MALCOLM: A MIDDLELAYER FRAMEWORK FOR GENERIC CONTINUOUS SCANNING

T. Cobb, M. Basham, G. Knap, C. Mita, M. Taylor, G. D. Yendell, Diamond Light Source Ltd, Oxfordshire, UK
A. Greer, Observatory Sciences, Cambridge, UK

Abstract

Malcolm is a middlelayer framework that implements high level configure/run behaviour of control system components like those used in continuous scans. It was created as part of the Mapping project at Diamond Light Source to improve the performance of continuous scanning and make it easier to share code between beamlines. It takes the form of a Python framework which wraps up groups of EPICS PVs into modular "Blocks". A hierarchy of these can be created, with the Blocks at the top of the tree providing a higher level scanning interface to GDA, Diamond's Generic Data Acquisition software. The framework can be used as a library in continuous scanning scripts, or can act as a server via pluggable communications modules. It currently has server and client support for both pvData over pvAccess and JSON over websockets. When running as a webserver this allows a web GUI to be used to visualize the connections between these blocks (like the wiring of EPICS areaDetector plugins). This paper details the architecture and design of framework, and gives some examples of its use at Diamond.

INTRODUCTION

Diamond Light Source [1] is a third-generation 3 GeV synchrotron light source with 35 independent experimental stations attached to photon beamlines. A number of these beamlines use a technique called continuous scanning where motors are moved in a continuous trajectory while a detector takes a number of data frames synchronized with hardware trigger pulses as illustrated in Fig. 1. This technique increases the efficiency of an experiment by reducing the number of times a motor has to decelerate, settle and accelerate, effectively decreasing the scan dead-time.

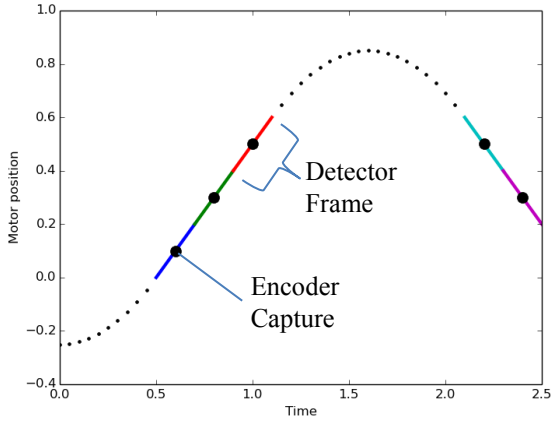

Figure 1: Detector frames synchronous with a motor undergoing a snake trajectory scan.

THE MAPPING PROJECT

Diamond has multiple beamlines capable of conducting mapping experiments where the sample is moved through the X-ray beam and a frame is sampled on one or more detectors at each point. In 2015, Diamond created the Mapping Project [2, 3] to enable all beamlines that conduct mapping to benefit from common features like live visualization and processing of data and optimisations like continuous scanning. A set of 5 beamlines with diverse techniques and detectors were selected to participate in the project to ensure that the stack of components being developed (as shown in Fig. 2) could be easily deployed on multiple beamlines and support a variety of experimental equipment and instrumentation.

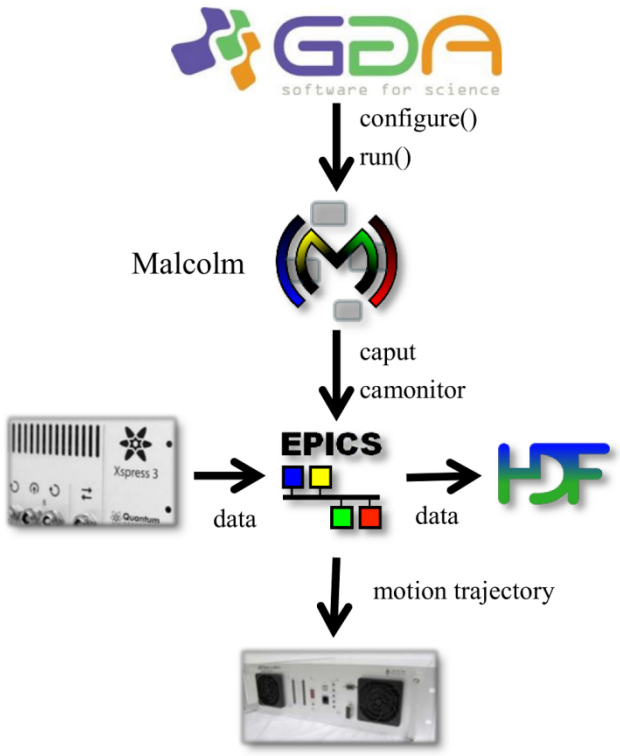

Figure 2: Mapping Project component stack.

