Query for Relationships outgoing from an element <http://sysml2.server.com/elements/123>:
	
	`GET queryBaseUri?`
	`oslc.where=sysml:out=<http://sysml2.server.com/elements/123> and`
	`rdf:type=<http://omg.org/ns/sysml/v2/metamodel%23Relationship>`
	`&oslc.prefix=sysml=<http://omg.org/ns/sysml/v2/metamodel%23>`
	
	Example 2: Query for Relationships incoming to an element <http://sysml2.server.com/elements/123>:
	
	`GET queryBaseUri?`
	`oslc.where=sysml:in=<http://sysml2.server.com/elements/123> and`
	`rdf:type=<http://omg.org/ns/sysml/v2/metamodel%23Relationship>`
	`&oslc.prefix=sysml=<http://omg.org/ns/sysml/v2/metamodel%23>`
	
	For Direction equal to *both*: perform and merge the separate queries on *in* and *out* values.

getRootElements	Not Available. OSLC Query does not currently support the ability to search for resources, where certain properties (owner in this case) are not set.
ProjectDataVersioning Service	

getBranchById	1. Discover a query capability for streams (oslc_config:Stream) for the specified project. See *OSLC Discovery* section.
	2. Perform a GET on the query base, specifying a query for the dcterms:identifier in a *oslc.where* query parameter, and specifying which properties, if any, of the component should be returned in the query result using an *oslc.select* query parameter.
	
	Example: Query for streams with identifier 123, returning all properties.
	
	`GET queryBaseUri?`
	`oslc.where=dcterms%3Aidentifier%3D%22123%22&`
	`oslc.select=*`

PIM Services	OSLC PSM

getTagById	1. Discover a query capability for baselines (oslc_config:Baseline) for the specified project. See *OSLC Discovery* section.
	2. Perform a GET on the query base, specifying a query for the dcterms:identifier in a *oslc.where* query parameter, and specifying which properties, if any, of the component should be returned in the query result using an *oslc.select* query parameter.
	
	Example: Query for baselines with identifier 123, returning all properties.
	
	`GET queryBaseUri?`
	`oslc.where=dcterms%3Aidentifier%3D%22123%22&`
	`oslc.select=*`

getCommits	Not available. The OSLC Configuration Management specification does not define any notion of a commit. Versions of specific resources may be fetched.

getBranches	1. Discover a query capability for streams (oslc_config:Stream) for the specified project. See *OSLC Discovery* section.
	2. Perform a GET on the query base, specifying which properties, if any, of the stream should be returned in the query result using an *oslc.select* query parameter.
	
	Example: Query for branches returning their name (dcterms:title) and id (dcterms:identifier)
	
	`GET queryBaseUri?`
	`oslc.select=dcterms%3Atitle%2Cdcterms%3Aidentifier`

PIM Services	OSLC PSM

createCommit	Not available. The OSLC Configuration Management specification does not define any notion of a commit. New versions may be created in the context of a stream or change set. However, the OSLC Configuration Management specification does not define any means to commit a change set or deliver the changes in a stream or change set to another stream.
	For creating new versions of a resource in the context of a stream or a change set, see below.
	
	1. Execute a PUT on version resource concept URI with configuration context parameter/header set to the URI of a stream (branch). Body contains the updated RDF representation of the element version.
	
	Note that this only supports a new commit along a branch with the previous latest commit being its previous commit. If a client wants to commit from an earlier version, they must first create a stream (branch) from that earlier baseline (commit) and then use a PUT with that new stream.
	The OSLC Configuration Management specification does not currently define any mechanisms for creating new versions of multiple versioned resources in a single REST operation.
	Example 1:
	
	`PUT conceptResourceUri?`
	`oslc_config.context=urlEncodedStreamUri`
	
	Example 2:
	
	`Headers: Configuration-Context=streamUri`
	`PUT conceptResourceUri`

getCommitById	1. Discover a query capability for commits (a API-specific RDF type) for the specified project. See *OSLC Discovery* section.
	2. Perform a GET on the query base, specifying a query for the dcterms:identifier in a *oslc.where* query parameter, and specifying which properties, if any, of the commit should be returned in the query result using an *oslc.select* query parameter.
	
	Example: Query for commits with identifier 123, returning all properties.
	
	`GET queryBaseUri?`
	`oslc.where=dcterms%3Aidentifier%3D%22123%22&`
	`oslc.select=*`

getCommitChange	Not available. The OSLC Configuration Management specification does not define any notion of a commit. New versions may be created in the context of a stream or change set. However, the OSLC Configuration Management specification does not define any means to commit a change set or deliver the changes in a stream or change set to another stream.
getCommitChangeById	Not available. The OSLC Configuration Management specification does not define any notion of a commit. New versions may be created in the context of a stream or change set. However, the OSLC Configuration Management specification does not define any means to commit a change set or deliver the changes in a stream or change set to another stream.

