MXLIMS model overview

version 0.2.0

This version makes some major changes relative to previous versions. We now have precise model definitions in Pydantic, JSON schemas, and html documentation for parts of the model, which should be enough to start trying to use it.

Introduction

The MXLIMS aims to make a precise, defined data model that allows you to store and transfer relevant metadata to go with the main data produced. The initial scope is for macromolecular crystallography, but the approach is general and can be expanded as far as someone is willing to take it. The approach is that of MongoDB or metadata catalogues like ICAT and SciCat; a minimum of linked core object, which contain metadata to accommodate the infinite variety of specific data one might want to store. The metadata are modelled separately, so that all data are precisely defined; the lack of inter-object links and the versioning of the metadata schemas facilitates making local changes without unpredictable consequences.

The modelling is based on work, discussions and use cases from various people in the world of synchrotron crystallography over a number of years. Of particular note is ICAT (the Job class is a core ICAT class), mmCIF, Ed Daniels for modeling of samples and shipping, Global Phasing and Olof Svensson for handling workflows and multi-sweep experiments, and Kate Smith and May Sharpe for discussions on SSX use cases.

MXLIMS model

The core model is built to support full provenance tracking through a series of experiments and calculations. It consists of four types of objects:

- Datasets – the data we want to look at
- Jobs – the experiments or calculations that produce the datasets
- PreparedSamples – the materials being investigated, their contents and provenance
- LogisticalSamples – the nested sample holders and locations that are submitted to experiments, from Dewars through plates and pucks to crystals and drop locations.

These four seemed the minimum number of basic classes appropriate to represent synchrotron crystallography and provenance tracking. Each has data that belong in this kind of class, and that cannot be represented elsewhere in a natural manner.

For a view of the model core, see Figures 1 and 2 below.

The MxlimsObject is the superclass for model objects. The uuid attribute gives an identifier to each object; the mxlims_type (and the version) determine which schema is used for the parameters (metadata) and what constraints there may be on inter-object links

LogisticalSamples are organised as nested containers, e.g. Shipments containing Plates. containing Drops, containing locations.

Datasets can have either a source (the Job that created them) of a derived_from link. The Dataset.role specifies the role that the Dataset ha relative to the job that created it; this allows yo to distinguish e.g. Characterisation sweeps from data sweeps in a multi-sweep workflow experiment. Information sufficient to identify the location of data files must be given in the type-specific metadata.

Database core model

The database implementation shown is shows a natural way that the MXLIMS model could be stored in a database or LIMS system. There is no actual database implementation at the moment and Global Phasing, for one , does not plan to make such a system, but this diagram is if nothing else a good way to show the inherenet structure of MXLIMS.

