MXLIMS model overview

Table of Contents

Version 0.6.5

The MXLIMS model basic structure is now stable, but the contents continue to develop; most changes reflect the needs of sample shipment. The full model documentation can be found in the docs/html/ directory. There are a number of individual model changes, listed in the ReleaseNotes.md file

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 types, 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 model changes without unpredictable consequences.

The main requirement of the model 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. Another is to be able to support a LIMS system, including provenance tracking. 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 have very few core classes, and 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 extra keyword-value parameters without abusing 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 Daniel (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, Dennis Stegmann, and May Sharpe (SLS) for multi-synchrotron shipment and SSX use cases.

Core MXLIMS model

The core model is illustrated in Figure 1 (below)

The model is organised around four core abstract classes:

- Sample, describing sample content and composition
- LogisticalSample, describing the objects that you ship, mount and investigate in experiments: Plates, Pucks, Wells, Drops, Crystals, ...
- Dataset, describing the actual data produced
- Job, describing the experiments and calculations that produce the data.

These four classes, together with the links between them, allow you to track the provenance and history of all data produced. Links between objects are stored using (in essence) foreign keys as in a database (`sampleId, inputDataIds`, …), pointing to the UUIDs of other objects. All model objects are subclasses of the four core classes; the actual data for each class of model object are defined separately.

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). 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 field is discouraged, but allows for adding extra information quickly without breaking or abusing the model. The `namespacedExtensions` field is intended for site-specific extensions that are defined by published, schemas. Namespaces could be e.g. ‘ESRF’ or ‘GPhL’ and the relevant schemas are to be maintained by the owner of the namespace.

The **Sample** class describes the material of the sample, including contents and components, creation date, identifiers and batch numbers. The distinction between Sample and logisticalSample arises from the fact that particularly in serial crystallography a given Sample can be spread over several

Crystals in a well (slurry, cubic lipid phase mounting,…) or even sample holders. The Sample comes with two links to other Samples: `medium` and `mainComponent`, allowing you to specify separately an actual sample (e.g. class ‘MacromoleculeSample’) the crystallisation buffer (class ‘Medium‘), and the macromolecule being crystallised (class ‘Macromolecule’). The components of the actual sample would then be the sum of the components specified for the medium, the macromolecule, and the sample itself. From a modelling point of view it would have been cleaner to have a single sample class and specify all components *de novo* for each instance, but the alternative proposed here allows you to specify macromolecule and buffer (if desired) only once even for a shipment that might contain hundreds of samples with the same macromolecule.

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 can only be identified during the experiment, so the Crystal object would often be generated only during the actual experiment. Jobs (experiments) can be applied to the several types of LogisticalSample that can correspond to a single Sample, 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) can start other jobs (e.g. X-ray centring, characterisation, or acquisition) if desired.

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. Except for the Macromolecule class there is no class that can correspond to a sample component as such, be it Lysozyme, dithiothreitol, or fetal calf serum, so we mostly cannot have a single object with a `uuid` that we can refer to for sample components. Except for the Medium and Macromolecule classes 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.