PIM Services	OSLC PSM
deleteBranch	Execute an HTTP DELETE on the URI of the stream. Note that servers may reject the delete request if the stream is used or referenced by other configurations. A server may also support archiving (soft delete) a stream using a HTTP PUT with content that includes a oslc:archived "true"^^xsd:boolean property.
getDefaultBranch	Not available. The OSLC Configuration Management specification does not define any notion of a default stream.
deleteTag	Execute an HTTP DELETE on the URI of the baseline. Note that a server might reject a request to delete a baseline if it is used or referenced by other configurations. A server may also support archiving (soft delete) a baseline using a HTTP PUT with content that includes a oslc:archived "true"^^xsd:boolean property.
setDefaultBranch	Not available. The OSLC Configuration Management specification does not define any notion of a default stream.
getHeadCommit	Not available. The OSLC Configuration Management specification does not define any notion of a commit. New versions may be created in the context of a stream or change set. However, OSLC Configuration Management does not define any means to commit a change set or deliver the changes in a stream or change set to another stream.
getTaggedCommit	Not available. The OSLC Configuration Management specification does not define any notion of a commit. New versions may be created in the context of a stream or change set. However, OSLC Configuration Management does not define any means to commit a change set or deliver the changes in a stream or change set to another stream. A baseline can provide a set of versioned resources but it is not a commit.

getTags	1. Discover a query capability for baselines (oslc_config:Baseline) for the specified project. See OSLC Discovery section.
	2. Perform a GET on the query base, specifying which properties, if any, of the baseline should be returned in the query result using an oslc.select query parameter.
	
	Example: Query for tags returning their name (dcterms:title) and id (dcterms:identifier)
	
	`GET queryBaseUri?`
	`oslc.select=dcterms%3Atitle%2Cdcterms%3Aidentifier`

createTag	1. Get the baselines LDPC URI from the RDF of a stream. 2. POST the RDF representation of the baseline to the LDPC URI (with a Content-Type header set to an RDF media type supported by the server). The RDF content should be compatible with the resource shape for Baseline as described in the specification (see https://docs.oasis-open-projects.org/oslc-op/config/v1.0/ps01/config-resources.html#BaselineShape) Example: Create a baseline from a stream `POST baselinesLdpcUri`

PIM Services	OSLC PSM

createBranch	A server might support either or both of the following:
	A) Use a creation factory:
	
	1. Discover a creation factory for streams (oslc_config:Stream) for the specified project. See *OSLC Discovery* section.
	2. POST the RDF content describing the stream (with a Content-Type header set to an RDF media type supported by the server) to the creation factory. The RDF content should be compliant with the resource shape specified for the creation factory, and that resource shape must be compatible with the resource shape for Stream as described in the specification (see https://docs.oasis-open-projects.org/oslc-op/config/v1.0/ps01/config-resources.html#StreamShape)
	
	Example:
	
	`POST creationFactoryUri`
	
	or
	B) Use the streams LDPC of a component
	
	1. Get the URI of the streams LDPC from the RDF of a component.
	2. POST the RDF content describing the stream (with a Content-Type header set to an RDF media type supported by the server) to the LDPC. The RDF content should be compatible with the resource shape for Stream as described in the specification (see https://docs.oasis-open-projects.org/oslc-op/config/v1.0/ps01/config-resources.html#StreamShape)
	
	Example:
	
	`POST streamsLdpcUri`

mergeIntoBranch	Not Available
diffCommits	Not Available
QueryService	
getQueryById	Not Available OSLC does not provide any RDF representation or persistence mechanisms of OSLC queries.
getQueries	Not Available OSLC does not provide any RDF representation or persistence mechanisms of OSLC queries.
executeQueryById	Not Available OSLC does not provide any RDF representation or persistence mechanisms of OSLC queries.
createQuery	Not Available OSLC does not provide any RDF representation or persistence mechanisms of OSLC queries.

PIM Services	OSLC PSM
updateQuery	Not Available OSLC does not provide any RDF representation or persistence mechanisms of OSLC queries.
deleteQuery	Not Available OSLC does not provide any RDF representation or persistence mechanisms of OSLC queries.

executeQuery	1. Discover a query capability for the specified project. See *OSLC Discovery* section.
	2. Perform a GET on the query base, specifying the following. See OSLC Query Capability in OSLC Nomenclature
	    1. The *oslc.where* query parameter to filter for the desired elements
	    2. The *oslc.select* query parameter, to define the element properties that should be returned in the query result.

ExternalRelationship Service	OSLC does/can not differentiate between properties/relationships/elements that are external or otherwise. The services below are just specific examples of the getElements and getElementById service operation mappings above.
getExternalRelationships ById	See getElementById under ElementNavigationService
getExternalRelationships ByElementEnd	See getRelationshipsByRelatedElement under ElementNavigationService
getExternalRelationships	See getElements under Element Navigation Service
ProjectUsageService	
createProjectUsage	Not available. The OSLC Configuration Management specification does not define any notion of component usage.
deleteProjectUsage	Not available. The OSLC Configuration Management specification does not define any notion of component usage.
getProjectUsages	Not available. The OSLC Configuration Management specification does not define any notion of component usage.

