MXLIMS model overview

Contents

Version 0.5.0

This version version represents a major reorganisation of the model structure (hence the double jump from version 0.3.0 to 0.5.0) while the model contents remain almost unchanged. It also includes proper (autogenerated) documentation. The result should be mature enough for actual use, and is no longer seen as mainly a starting point for discussion. Contributions, feedback and change proposals remain welcome and will be acted on, but it is expected that from now on MXLIMS will mainly change as the result of people providing new modelling or trying to implement the result.

Introduction

MXLIMS aims to make a precise, defined data model that allows you to store and transfer relevant metadata to go with the actual 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 support for site-specific data, the lack of inter-object links in the metadata and the versioning of the metadata schemas, allow you to make local changes without unpredictable consequences.

One requirement of the model is to support a LIMS system, including provenance tracking. Another is to define an API supporting information passing between programs and sites, from experiment planning and input parameters, through instructions for acquisition queues, metadata for results, and input to subsequent processing steps. A third requirement is maintainability. A fully precise model would require individual tables for each separate kind of data, but the experience of ISPyB shows that such models become complex, rigid, and hard to maintain, particularly when multiple sites need to make changes to support their specific needs. To mitigate this problem it is a core principle in MXLIMS to cut down the number of classes by using the same schemas for similar use cases, and for all stages of the experimental process from the initial diffraction plans to the final result. Additionally, MXLIMS has specific slots where you can store additional keyword-value parameters without misusing or breaking the main model.

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 (Icebear) and Karl Levik (Diamond) for modelling of samples and shipping, Global Phasing and Olof Svensson (ESRF) for handling workflows and multi-sweep experiments, and Kate Smith and May Sharpe (SLS) for discussions on SSX use cases. Most recently the model has been modified to reflect input from the MXCuBE AbstractLIMS working group.

MXLIMS model

The MXLIMS model is structured in three layers to support the various use cases.

- The *core* layer defines the (very few) abstract classes that underpin the entire model, as well as the inter-object links that are supported. This layer maps to the database tables needed for the model
- The *raw object* layer extends the core layer and contains JSON schemas for each individual object and data type, defining all the data fields (ICAT ‘metadata’) for each.
- The final *object* layer extends the raw objects by adding fields that support nesting JSON objects within other objects to form a tree. This layer also handles the constraints on interobject links that determine (e.g.) that it is pucks that can contain pins and not *vice versa*.

Core layer

The core layer is shown in figure 1.