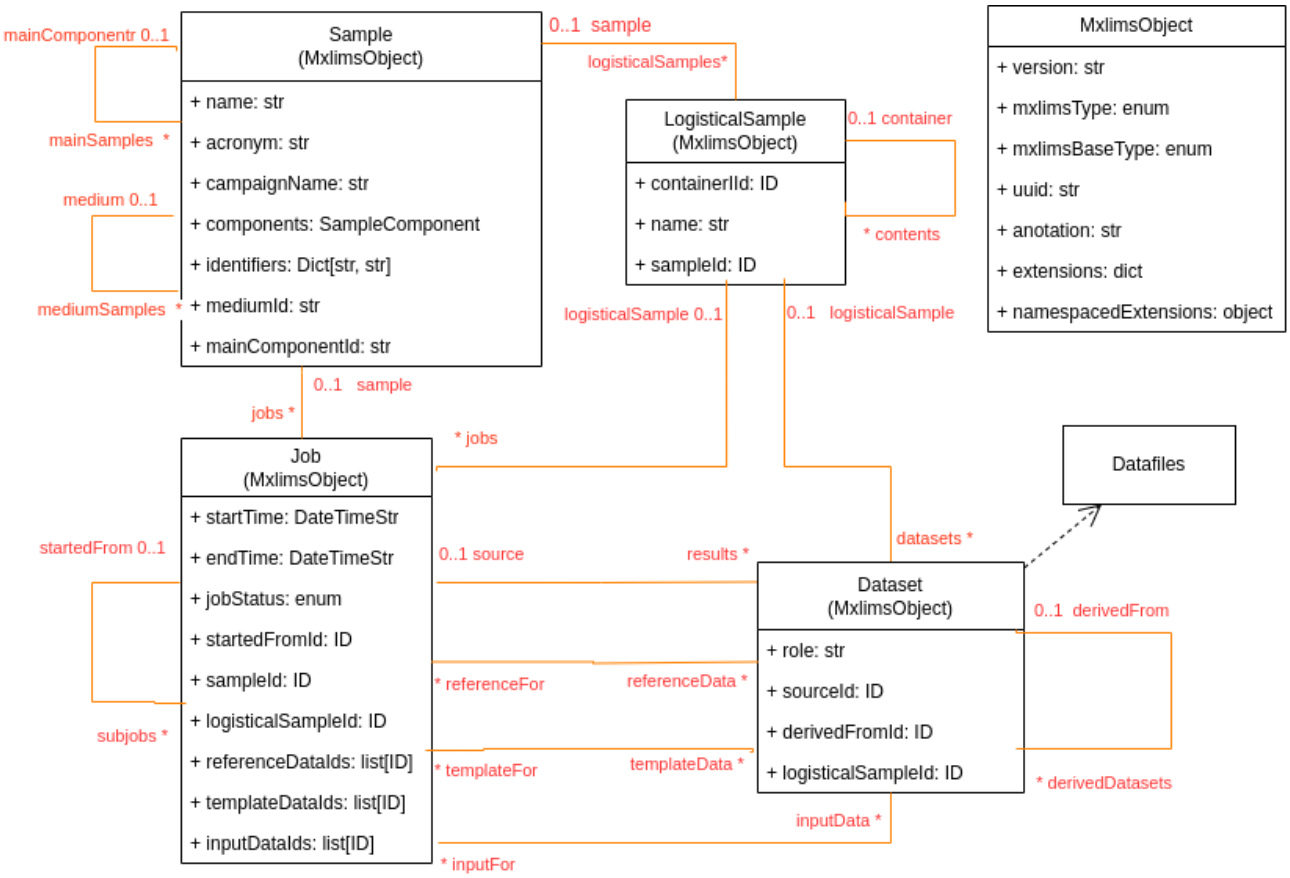

Figure 1 **<u>The core model</u>** The diagram shows the four core abstract classes, together with their attributes and the foreign key fields that are used to specify inter-object links. The orange lines show the inter-object links, all of which are two-way, together with the associated role names. These form part of the API defined by the model, but are not stored separately. Instead they must be derived by model implementations from the stored foreign keys.

Model Contents

The core abstract classes can be used to make an unlimited number of subclasses in order to model various scientific fields. At the moment the model is limited to macromolecular crystallography, including shipping, experiments, and processing. Each particular class has a defined set of attributes (‘metadata’), as well as limits on which types of objects it can be linked to, within the bounds of the inter-object links defined in the core. The model does not allow new inter-object links to be specified in the metadata. The classes currently supported in the model are shown in figures 2 and 3.

