MXLIMS model overview

version 0.3.0

This version, again, makes major changes relative to previous versions. The model source is now in JSON schema rather than Pydantic, the JSON structure handling of links between objects have changed, and there are some model changes. Furthermore, the Shipment model of Ed Daniel has been tentatively ported to the main framework, and teh model has been expanded to support mesh and line scans.

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, 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 modelling 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. Most recently the model has been modified to reflect input from the MXCuBE AbstractLIMS working group.

This is still a draft - there are some high-level decisions still open about the organisation of the model in JSON, tag names etc.

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 drop locations and crystals.

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 version determines which schema versions are used for the parameters (metadata).

The metadata are kept in the data attribute, which is a subtype of the MxlimsData class. It includes an mxlims_type string that identifies which kind of data we are talking about.

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 Locations. 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 generally not be generated until needed.

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 derived_from link. The Dataset.role specifies the role that the Dataset has relative to the job that created it; this allows you 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.

It should be noted that these four abstract classes delimit the kind of objects we can model. There is for instance no object that can correspond to a sample component, be it Lysozyme, dithiothreitol, of fetal calf serum, so we cannot have a single object with a uuid that we can refer to for sample components. Sam[ple 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.

The model has two slots for site-specific extensions, in order to allow flexible use without breaking the model. Each MxlimsObject has an ‘extensions’ attribute that is an unconstrained keyword:value dictionary. The use of such extensions is discouraged, but is better than breaking or abusing the model. Each Metadata class, (under MxlimsData) further has a namespaced_extensions field, where you can put site-specific 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.

In actual use the objects you are dealing with will be subtypes of the classes shown, as can be seen in the mxlims/schemas/objects directory. Each subclass has its own definition of allowed metadata, including its ‘mxlims_type’ that serves to identify it. In reality there will also be constraints on links between objects of various types; at it simplest you cannot ship Plates inside a Dewar. As the model is currently structured, these constraints can only be partially supported in JSON schemas, and would have to be handled outside the framework in a database. This would seem to be an unavoidable cost of working with a few, generic tables with added metadata, as opposed to the ISPyB way of having (and maintaining!) separate tables for every kind of object.

Database core model

The database implementation shown is 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 inherent structure of MXLIMS.