A Annex: Conformance Test Suite

(Normative)

A.1 Project Service Conformance Test Cases

Operation create_project

PIM-PS-001

Description	Create Project - success
Input	1. `_description : String[0..1]` 2. `_name : String[1]`
Scenario	
Precondition (OCL)	`let _record : Record = Record.allInstances()`
Steps	Execute operation `create_project(name = _name, description = _description) : Project[1]`

Result	1. Result, defined as `project : Project[1]`, is a created Project 2. The `id` attribute value of the created Project is randomly generated and universally unique, including (but not limited to) not being used as the `id` attribute value for any previously existing Record 3. A Branch is created and specified as the `defaultBranch` attribute value for the created Project 4. The `id` attribute value of the created Branch (`defaultBranch`) is randomly generated and universally unique, including (but not limited to) not being used as the `id` attribute value for any previously existing Record. 5. The `name` attribute value of the created Branch (`defaultBranch`) is specified as `'main'` 6. The `description` attribute value is specified as inputted 7. The `name` attribute value is specified as inputted

Postcondition (OCL)	`let project : Project = create_project(_name, _descripti`
	`project->size() = 1`
	`Project.allInstances()->includes(project)`
	`_record.id->excludes(project.id)`
	`project.defaultBranch->notEmpty()`
	`_record.id->excludes(project.defaultBranch.id)`
	`project.defaultBranch.name = 'main'`
	`project.description = _description`
	`project.name = _name`

Operation get_projects

PIM-PS-002

Description	Get Projects - success
Input	None
Scenario	
Precondition (OCL)	
Steps	Execute operation `get_projects() : Project[0..*]`
Result	Result, defined as `project : Project[0..*]`, is all of the existing Projects
Postcondition (OCL)	`let project : Project = get_projects()` `project = Project.allInstances()`

Operation get_project_by_id

PIM-PS-003

Description	Get Project by ID - exists
Input	1. `_projectId : UUID[1]`
Scenario	A Project with an `id` attribute value equal to `_projectId` exists
Precondition (OCL)	`let _project : Project = Project.allInstances()->select( _project->size() = 1`
Steps	Execute operation `get_project_by_id(projectId = _projectId) : Project[0..1]`
Result	Result, defined as `project : Project[1]`, is the Project with an `id` attribute value equal to `_projectId`
Postcondition (OCL)	`let project : Project = get_project_by_id(_projectId) project = _project`

PIM-PS-004

Description	Get Project by ID - does not exist
Input	1. `_projectId : UUID[1]`
Scenario	A Project with an `id` attribute value equal to `_projectId` does not exist
Precondition (OCL)	`Project.allInstances()->select(projectId = _projectId)->`
Steps	Execute operation `get_project_by_id(projectId = _projectId) : Project[0..1]`
Result	1. Result, defined as `project : Project[0]`, does not include any Projects 2. Result communicates that a Project with the provided ID does not exist
Postcondition (OCL)	`let project : Project = get_project_by_id(_projectId) project->isEmpty()`

A.2 Element Navigation Service Conformance Test Cases

Operation get_elements

PIM-EN-001

Description	Get Elements - success
Input	1. `_project : Project[1]` 2. `_commit : Commit[1]`
Scenario	1. The inputted Project exists 2. The inputted Commit exists 3. The inputted Commit belongs to the inputted Project

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`Commit.allInstance()->includes(_commit)`
	`_commit.owningProject = _project`

Steps	Execute operation `get_elements(project = _project, commit = _commit) : Element[0..*]`
Result	Result, defined as `element : Element[0..*]`, is all Elements at the inputted Commit
Postcondition (OCL)	`let element : Element = get_elements(_project, _commit)` `element = _commit.version.data->select(oclIsKindOf(Eleme`

Operation get_element_by_id

PIM-EN-002

Description	Get Element by ID - success

Input	1. `_project : Project[1]` 2. `_commit : Commit[1]` 3. `_elementId : UUID[1]`
Scenario	1. The inputted Project exists 2. The inputted Commit exists 3. The inputted Commit belongs to the inputted Project 4. An Element with an `id` attribute value equal to `_elementId` exists at the inputted Commit

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`Commit.allInstance()->includes(_commit)`
	`_commit.owningProject = _project`
	`let _element : Element = _commit.version->select(identit`
	`_element.size() = 1`

Steps	Execute operation `get_element_by_id(project = _project, commit = _commit, elementId = _elementId) : Element[0..1]`
Result	Result, defined as `element : Element[1]`, is the Element with an `id` attribute value equal to `_elementId` at the inputted Commit
Postcondition (OCL)	`let element : Element = get_element_by_id(_project, _com` `element = _element`