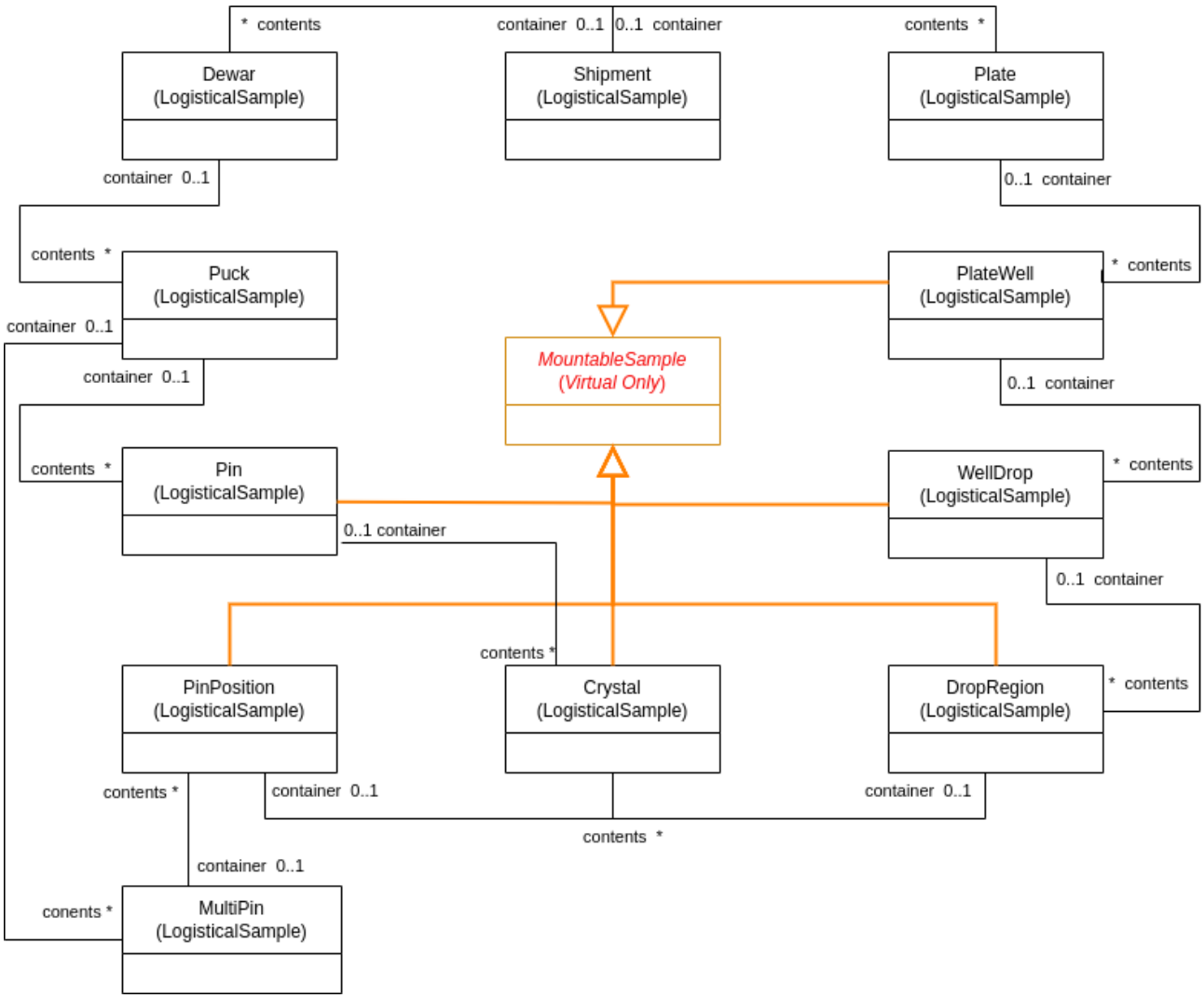

Figure 2. Logistical Samples. The figure shows the organisation of logistical samples (roughly: sample containers and crystals), with their container-content relationships. The '*MountableSample*' class is not actually a class in the model, but was added to make the diagrams easier to interpret. The classes shown (orange, broad-headed arrows) as extensions of 'MountableSample' (Pin, PinPosition, PlateWell, WellDrop, DropRegion and Crystal) are the logistical sample classes that can be used to perform a single experiment, and therefore those that can be linked to a Sample, to Jobs, and to Datasets.