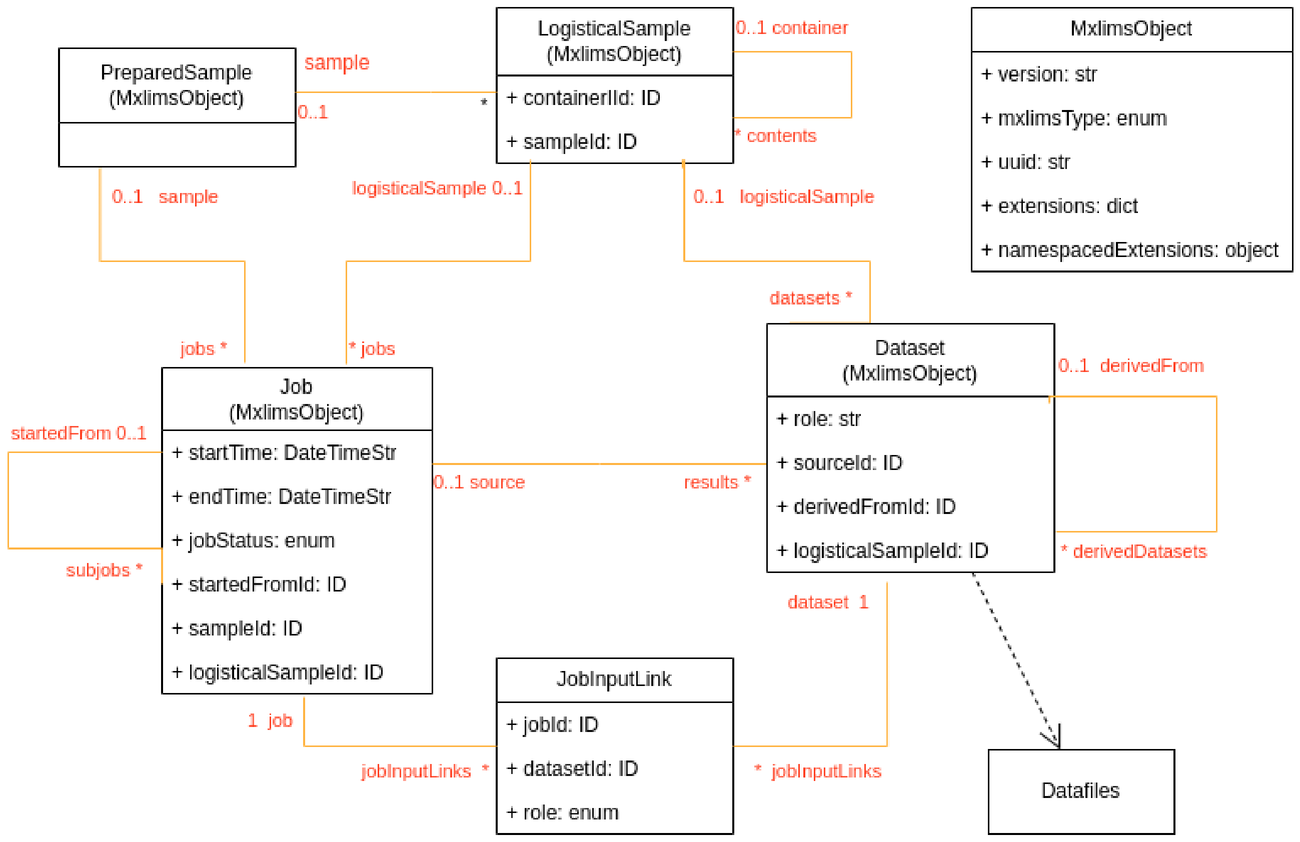

Figure 1 The core model, as organised for storing in a database. Inter-object links (in orange and red) are in principle two-way, and are mediated by the ...Id attributes that serve as foreign keys. The mxlimeType attribute distinguish between different subtypes, and metadata are defined separately for each subtype. The full model includes limitations on which types can be linked together, but these constraints are not reflected in the core model.

The JobInput_link class implements a series of many-to-many links between Jobs and Datasets intended to support different types of input data. The JobInputLink.role attribute, which specifies the type of link, currently has three supported values: “inputData”, “referenceData”, and “templateData”.

The four main classes are:

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

These four seemed the minimum number of basic classes appropriate to represent synchrotron crystallography and provenance tracking.

The **MxlimsObject** is the superclass for all model objects. The uuid attribute gives an identifier to each object; the version determines which schema versions are used for the parameters (metadata). The uuid is modelled as optional to allow for omitting it in JSON messages, but would be mandatory within a LIMS system. There are two slots for site-specific extensions, in order to allow flexible use without breaking the model. The 'extensions' attribute is an unconstrained keyword:value dictionary. The use of this attribute is discouraged, but allows for adding extra information quickly without breaking or abusing the model. The namespaced_extensions field is intended for extensions that are defined by published, site-specific schemas. Namespaces could be e.g. 'ESRF' or 'GPhL' and the relevant schemas are to be maintained by the owner of the namespace.

The **PreparedSample** class describes the material of the sample, including contents and components, creation date, identifiers and batch numbers. The same PreparedSample can be used in several different LogisticalSamples, at several different levels (well, drop, location, crystal).

LogisticalSamples are organised as nested containers, e.g. Shipments containing Plates. containing Drops, containing Wells. The lowest level of the hierarchy would be the Crystal. In practice a loop or drop location may contain multiple crystals that are not identified until during the experiment, so the Crystal object would often not be generated until needed during the experiment. Jobs (experiments) can be applied to several types of LogisticalSample provided they correspond to a single PreparedSample, e.g. to Pins, Wells, Drops, or Crystals.

Jobs describe an experiment or calculation that can generate Datasets. Jobs can have inputs such as template data (for diffraction plans), reference data (e.g. reference mtz files), or plain input data (for processing jobs), and produce Dataset results. Jobs can be nested, so that one job (e.g. a workflow run) starts other jobs (e.g. X-ray centring, characterisation, or acquisition).