PIM-EN-003

Description	Get Element by ID - does not exist at Commit

Input	1. `_project : Project[1]` 2. `_commit : Commit[1]` 3. `_elementId : UUID[1]`
Scenario	1. The inputted Project exists 2. The inputted Commit exists 3. The inputted Commit belongs to the inputted Project 4. An Element with an `id` attribute value equal to `_elementId` does not exist at the inputted Commit

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`Commit.allInstance()->includes(_commit)`
	`_commit.owningProject = _project`
	`_commit.version->select(identity.id = _elementId and dat`

Steps	Execute operation `get_element_by_id(project = _project, commit = _commit, elementId = _elementId) : Element[0..1]`
Result	1. Result, defined as `element : Element[0]`, does not include any Elements 2. Result communicates that an Element with the provided ID at the inputted Commit does not exist
Postcondition (OCL)	`let element : Element = get_element_by_id(_project, _com` `element->isEmpty()`

Operation get_relationships_by_source

PIM-EN-004

Description	Get Relationships by source (Element) - success

Input	1. `_project : Project[1]` 2. `_commit : Commit[1]` 3. `_elementId : UUID[1]`
Scenario	1. The inputted Project exists 2. The inputted Commit exists 3. The inputted Commit belongs to the inputted Project 4. An Element with an `id` attribute value equal to `_elementId` exists at the inputted Commit

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`Commit.allInstance()->includes(_commit)`
	`_commit.owningProject = _project`
	`let _element : Element = _commit.version->select(identit`
	`_element.size() = 1`

Steps	Execute operation `get_relationships_by_source(project = _project, commit = _commit, elementId = _elementId) : Relationship[0..*]`
Result	Result, defined as `relationship : Relationship[0..*]`, is all the Relationships whose `source` attribute value includes the Element with `id` attribute value equal to `_elementId`
Postcondition (OCL)	`let relationship : Relationship = get_relationship_by_so` `relationship = _commit.version->select(data.oclIsKindOf(`

Operation get_relationships_by_target

PIM-EN-005

Description	Get Relationships by target (Element) - success

Input	1. `_project : Project[1]` 2. `_commit : Commit[1]` 3. `_elementId : UUID[1]`
Scenario	1. The inputted Project exists 2. The inputted Commit exists 3. The inputted Commit belongs to the inputted Project 4. An Element with an `id` attribute value equal to `_elementId` exists at the inputted Commit

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`Commit.allInstance()->includes(_commit)`
	`_commit.owningProject = _project`
	`let _element : Element = _commit.version->select(identit`
	`_element.size() = 1`

Steps	Execute operation `get_relationships_by_target(project = _project, commit = _commit, elementId = _elementId) : Relationship[0..*]`
Result	Result, defined as `relationship : Relationship[0..*]`, is all the Relationships whose `target` attribute value includes the Element with `id` attribute value equal to `_elementId`
Postcondition (OCL)	`let relationship : Relationship = get_relationship_by_ta` `relationship = _commit.version->select(data.oclIsKindOf(`

A.3 Project and Data Versioning Service Conformance Test Cases

Operation create_branch

PIM-PCB-001

Description	Create Branch - success

Input	1. `_project : Project[1]` 2. `_head : Commit[1]` 3. `_name : String[1]`
Scenario	1. The inputted Project exists 2. The inputted Commit (`_head`) exists 3. The inputted Commit (`_head`) belongs to the inputted Project

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`Commit.allInstance()->includes(_head)`
	`_head.owningProject = _project`
	`let _record : Record = Record.allInstances()`

Steps	Execute operation `create_branch(project = _project, head = _head, name = _name) : Branch[1]`
Result	1. Result, defined as `branch : Branch[1]`, is a created Branch in the inputted Project 2. The `id` attribute value of the created Branch is randomly generated and universally unique, including (but not limited to) not being used as the `id` attribute value for any previously existing Record. 3. The `owningProject` attribute value of the created Branch is specified as inputted, i.e. equal to `_project` 4. The `head` attribute value of the created Branch is specified as inputted 5. The `name` attribute value of the created Branch is specified as inputted

Postcondition (OCL)	`let branch : Branch = create_branch(_project, _head, _na`
	`branch->size() = 1`
	`Branch.allInstances()->includes(branch)`
	`_project.branch->includes(branch)`
	`_record.id->excludes(branch.id)`
	`branch.owningProject = _project`
	`branch.head = _head`
	`branch.name = _name`

Operation get_branches

PIM-PCB-002

Description	Get Branches - success

Input	1. `_project : Project[1]`
Scenario	The inputted Project exists
Precondition (OCL)	`Project.allInstances()->includes(_project)`
Steps	Execute operation `get_branches(project = _project) : Branch[0..*]`
Result	Result, defined as `branch : Branch[0..*]`, is all Branches in the inputted Project
Postcondition (OCL)	`let branch : Branch = get_branches(_project)` `branch = _project.branch`