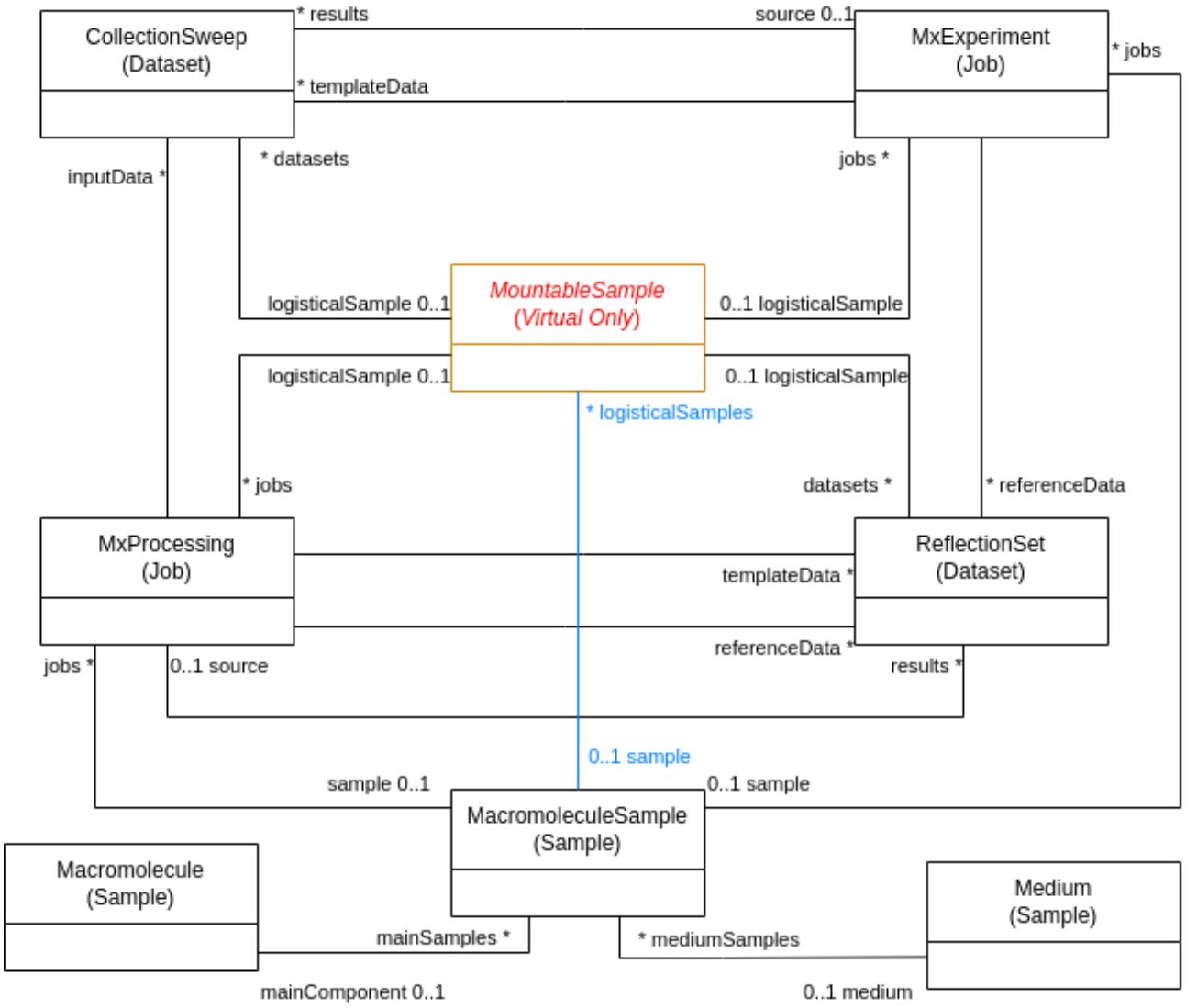

Figure 3, Model Classes. This figure shows the model classes in the current model, apart from the logistical sample classes (Figure 2), as well as the links between them. The ‘MountableSample’ is not an actual class in the model but was added to the diagram to stand in for the logistical samples that share the links shown here, as described in Figure 2. Note that the MxExperiment class does not have any inputData link. The link between MountableSample and MacromoleculeSample is no different from the other links in the diagram; it is shown in blue only to distinguish it graphically from the lines that cross it.

The class used for Crystallography is MacromoleculeSample; it has an optional link to a Medium, describing the crystallisation medium, and a Macromolecule, describing the macromolecule (‘mainComponent’) being studied. This allows you to specify Macromolecule and Medium only once in a shipment document, even if both are used in a large number of samples, while specifying sample-specific ligands in the sample itself. The total composition of the sample is the sum of all the components specified in Medium, Macromolecule, and MacromoleculeSample. Note that the expected space group and unit cell cannot naturally be specified either in the Macromolecule, since

the crystal form may not be the same everywhere, nor in the MacromoleculeSample, since there are no other macromolecule-specific attributes in this object. The expected space group, unit cell (and relative radiation sensitivity) are therefore specified in the MxExperiment template attached to the LogisticalSample.

Content details

Figure 4 shows an example of the structures you would use to specify a shipment, including the diffraction plan. The structure is rather more complicated than typical shipment data sheets, which are often specified as a single, flat spreadsheet with one line per sample, but this is largely a matter of spreading the same data slots over multiple containers. The attribute names should show what kind of information goes where. The information is distributed across objects describing the sample, the sample holder, and the experiment and processing jobs that are being planned. MXLIMS uses the same data objects for the diffraction plan, acquisition queue input, and specification of the final results. MxExperiment and MxProcessing objects are specified ahead of time to hold input parameters, and then fully populated once the experiment is run. CollectionSweep and ReflectionSet objects can be handled in a similar way, which would be the case if one wanted to pass control information to an acquisition queue, but for shipment diffraction plans the same kind of (partially populated) object is attached to the Jobs as a template, from which parameters are taken for the actual execution and results.