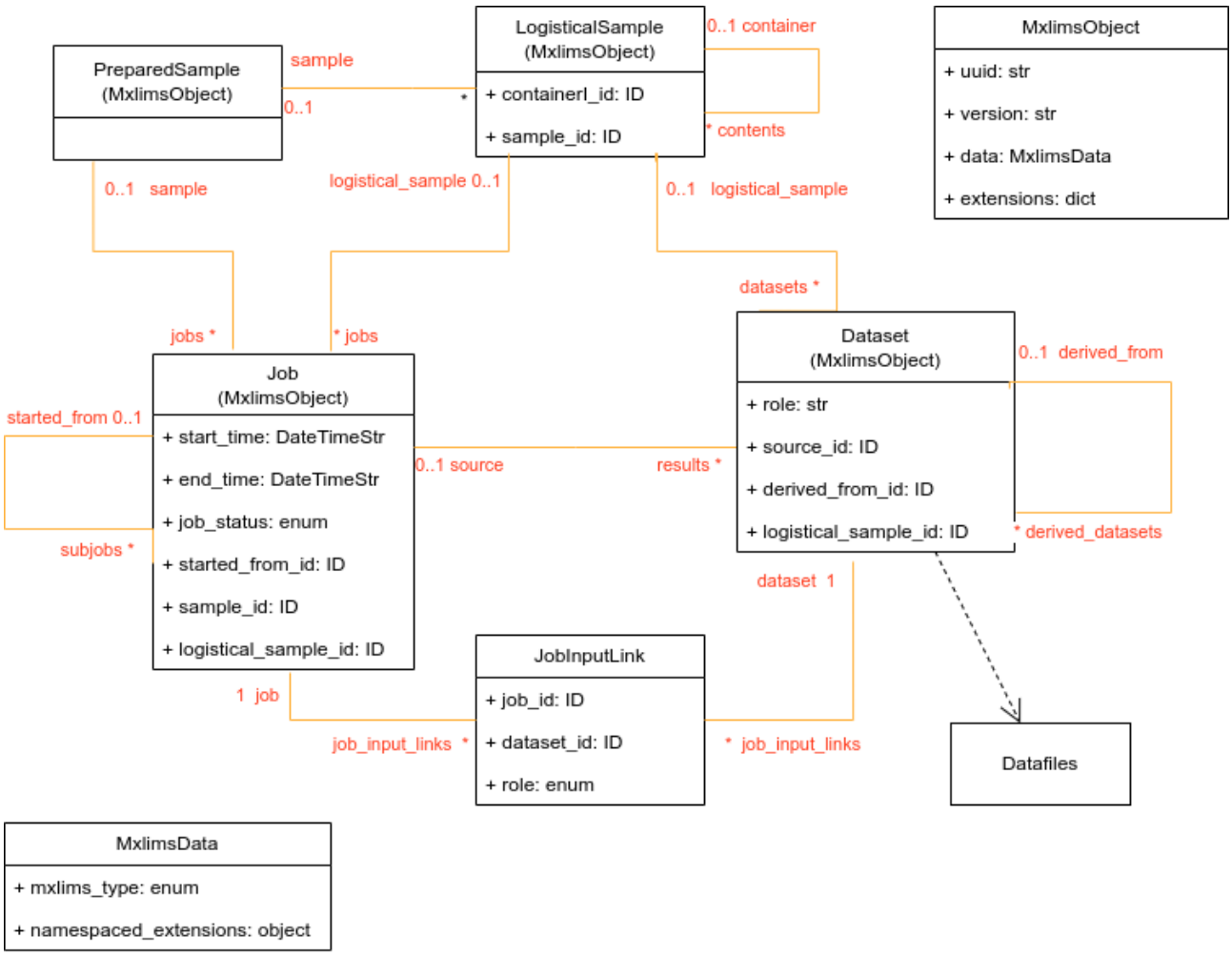

Figure 1 The core model, as organised for storing in a database. Inter-object links (in orange and red) are in principle two-way, and are mediated by the ..._id attributes that serve as foreign keys. Metadata are handled by the data' attribute, which are specified by theri own schemas.

The JobInput_link class implements a series of many-to-many links between Jobs and Datasets intended to support different types of input data, reference data, or templates. The input_link.role attribute, which specifies the type of link, currently has three supported values: "input", "reference", and "template".

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 well supported in JSON (unlike for instance YAML).

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

As a starting point the object ..._id that serve as foreign keys are also stored in the JSON files and allow you to reconstruct the object graph at will after reading the data. The MxlimsMessage object in the diagram is a schema that supports this – you simply give separate lists of the four basic types of object and let the foreign keys do the rest. This, however, gives no possibility of constraining the types of which objects are linked to which. As a partial remedy, the JSON schemas have slots for lists of contained objects for a number of teh inter-object links, specifically for LogisticalSample.contents, LogisticalSample.jobs, Job.subjobs, Job.results, and Job.template_data. These are strictly speaking redundant relative to the ..._id fields, but they allow for constraining the type of the objects on the other side of the link, and for having JSON formed as a single object with nested contents rather than as lists of independent objects that need to be linked up after the fact. Specifically these fields let you have a Shipment with nested containers, and jobs and template data to support diffraction plan (but not Samples) as a single JSON object, as well as a Job (e.g. experiment) complete with subjobs and results. Allowing this facility for only some of the (in principle) two-way links minimises the problem of having the same object appear in two different places in a JSON document. Allowing the same flexibility for all the two-way links would give better flexibility and tyope control, but would leave a major problem with duplicate appearances of objects.

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 CollectionSweep 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

From version 0.3.0 onwards the model is specified in JSON schemas, which can be found in mxlims/schemas/. They are divided into core/ ,objects/ and datatypes/. mxlime/pydantic contains the pydantic code generated from the JSON schemas using the https://docs.pydantic.dev/latest/integrations/datamodel_code_generator/.

mxlims/schemas/docs contains JSON documentation autogenerated using generate-schema-doc (https://pypi.org/project/json-schema-for-humans/); as of 20250127 this generates only partial documentation, and some changes may be necessary.