Operation get_branch_by_id

PIM-PCB-003

Description	Get Branch by ID - success
Input	1. `_project : Project[1]` 2. `_branchId : UUID[1]`
Scenario	1. The inputted Project exists 2. A Branch with an `id` attribute value equal to `_branchId` exists in the inputted Project

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`let _branch : Branch = _project.branch->select(id = _bra`
	`_branch.size() = 1`

Steps	Execute operation `get_branch_by_id(project = _project, branchId = _branchId) : Branch[0..1]`
Result	Result, defined as `branch : Branch[1]`, is the Branch with an `id` attribute value equal to `_branchId` in the inputted Project
Postcondition (OCL)	`let branch : Branch = get_branch_by_id(_project, _branch` `branch = _branch`

PIM-PCB-004

Description	Get Branch by ID - does not exist in Project

Input	1. `_project : Project[1]` 2. `_branchId : UUID[1]`
Scenario	1. The inputted Project exists 2. A Branch with an `id` attribute value equal to `_branchId` does not exist in the inputted Project
Precondition (OCL)	`Project.allInstances()->includes(_project)` `_project.branch->select(id = _branchId)->isEmpty()`
Steps	Execute operation `get_branch_by_id(project = _project, branchId = _branchId) : Branch[0..1]`
Result	1. Result, defined as `branch : Branch[0]`, does not include any Branches 2. Result communicates that a Branch with the provided ID in the inputted Project does not exist
Postcondition (OCL)	`let branch : Branch = get_branch_by_id(_project, _branch` `branch->isEmpty()`

Operation delete_branch

PIM-PCB-005

Description	Delete Branch - success
Input	1. `_project : Project[1]` 2. `_branchId : UUID[1]`
Scenario	1. The inputted Project exists 2. A Branch with an `id` attribute value equal to `_branchId` exists in the inputted Project

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`let _branch : Branch = _project.branch->select(id = _bra`
	`_branch.size() = 1`

Steps	Execute operation `delete_branch(project = _project, branchId = _branchId) : Branch[0..1]`

Result	1. Result, defined as `branch : Branch[1]`, is the Branch with an `id` attribute value equal to `_branchId` in the inputted Project 2. Branch with an `id` attribute value equal to `_branchId` does not exist

Postcondition (OCL)	`let branch : Branch = delete_branch(_project, _branchId)`
	`_project.branch->excludes(branch)`
	`Branch.allInstances()->excludes(branch)`
	`_project.branch->select(id = _branchId)->isEmpty()`
	`Branch.allInstances()->select(id = _branchId)->isEmpty()`

PIM-PCB-006

Description	Delete Branch - does not exist in Project
Input	1. `_project : Project[1]` 2. `_branchId : UUID[1]`
Scenario	1. The inputted Project exists 2. A Branch with an `id` attribute value equal to `_branchId` does not exist in the inputted Project
Precondition (OCL)	`Project.allInstances()->includes(_project)` `_project.branch->select(id = _branchId)->isEmpty()`
Steps	Execute operation `delete_branch(project = _project, branchId = _branchId) : Branch[0..1]`
Result	1. Result, defined as `branch : Branch[0]`, does not include any Branches 2. Result communicates that a Branch with the provided ID in the inputted Project does not exist
Postcondition (OCL)	`let branch : Branch = delete_branch(_project, _branchId)` `branch->isEmpty()`

Operation get_default_branch

PIM-PCB-007

Description	Get default Branch - success

Input	1. `_project : Project[1]`
Scenario	The inputted Project exists
Precondition (OCL)	`Project.allInstances()->includes(_project)`
Steps	Execute operation `get_default_branch(project = _project) : Branch[1]`
Result	1. Result, defined as `branch : Branch[1]`, is the `defaultBranch` attribute value of the inputted Project
Postcondition (OCL)	`let branch = get_default_branch(_project) branch = _project.defaultBranch`

Operation set_default_branch

PIM-PCB-008

Description	Set default Branch - success
Input	1. `_project : Project[1]` 2. `_branchId : UUID[1]`
Scenario	1. The inputted Project exists 2. A Branch with an `id` attribute value equal to `_branchId` exists in the inputted Project
Precondition (OCL)	`Project.allInstances()->includes(_project) let _branch : Branch = _project.branch->select(id = _bra _branch.size() = 1`
Steps	Execute operation `set_default_branch(project = _project, branchId = _branchId) : Project[1]`

Result	1. Result, defined as `project : Project[1]`, is the inputted Project 2. The `defaultBranch` attribute value of the inputted Project is specified as the Branch with an `id` attribute value equal to `_branchId`
Postcondition (OCL)	`let project : Project = set_default_branch(_project, _br` `project.defaultBranch = _branch`

PIM-PCB-009

Description	Set default Branch - does not exist in Project
Input	1. `_project : Project[1]` 2. `_branchId : UUID[1]`
Scenario	1. The inputted Project exists 2. A Branch with an `id` attribute value equal to `_branchId` does not exist in the inputted Project

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`_project.branch->select(id = _branchId)->isEmpty()`
	`let _defaultBranch : Branch = _project.defaultBranch`