Datasets can have either a source (the Job that created them) or a derivedFrom link (but not both). The Dataset.role specifies the role that the Dataset has relative to the job that created it; this allows you to distinguish different types of output. Information sufficient to identify the location of data files must be given in the type-specific metadata.

It should be noted that these four abstract classes delimit the kind of objects that can be modelled. There is for instance no object that can correspond to a sample component as such, be it Lysozyme, dithiothreitol, or fetal calf serum, so we cannot have a single object with a uuid that we can refer to for sample components. Sample components have to be specified as part of samples, with some resulting duplication in giving the various identifiers, names and Smiles strings together every time.

Objects layers

The Structure of the raw and final object layers are shown in figure 2.

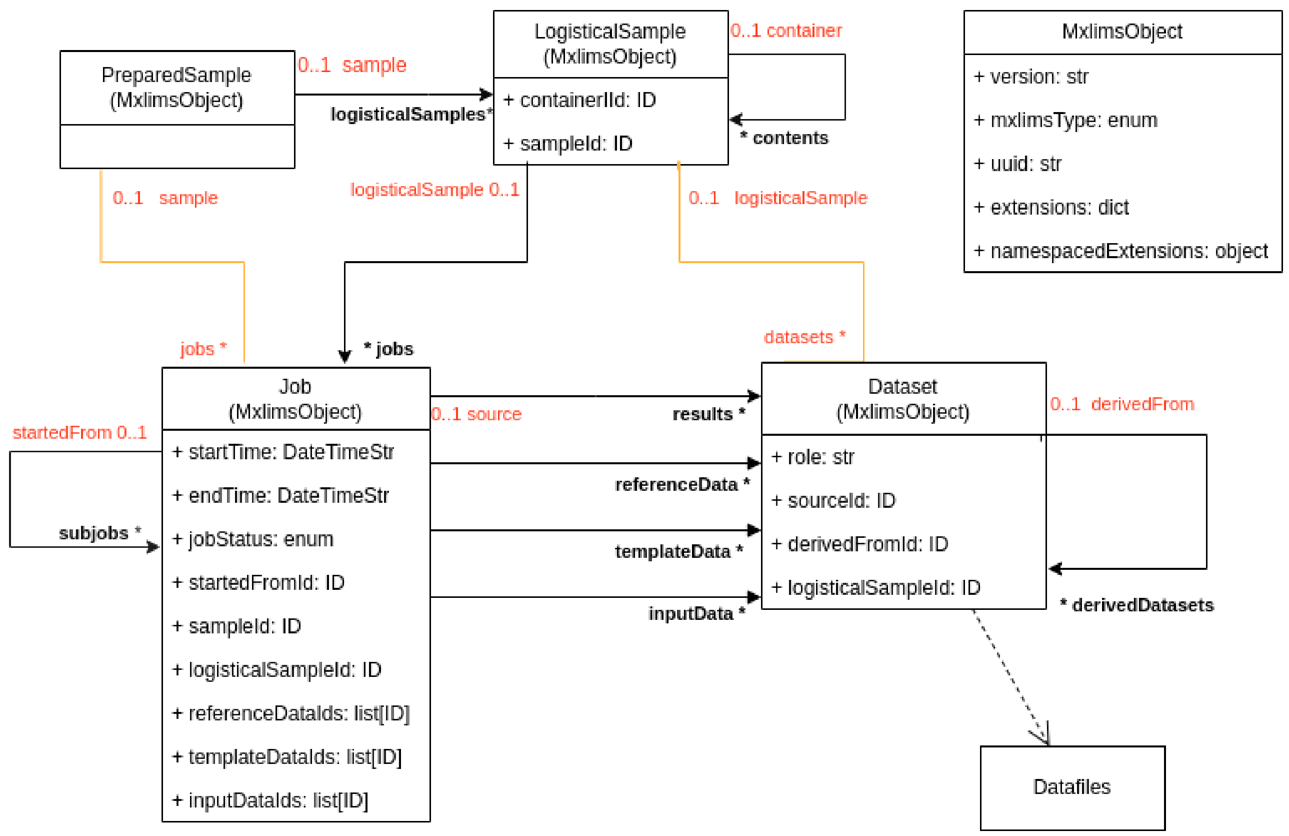

Figure 2 The base classes for the raw and final MXLIMS model layers. The (directional) links shown in black are modelled separately for each subtype and added to the 'raw' schemas to give the final object specification. In addition to specifying the link typing, these links define which JSON structures can be included as (hierarchical) substructures. Note that these containment links form a tree structure, as required for a JSON document.