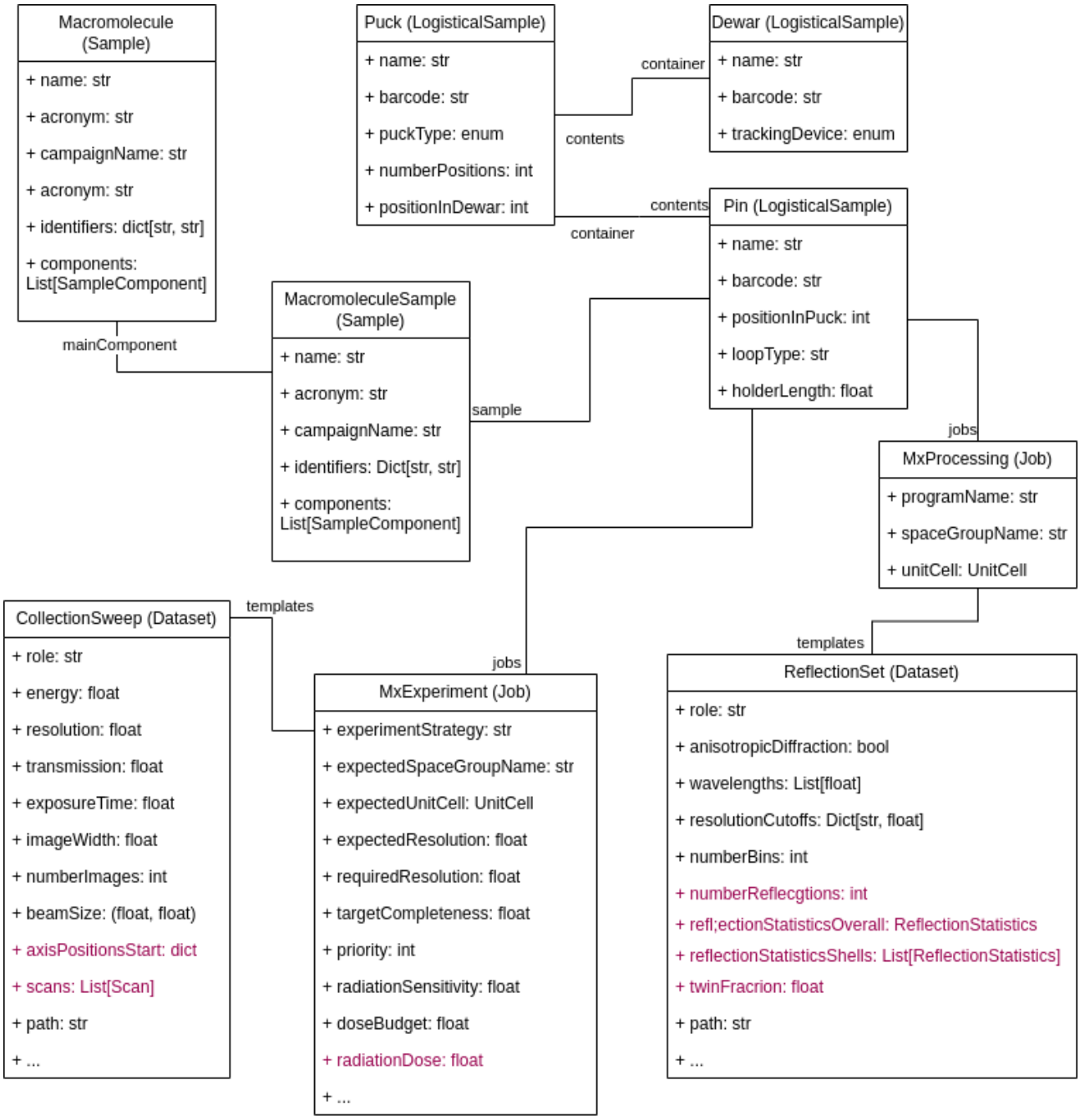

Figure 4 Extended shipment example. Fields in black would be specified as part of a shipment. Fields in purple are only filled in after the experiment/processing is run; they are shown here to illustrate the kind of additional information kept in these objects. The fields given for MxExperiment, CollectionSweep and ReflectionSet are incomplete, to save space, but should be enough to show what kinds of parameters are involved. Note that you could (if desired) specify the actual Dataset objects ahead of time instead of using templates. You can also have multiple templates, e.g. multiple processing jobs of different types, with attached template ReflectionSets.