Steps	Execute operation `set_default_branch(project = _project, branchId = _branchId) : Project[1]`
Result	1. Result, defined as `project : Project[1]`, is the inputted Project 2. The `defaultBranch` attribute value of the inputted Project is `unchanged` 3. Result communicates that a Branch with the provided ID in the inputted Project does not exist
Postcondition (OCL)	`let project : Project = set_default_branch(_project, _br` `project.defaultBranch = _defaultBranch`

Operation create_commit

PIM-PCB-010

Description	Create Commit - success
Input	1. `_project : Project[1]` 2. `_change : DataVersion[1..*]` 3. `_branch : Branch[0..1]` 4. `_previousCommit : Commit[0..*]`
Scenario	1. The inputted Project exists 2. The inputted Branch, optional and if provided, exists in the inputted Project 3. The inputted Commits (`_previousCommit`), optional and if provided, exists in the inputted Commit

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`_branch->isEmpty() or _project.branch->includes(_branch)`
	`_previousCommit->isEmpty() or _previousCommit->forAll(ow`
	`let _record : Record = Record.allInstances()`
	`let _head : Commit = _branch.head`

Steps	Execute operation `create_commit(project = _project, change = _change, branch = _branch, previousCommit = _previousCommit) : Commit[1]`

Result	1. Result, defined as `commit : Commit[1]`, is a created Commit 2. The `id` attribute value of the created Commit is randomly generated and universally unique, including (but not limited to) not being used as the `id` attribute value for any previously existing Record. 3. The `owningProject` attribute value of the created Commit is specified as inputted (`_project`) 4. The `change` attribute value of the created Commit is specified as inputted 5. The `previousCommit` attribute value of the created Commit is specified as the union of the head attribute value of the inputted Branch, if provided, and the inputted Commits (`_previousCommit`) 6. The `version` attribute value of the created Commit includes the new changes 7. The `head` attribute value of the inputted Branch, if provided, is updated to the created Commit

Postcondition (OCL)	`let commit : Commit = create_commit(_project, _change, _`
	`commit->size() = 1`
	`_record.id->excludes(commit.id)`
	`commit.owningProject = _project`
	`commit.change = _change`
	`commit.previousCommit = _head->union(_previousCommit)`
	`commit.version->includes(_change)`
	`_branch->isEmpty() or _branch.head = commit`

Operation get_commit_by_id

PIM-PCB-011

Description	Get commit by ID - success
Input	1. `_project : Project[1]` 2. `_commitId : UUID[1]`
Scenario	1. The inputted Project exists 2. A Commit with an `id` attribute value equal to `_commitId` exists in the inputted Project

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`let _commit : Commit = _project.commit->select(id = _com`
	`_commit.size() = 1`

Steps	Execute operation `get_commit_by_id(project = _project, commitId = _commitId) : Commit[0..1]`
Result	Result, defined as `commit : Commit[1]`, is the Commit with an `id` attribute value equal to `_commitId` in the inputted Project
Postcondition (OCL)	`let commit : Commit = get_commit_by_id(_project, _commit commit = _commit`

PIM-PCB-012

Description	Get commit by ID - does not exist in Project
Input	1. `_project : Project[1]` 2. `_commitId : UUID[1]`
Scenario	1. The inputted Project exists 2. A Commit with an `id` attribute value equal to `_commitId` does not exist in the inputted Project
Precondition (OCL)	`Project.allInstances()->includes(_project) _project.commit->select(id = _commitId)->isEmpty()`
Steps	Execute operation `get_commit_by_id(project = _project, commitId = _commitId) : Commit[0..1]`
Result	Result, defined as `commit : Commit[0]`, does not include any Commits
Postcondition (OCL)	`let commit : Commit = get_commit_by_id(_project, _commit commit->isEmpty()`

Operation get_head

PIM-PCB-013

Description	Get head Commit of Branch - success

Input	1. `_project : Project[1]` 2. `_branch : Branch[1]`
Scenario	1. The inputted Project exists 2. The inputted Branch exists 3. The inputted Branch belongs to the inputted Project

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`Branch.allInstance()->includes(_branch)`
	`_branch.owningProject = _project`

Steps	Execute operation `get_head(project = _project, branch = _branch) : Commit[0..1]`
Result	Result, defined as `head : Commit[0..1]`, is the Commit that is the `head` attribute value of the inputted Branch

Postcondition (OCL)	`let head : Commit = get_head(_project, _branch)`
	`Bag{0, 1}->includes(head->size())`
	`head = _project.head`

PIM-PCB-014

Description	Get head Commit of Branch - Branch does not exist
Input	1. `_project : Project[1]` 2. `_branch : Branch[1]`
Scenario	1. The inputted Project exists 2. The inputted Branch does not exist
Precondition (OCL)	`Project.allInstances()->includes(_project)` `Branch.allInstance()->excludes(_branch)`
Steps	Execute operation `get_head(project = _project, branch = _branch) : Commit[0..1]`