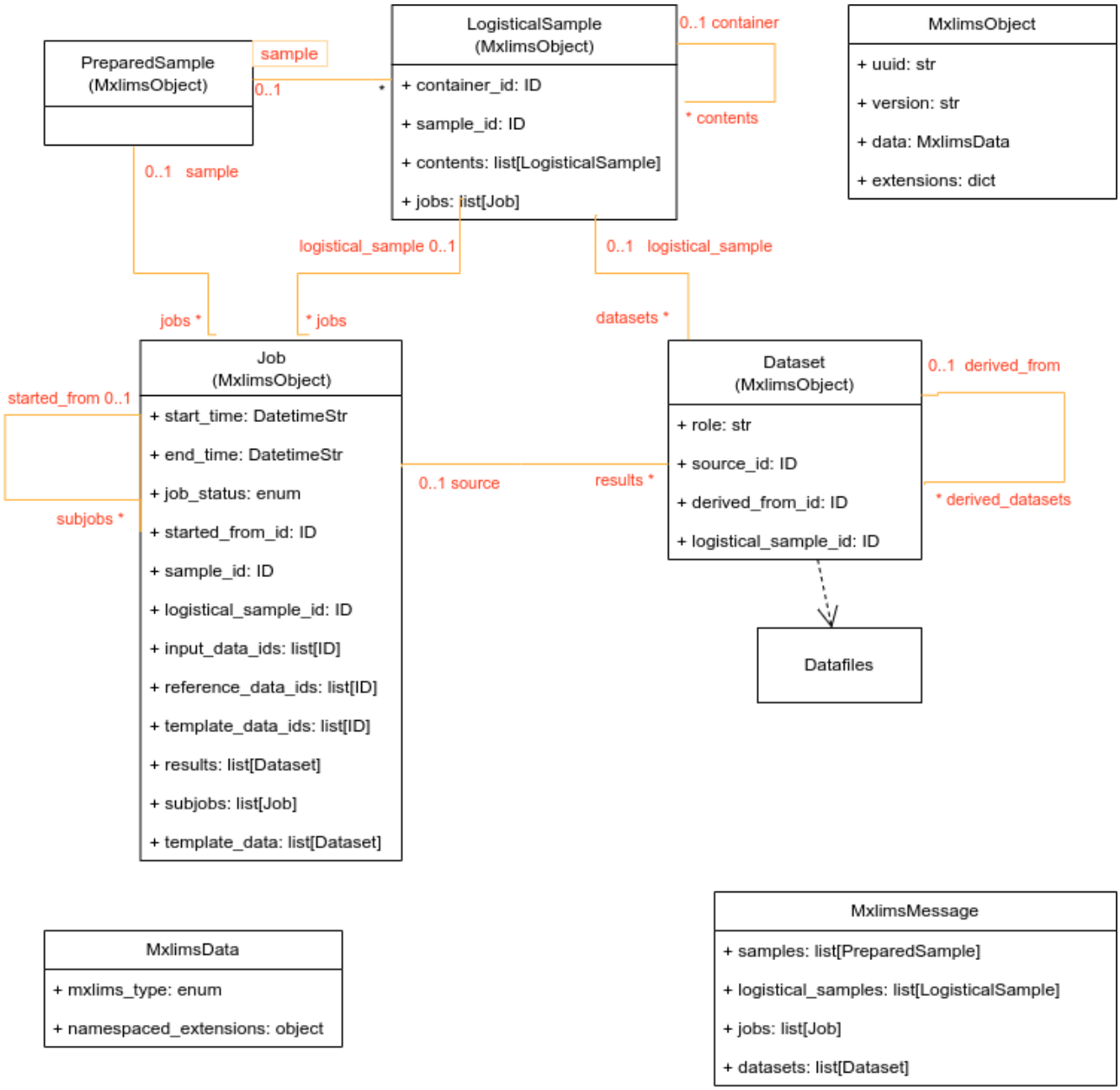

Figure 2 JSON implementation of the core MXLIMS model. The two-way inter-object links supported by the model are shown in orange and red for information, but the actual data are those given inside the classes, both as ..._id and as lists of contents (see discussion above). The MxlimsMessage class shows the most general JSON document that can contain any combination of objects. In actual use it is expected that applications would make more restricted schemas, limited to the kind of contents that the application supports.