Model Implementation

Specification

The complete model specification is given as a series of JSON schemas. Each object (including the abstract classes) has one entry in the schemas/data/ directory, that specifies the data attributes. The schemas/datatypes/ directory contains enumerations (e.g. space group names) and data structures (e.g. Unit cells) that are used within actual objects. The schemas/references directory contains objects used to specify inter-object links in the model specification and within JSON messages. All files in this directory are generated from the information in schemas/data/. The schemas/objects/ directory contains the completed model objects. This is a matter of combining the metadata specification (from schemas/data) and the core abstract class specification (from schemas/core) and adding constraints on the type of object that this object can be linked to.

Inter-object links

The handling of inter-object links in the specification is complex, since there have to be handled differently in the specification JSON schemas, the JSON data files, and the implementations. The diagrams above show the foreign-key IDs that are stored in implementations (with some differences between simple and database implementations), and the derived API fields that are used to access the links. The JSON schemas, on the other hand, follow the JSON files and represents links as a series of 'Ref' objects (e.g. DewarRef, MxExperimentRef, …). These have two fields, a fixed `mxlimsType`, and `$ref`. The `mxlimsType` is optional in JSON files but defines the type of object linked to and so provides enough information to generate the implementations. The `$ref` specifies a JSON reference composed of a regexp of the form`"^#/Pin/Pin[1-9][0-9]*$"` (for a link to a Pin object). This matches the structure of the MxlimsMessage.json and other message classes, which allows MxlimsMessage files to be validated against the schema so as to ensure that references are to objects defined within the document and of the correct type.

In an implementation (primitive or LIMS) it is necessary to handle two-way links and keep track of networks of objects even more than it is for message files. This is done by storing links as foreign keys that point to the uuid field of the linked-to object; for this reason the uuid is mandatory within implementations, even though it is optional in message files. The object uuid is not used for links in JSON files and so is not mandatory, but objects with no uuid will be given one on reading. It is possible to have a foreign-key uuid that does not match a known object, e.g. because the linked-to object has been deleted (foreign keys are not automatically cleaned in the reference implementation), or because data have been imported that contained a foreign key but not the object it points to.

This implementation has a potential problem in the case where imported data contain an object with a uuid that clashes with another object already in the system This could happen e.g. if multiple shipments refer to the same Medium or Macromolecule, or a sample is resubmitted. The system allows for three different behaviours in these cases:

- ‘reject_new’ skips the input object, and lets all foreign keys in the input point to the pre-existing object
- ‘update_old’ sets all defined attributes in the new object on top of the old object; foreign key links from either the input or existing data point to the same updated object
- ‘error’ raises an error.

JSON files

The JSON representation of the model follows the JSON schemas (as indeed it should). This fulfils the key requirement that it should be possible to validate a JSON message against the model specification. The way multiple objects are combined into a single message is shown in the messages/ MxlimsMessage.json and MxlimsMessageStrict.json file, that supports unlimited numbers of all currently defined objects.

It is a major problem for a JSON representation that JSON is limited to tree structures, whereas the current model describes an infinite object network. The same object could appear in multiple different contexts within the same file, forcing you to deal with either crosslinks or redundant copies of the data. Another problem is how to handle links to objects that are not part of a given message e.g. because you are referring to a sample record that has been transferred previously and already resides within the destination LIMS system. The solution we have adopted is to avoid nesting JSON objects within each other and present them instead as a single top-level layer of objects, with a separate list for each object type. All inter-object links are treated as crosslinks, and are specified as JSON references, as described in the schemas/references/ directory. This makes the files somewhat less readable by humans (who are more at home with nested data structures than with networks of foreign keys) but allows you to validate that the object at the other end of a crosslink is indeed of the correct type. The MxlimsMessageStrict schema does not allow links to objects outside the message, whereas the MxlimsMessage schema does, using object stubs containing the foreign-key uuid to make sure that all links formally point to object within the file. These two message schemas both cover all classes currently in the model; it should be simple to make additional schemas that allow only a limited number of classes.