Result	1. Result, defined as `head : Commit[0]`, does not include any Commits 2. Result communicates that the inputted Branch does not exist
Postcondition (OCL)	`let head : Commit = get_head(_project, _branch)` `head->isEmpty()`

PIM-PCB-015

Description	Get head Commit of Branch - Branch not in Project
Input	1. `_project : Project[1]` 2. `_branch : Branch[1]`
Scenario	1. The inputted Project exists 2. The inputted Branch exists 3. The inputted Branch does not belong to the inputted Project

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`Branch.allInstance()->includes(_branch)`
	`_branch.owningProject <> _project`

Steps	Execute operation `get_head(project = _project, branch = _branch) : Commit[0..1]`
Result	1. Result, defined as `head : Commit[0]`, does not include any Commits 2. Result communicates that the provided input is invalid
Postcondition (OCL)	`let head : Commit = get_head(_project, _branch)` `head->isEmpty()`

A.4 Query Service Conformance Test Cases

Operation create_query

PIM-QS-001

Description	Create Query - success

Input	1. `_project : Project[1]` 2. `_select : String[0..*]` 3. `_scope : DataIdentity[0..*]` 4. `_where : Constraint[0..1]` 5. `_orderBy : String[0..*]`
Scenario	1. The inputted Project exists
Precondition (OCL)	`Project.allInstances()->includes(_project)` `let _record : Record = Record.allInstances()`
Steps	Execute operation `create_query(project = _project, select = _select, scope = _scope, where = _where, orderBy = _orderBy) : Query[1]`
Result	1. Result, defined as `query : Query[1]`, is a created Query in the inputted Project 2. The `id` attribute value of the created Query is randomly generated and universally unique, including (but not limited to) not being used as the `id` attribute value for any previously existing Record. 3. The `owningProject` attribute value of the created Query is specified as inputted, i.e. equal to `_project` 4. The `select` attribute value of the created Query is specified as inputted 5. The `scope` attribute value of the created Query is specified as inputted 6. The `where` attribute value of the created Query is specified as inputted 7. The `orderBy` attribute value of the created Query is specified as inputted

Postcondition (OCL)	`let query : Query = create_query(_project, _select, _sco`
	`query->size() = 1`
	`Query.allInstances()->includes(query)`
	`_project.query->includes(query)`
	`_record.id->excludes(query.id)`
	`query.owningProject = _project`
	`query.select = _select`
	`query.scope = _scope`
	`query.where = _where`
	`query.orderBy = orderBy`

PIM-QS-002

Description	Execute Query - success
Input	1. `_query : Query[1]` 2. `_commit : Commit[1]`
Scenario	1. The inputted Query exists 2. The inputted Commit exists 3. The inputted Query and Commit both belong to the same Project

Precondition (OCL)	`Query.allInstances()->includes(_query)`
	`Commit.allInstances()->includes(_commit)`
	`_query.owningProject = _commit.owningProject`

Steps	Execute operation `execute_query(query = _query, commit = _commit) : Data[0..*]`
Result	1. Result, defined as `data : Data[0..*]`, is all of the Datas at the inputted Commit that satisfy the conditions of the inputted Query 2. If inputted Query has a non-empty `scope` attribute value, result only includes Data within the Query's scope

Postcondition (OCL)	`let data : Data = execute_query(_query, _commit)`
	`_commit.versionedData->includesAll(data)`
	`_query.scope->isEmpty() or _query.scope->includesAll(dat`

A.5 External Relationship Service Conformance Test Cases

A.6 Project Usage Service Conformance Test Cases

A.7 Cross-Cutting Conformance Test Cases

Operations with Invalid Input

PIM-CC-001

Description	Execute operation - missing Project input
Input	1. `_project : Project[0]`
Scenario	

Precondition (OCL)	
Steps	Execute any of the following operations, defined as `x(project : Project[1], …)`: • `get_elements(project = _project, …)` • `get_element_by_id(project = _project, …)` • `get_relationships_by_source(project = _project, …)` • `get_relationships_by_target(project = _project, …)` • `create_branch(project = _project, …)` • `get_branches(project = _project)` • `get_branch_by_id(project = _project, …)` • `delete_branch(project = _project, …)` • `get_default_branch(project = _project)` • `set_default_branch(project = _project, …)` • `create_commit(project = _project, …)` • `get_commit_by_id(project = _project, …)` • `get_head(project = _project, …)` • `create_query(project = _project, …)`
Result	1. Result, defined as `result : OclAny[0]`, does not include any OclAny 2. Result communicates that `project` is a required input
Postcondition (OCL)	`let result : OclAny = x(_project, …)` `result->isEmpty()`

PIM-CC-002

Description	Execute operation - missing Commit input
Input	1. `_commit : Commit[0]`
Scenario	
Precondition (OCL)	