The raw object layer contain the same Id fields and interobject links as the core layer. Individual objects are subclasses of the core classes, and include specification of the (meta)data fields (not shown in Fig. 2). The core JobInputLink class is not used outside of database implementations; instead the Job-Dataset many-to-many links are stored as lists of Dataset Ids in the Job.inputDataIds, Job.referenceDataIds and Job.templateDataIds fields.

The final object layer extends the raw object layer by adding fields for including JSON objects to form a tree structure. PreparedSamples can include nested LogisticalSamples, LogisticalSamples can include Jobs, and Jobs can include subjobs, results, and the various kinds of input Datasets. Each specific kind of object specifies which type of objects it can contain. Dewars, for instance, can contain only Pucks, Pins can contain only PinPositions, and MxExperiments cannot contain inputData and can produce only Collection Sweeps, etc. These containment links are the only way in the model for constraining the types of objects that can be linked.

The model contains the following objects:

PreparedSample: CrystallographicSample

Job: MxExperiment, MxProcessing

Dataset: CollectionSweep, ReflectionSet

LogisticalSample: DewarShipment, Dewar, Puck, Pin, PinPosition, Crystal, PlateShipment , Plate, PlateWell, WellDrop, DropRegion

Object links

The handling of links between objects is complicated by the multiple requirements of the model. In principle a data model that allows for provenance tracking must support an infinite net of objects, which would require a database implementation with a table for each object type in order to specify the model fully. This approach, of course, has been rejected as too rigid. A database implementation limited to the tables in the core schema can still support an infinite network of two-way links, but has no good way of restricting the types of linked objects, so these constraints must be handled ad-hoc. The same goes for a JSON implementation based on the (non-nested) raw objects.

The final objects do allow nesting, and the containment links can be typed for each kind of object. However, nested JSON objects are limited to tree structures, which cannot fully represent the model. One problem is that the same object (e.g. PreparedSample or Dataset) can appear multiple times in the tree, giving problems with duplication or non-tree links. Another is that some links, e.g. between Dataset and LogisticalSample have no place in the tree structure. In practice it is necessary to represent the model with a mixture of containment and foreign-key links in JSON documents, and leave it to the implementation of the consuming programs to sort out the links. This still leaves the problem that while a foreign key (e.g. a sampleId) in a JSON document points to a specific PreparedSample object, there is no way of guaranteeing that the relevant object is available, either in the same document or in the program information storage.

The mxlims/messages directory shows some examples of JSON schemas that can be used to transfer consistent groups of linked objects.

Tools and dependencies

The model specification is written in JsonSchema. We use version 07 as this is the highest version supported by the documentation generator.

Html documentation is generated using json-schema-for-humans (https://coveooss.github.io/json-schema-for-humans/#/). The command used is:

```
generate-schema-doc --link-to-reused-ref schemas docs/html
```

Conversion to Pydantic is done with datamodel_code_generator (https://docs.pydantic.dev/latest/integrations/datamodel_code_generator/). The command used is:

```
datamodel-codegen --input-file-type jsonschema --output-model-type
pydantic_v2.BaseModel --use-schema-description --use-double-quotes
--use-default --target-python-version 3.10 --snake-case-field --
capitalise-enum-members --use-title-as-name --input schemas --
output pydantic
```

The tools do impose some limitation on the modelling. Json-schema-for-humans seems to be able to handle the full JSON specification (up to version 07), but the Pydantic generation does not support the full range of JSON schemas. There are some problems for complex logical constraints, which may indeed make more sense in the context of document validation than in the context of Python data storage classes. To obtain clearer Pydantic, we have

- Modelled all enumerations as separate schemas rather than as part of field specifications.
- Modelled alternative variants as separate schemas and tried to avoid “oneOf” constraints, since these would anyway result in separate Pydantic classes. For instance we have separate PlateShipment and DewarShipment, rather than a single Shipment schema containing *either* plates *or* dewars.
- Where “oneOf” constraints have been retained the Pydantic model sometimes lack some of the model constraints, e.g. the constraint that a Dataset must contain *either* a sourceId *or* a derivedFromId.