Simple python implementation

There is a prototype Python implementation of MXLIMS, using Pydantic. It implements the core model as shown in Figure 1. Objects are stored centrally in a few uuid:object dictionaries (kept as a singleton in the MxlimsImplementation._objects_by_id class attribute). Links are stored as foreign keys (as shown in Figure 1) and the attributes defined by the links drawn in Figure 1 are implemented as properties that interrogate the central dictionaries. The properties include type validation, so that using them you are protected from e.g. putting a Dewar inside a Puck. It is possible to modify the stored foreign keys directly (which might be necessary if you have a link to an object that is not currently available), but not to modify the Ids.

The implementation is generated from the information in the JSON schema specification, by running the mxlims/impl/generate_code.py script. It is completely based on Pydantic classes, which are generated (with two exceptions) using the `datamodel-codegen` command. Attribute names in Pydantic use snake_case, whereas the camelCase names used in JSON are kept as aliases. The actual objects to import are found in the mxlims/pydantic/objects directory, and these are generated directly by the MXLIMS code generation script, in order to provide the subclassing, type checking, and integration to the MxlimsImplementation class that is not available directly from Pydantic. The mxlims/pydantic/mxlims_messages.py, which contains the classes for generating MXLIMS messages is also generated directly by the MXLIMS script, as is the mxlims/impl/link_specification.yaml file that is used by the implementation as a data source to summarise the specification. Data import and export can be done by the MxlimsBase from_message_file, from_pydantic_objects, and export_message functions, and the helper functions these call. An example of the use and import/export of MXLIMS data can be seen in the rhfogh:rhfogh_mxlims_2025 branch of mxcubecore (note the mxcubecore/utils/mxlims.py file).

It should be noted that the behaviour of links upon deletion is implicit in the structure of the model. In the absence of mechanisms to ensure otherwise there are no cascading deletes, and foreign key attributes (such as sourceId or sampleId) remain populated even if the objects they point to are deleted. The many-to-many links referenceData, inputData and templateData are stored on the Job side of the link, with the effect that these links are removed when the relevant Job is deleted.

The prototype implementation does not include any mechanisms for access control, or for clearing the stored MXLIMS objects. This would have to be added as part of a proper implementation – which would anyway require integration with the beamline LIMS in some shape or form.

Database implementation

Providing a full database implementation is outside the scope of the MXLIMS working group, but the MXLIMS model was designed specifically to work well with a database implementation using a system like MongoDB. The same API can be supported in a database implementation with only minimal modifications, as shown in figure 5.