Steps	Execute any of the following operations, defined as `x(commit : Commit[1], …)`: • `get_elements(…, commit = _commit)` • `get_element_by_id(…, commit = _commit, …)` • `get_relationships_by_source(…, commit = _commit, …)` • `get_relationships_by_target(…, commit = _commit, …)` • `create_branch(…, head = _commit, …)`
Result	1. Result, defined as `result : OclAny[0]`, does not include any OclAny 2. Result communicates that `commit` is a required input
Postcondition (OCL)	`let result : OclAny = x(_commit, …)` `result->isEmpty()`

PIM-CC-003

Description	Execute operation - Project input does not exist
Input	1. `_project : Project[1]`
Scenario	The inputted Project does not exist
Precondition (OCL)	`Project.allInstances()->excludes(_project)`

Steps	Execute any of the following operations, defined as `x(project : Project[1], …)`: • `get_elements(project = _project, …)` • `get_element_by_id(project = _project, …)` • `get_relationships_by_source(project = _project, …)` • `get_relationships_by_target(project = _project, …)` • `create_branch(project = _project, …)` • `get_branches(project = _project)` • `get_branch_by_id(project = _project, …)` • `delete_branch(project = _project, …)` • `get_default_branch(project = _project)` • `set_default_branch(project = _project, …)` • `create_commit(project = _project, …)` • `get_commit_by_id(project = _project, …)` • `get_head(project = _project, …)` • `create_query(project = _project, …)`
Result	1. Result, defined as `result : OclAny[0]`, does not include any OclAny 2. Result communicates that the inputted Project does not exist
Postcondition (OCL)	`let result : OclAny = x(_project, …) result->isEmpty()`

PIM-CC-004

Description	Execute operation - Commit input does not exist
Input	1. `_commit : Commit[1]`
Scenario	The inputted Commit does not exist
Precondition (OCL)	`Commit.allInstances()->excludes(_commit)`

Steps	Execute any of the following operations, defined as `x(commit : Commit[1], …)`: • `get_elements(…, commit = _commit)` • `get_element_by_id(…, commit = _commit, …)` • `get_relationships_by_source(…, commit = _commit, …)` • `get_relationships_by_target(…, commit = _commit, …)` • `create_branch(…, head = _commit, …)`
Result	1. Result, defined as `result : OclAny[0]`, does not include any OclAny 2. Result communicates that the inputted Commit does not exist
Postcondition (OCL)	`let result : OclAny = x(_commit, …)` `result->isEmpty()`

PIM-CC-005

Description	Execute operation - Commit input is not owned by Project input
Input	1. `_project : Project[1]` 2. `_commit : Commit[1]`
Scenario	1. The inputted Project exists 2. The inputted Commit does not exist 3. The inputted Commit does not belong to the inputted Project

Precondition (OCL)	`Project.allInstances()->includes(_project)`
	`Commit.allInstances()->includes(_commit)`
	`_commit.owningProject <> _project`

Steps	Execute any of the following operations, defined as `x(project : Project[1], commit : Commit[1], …)`: • `get_elements(project = _project, commit = _commit)` • `get_element_by_id(project = _project, commit = _commit, …)` • `get_relationships_by_source(project = _project, commit = _commit, …)` • `get_relationships_by_target(project = _project, commit = _commit, …)` • `create_branch(project = _project, head = _commit, …)`
Result	1. Result, defined as `result : OclAny[0]`, does not include any OclAny 2. Result communicates that the provided input is invalid
Postcondition (OCL)	`let result : OclAny = x(_project, _commit, …) result->isEmpty()`

PIM-CC-006

Description	Execute operation - missing name input
Input	1. `_name : String[0]`
Scenario	
Precondition (OCL)	
Steps	Execute any of the following operations, defined as `x(name : String[1], …) : OclAny`: • `create_project(name = _name, …)` • `create_branch(…, name = _name, …)`
Result	1. Result, defined as `result : OclAny[0]`, does not include any OclAny 2. Result communicates that `name` is a required input
Postcondition (OCL)	`let result : OclAny = x(_name, …) result->isEmpty()`

PIM-CC-007

Description	Execute operation - missing UUID input
Input	1. `_id : UUID[0]`
Scenario	
Precondition (OCL)	
Steps	Execute any of the following operations, defined as `x(id : UUID[1], …) : OclAny`: • `get_project_by_id(projectId = _id)` • `get_element_by_id(…, elementId = _id)` • `get_relationships_by_source(…, elementId = _id)` • `get_relationships_by_target(…, elementId = _id)` • `get_branch_by_id(…, branchId = _id)` • `delete_branch(…, branchId = _id)` • `set_default_branch(…, branchId = _id)` • `get_commit_by_id(…, commitId = _id)`
Result	1. Result, defined as `result : OclAny[0]`, does not include any OclAny 2. Result communicates that the input for which `_id` is used is a required input
Postcondition (OCL)	`let result : OclAny = x(_id, …)` `result->isEmpty()`

B Annex: API and Services Examples

(Informative)