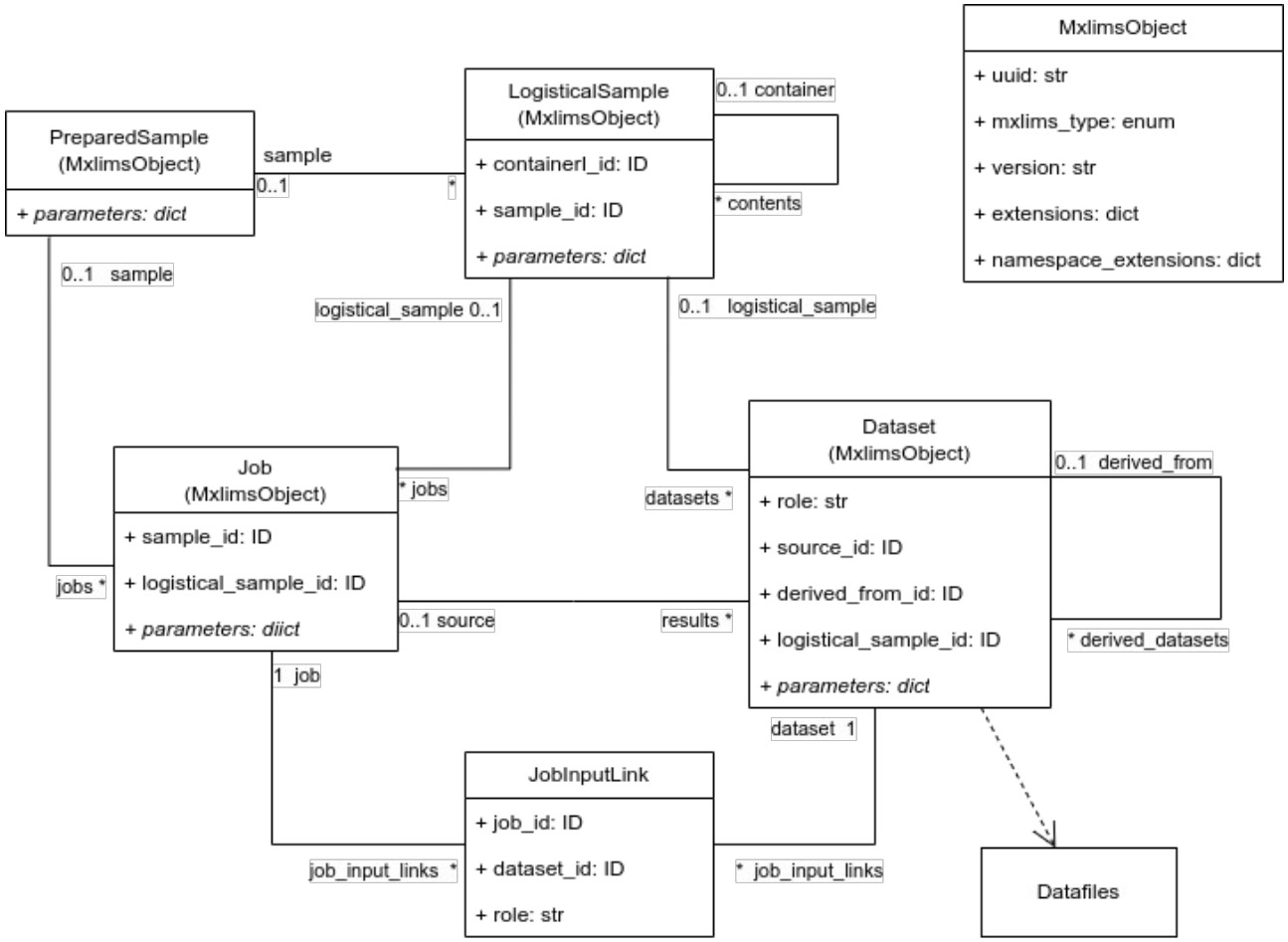

Figure 1 The core model, as organised for storing in a database. All links are mediated by the '..._id' attributes shown, which serve a foreign keys. Metadata are handled by the parameters dictionaries and by the many-to-many links implemented in the JobInput_link class.

The JobInput_link class implements a series of many-to-many links between Jobs and Datasets intended to support different types of input such as data, reference data, or templates. The Job input_link.role and the types of Jobs and Datasets allowed are defined as part of the metadata for different mxlims_types.

JSON schema core model

The first use of the MXLIMS model is to support structured messages through JSON schemas. Unlike a database implementation this does impose some constraints. MXLIMS data in principle forms a general object graph of unlimited size, whereas JSON files are tree structures and must be of reasonable length. Also there is no good way of handling duplicate appearances of a model object within a JSON file; duplicate copies cause problems with potential content differences, whereas intra-file links as not as well supported in JSON as in for instance YAML To keep things simple it was decided to implement most inter-object links as references, with only links starting on Job objects and the LogisticalSample.contents link being implemented as JSON content. This automatically limits the maximum extent of object graphs, and avoids the possibility of duplicate objects appearing within any single JSON file.

The JSON schema core modle is shown in Figure 2 (below)

As a result, the most extensive JSON object supported consists of a single Job, together with all inputs, results, and the directly linked PreparedSample and LogisticalSample. If you need to comine information from several Jobs, e.g. "*All jobs relating to a given sample*" or "an *MXExperiment together together with processing by several different programs*", you need to pass teh information as multiple JSON objects and integrate the information in the internal structure of the target program.

It is important to note that most attributes in the model are optional, and that the same mxlims_types are reused in as many contexts as possible. E.g. the MXCollectionSweep object can be used both as a template with default values or as part of a diffraction plan, as input to a beamline control system acquisition, to describe the result of an acquisition, and to describe input to a processing program. This reduces the need for specifying the same attributes in multiple, slightly different contexts.

Specific model

The JSON schema version of the model is implemented as Pydantic classes (https://docs.pydantic.dev/latest/). This has a number of advantages: Pydantic is more compact and easier to read and write than JSON schemas, but can be exported directly as high-quality JSON schemas as part of the Pydantic framework. Readers are invited to compare the Pydantic classes as defined in Pyhton code with the JSON schemas generated from them. Pydantic also provides an in-memory data structure for Python, with I/O as part of the framework. The generated JSON schemas are in mxlims/pydantic/jsonSchema, to distinguish them from the schemas in mxlims/jsonSchema that are written by hand and that are not (yet) merged with the other part of the model.