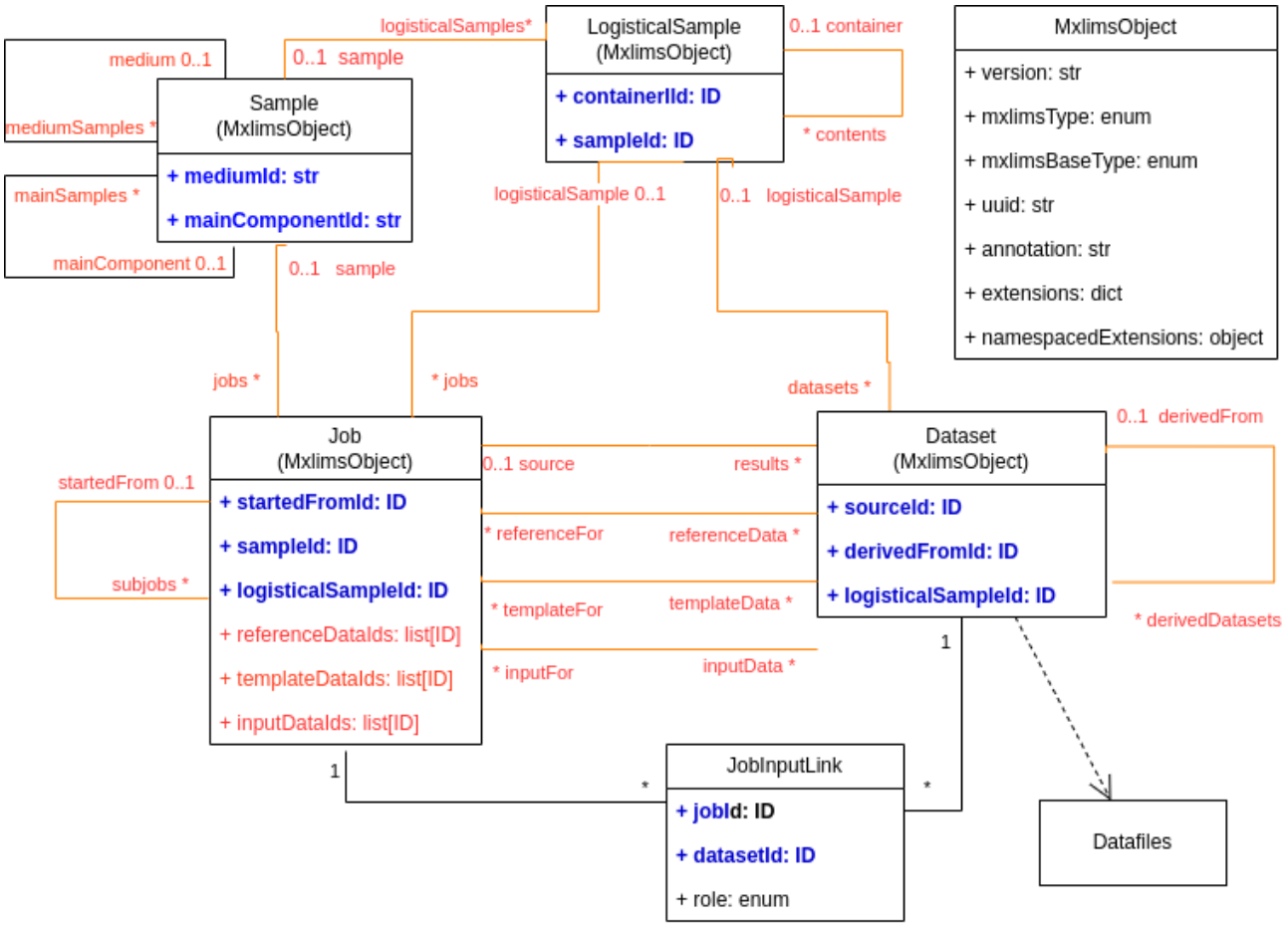

Figure 5, Database implementation The model can be implemented with one table for each core class, and with the same API as shown in figure 1, requiring only the additional JobInputLink table, in order to support the many-to-many links between the Job and Dataset classes. Foreign key attributes are in blue, while elements shown in orange and red are derived.

Since the model is already based on the use of foreign keys it translates seamlessly to a database implementation. The only difference regards the many-to-many links between the Job and Dataset classes (`inputData`, `templateData`, and `referenceData`), which are implemented using an additional table, JobInputLink. The attributes Job.referenceDataIds, Job.templateDataIds, and Job.inputDataIds are not part of the database implementation; to provide the same API as for other implementations these attributes must therefore be derived. The settings for cascading deletes (etc.) need to be be set appropriately in order to ensure the same behaviour as for different implementations (see preceding section).

The structure of the prototype implementation puts all the implementation code in the mxlims/impl/MxlimsBase,py in the MxlimsImplementation class. It should in principle be possible to change over to a database implementation by replacing this class.

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 (note the config file):

```
generate-schema-doc --config-file docs/schemadoc_config.json schemas docs/html
```

Conversion to pydantic is done in part with datamodel_code_generator (https://docs.pydantic.dev/latest/integrations/datamodel_code_generator/). In the longer run pydantic code generation is only relevant for the pydantic/data and pydantic/datatypes directories. The core classes must be implemented by hand, and the generation of code in pydantic/objects (and pydantic/references if ever required) is handled by bespoke code generation scripts/.

The command used is:

```
datamodel-codegen --input-file-type jsonschema --output-model-type pydantic_v2.BaseModel  --base-class mxlims.pydantic.MxlimsBase.BaseModel --use-schema-description --use-double-quotes --disable-timestamp --use-default --target-python-version 3.10 --snake-case-field --use-exact-imports --capitalise-enum-members --use-title-as-name --use-one-literal-as-default --input mxlims/schemas --output mxlims/pydantic
```

when run from the mxlims_data_model/ directory.

The resulting pydantic classes are subclassses of `mxlims.pydantic.MxBaseModel.BaseModel` which sets the necessary configuration

The JSON schema specification uses camelCase field names, which is the standard, while the pydantic classes are generated using snake_case field names, with the camelCase names as aliases.

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.

- Where “oneOf” constraints have been retained the pydantic model sometimes lack some of the model constraints, e.g. the constraint that a Dataset cannot contain *both* a `sourceId` *and* a `derivedFromId`. These cases would have to be handled with custom code.