Html documentation for the JSON schemas has been generated automatically using the generate-schema-doc program (https://github.com/coveooss/json-schema-for-humans, https://pypi.org/project/json-schema-for-humans/). The result has beed added to mxlims/pydantic/jsonDocs.

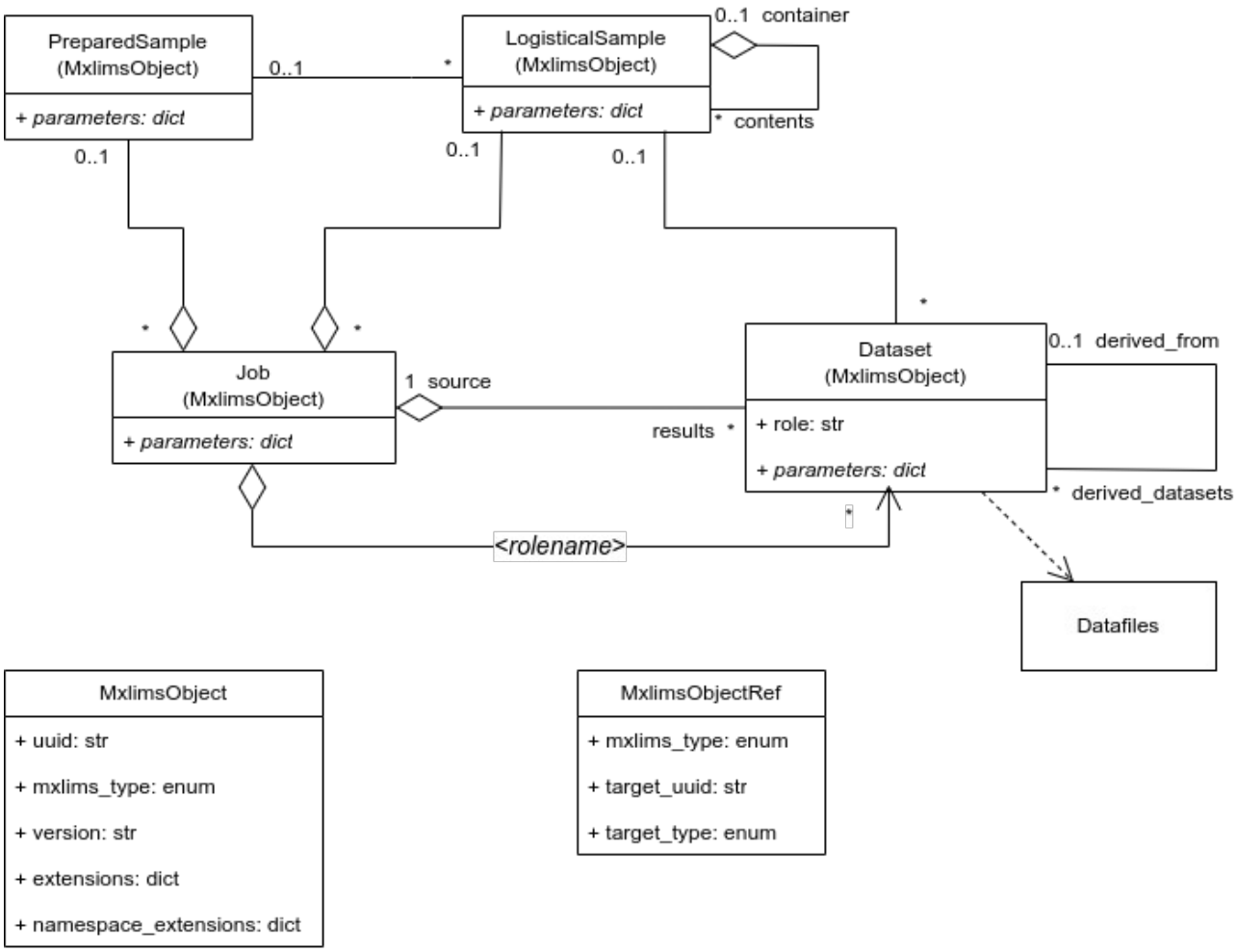

Figure 2 JSON implementation of the core MXLIMS model. Links with an empty diamond signify aggregation, i.e. all objects linked to from the Job class are contained within the Job JSON structure. Other links (in both directions) are implemented as MxlimsObjectRefs. Note that only links from Job objects and LogisticalSample.contents are implemented as actual JSON content. The link beteween Job and Dataset marked as <*rolename*> is special. This describes several one-way -to-many contained links with names given by <rolename>; these are the links implemented by the JobInput_link class in the database implementation.