New developments were made in trajectory scanning the Delta Tau Geobrick [4], writing multi-dimensional data using HDF5 SWMR [5], live processing and visualization of data in GDA [6] and DAWN [7], and the Malcolm [8] middlelayer that is the subject of this paper.

MALCOLM

Malcolm provides an abstraction layer on top of EPICS [9] that wraps up groups of PVs and presents a higher level scanning interface to GDA via pvAccess [10]. This means that it can take care of the variations in triggering schemes between different beamlines, and GDA only needs to pass high level scan parameters such as motion trajectory and exposure time down, rather than needing to know how all the underlying devices are wired up.

Blocks, Methods and Attributes

Malcolm defines Blocks, each with a series of Methods and Attributes, much like instances of classes in an object oriented language. These are arranged in a hierarchy as shown in Fig 3.

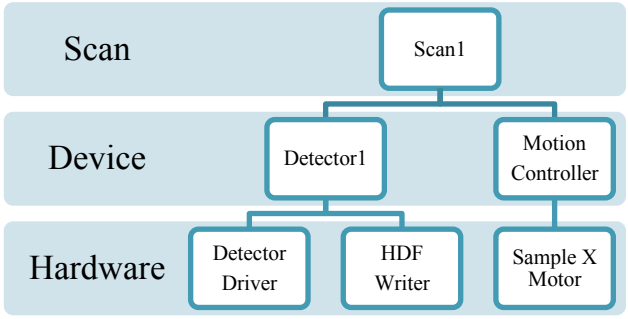

Figure 3: Layers of Malcolm Blocks.

The lowest level of Blocks in the Hardware Layer are just a collection of Attributes that correspond to EPICS devices like a single motor, or the areaDetector [11] HDF writer plugin.

The Device Layer above this contains Blocks that represent a whole Detector or Motor Controller. They have `configure()` and `run()` Methods and an Attribute that shows what state it is currently in. When these methods are called they co-ordinate their child Hardware Blocks to perform a scan according to the parameters they are passed.

The Scan Layer at the top exposes a scanning interface to GDA. They also have configure and run Methods that again co-ordinate their child Device Blocks to perform a scan.

Composing Blocks from Controllers and Parts

A key aim of the Mapping project was to make reusable components that could be deployed across multiple beamlines. Blocks with Methods and Attributes make a good interface to the outside world, but aren't the right size to make re-usable chunks of code. For instance, the configure Method of a Detector Block in the Device Layer might co-ordinate an areaDetector driver with a chain of a stats plugin and an HDF writer plugin, but writing a single object that did all of this would preclude using it with the same detector without the stats plugin.

Malcolm solves this by forming a Block by composition from a co-ordinating Controller and some behaviour defining Parts. The detector example above is illustrated in Fig. 4.

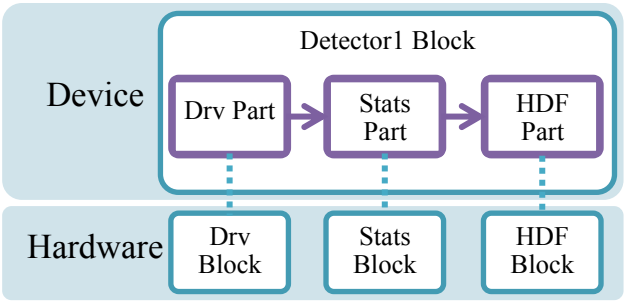

Figure 4: Layers of Malcolm Blocks.

Each element in the areaDetector driver and plugin chain is defined by a Block in the Hardware Layer that defines the PVs it exports. The Device Block is then formed from a single Controller and one Part for each child Block which contains the logic that shows how to use that Hardware Block within the current scan. This allows the external interface provided by PVs to be separated from small self-contained pieces of code that implement one particular type of logic.

Block definitions in YAML

As the number of objects in a system grows, configuring them becomes a more complicated task. Malcolm allows objects to be instantiated by writing configuration parameters in a structured YAML [12] file as shown in Fig. 5.

```
- builtin.parameters.string:
    name: mri
    description: Malcolm Resource ID

- builtin.parameters.string:
    name: prefix
    description: The root PV for records

- builtin.controllers.StatefulController:
    mri: $(mri)

- ADCore.includes.adbase_parts:
    prefix: $(prefix)

- ca.parts.CADoublePart:
    name: gainX
    description: Image Gain in X
    pv: $(prefix):GainX
    rbvSuff: _RBV
    widget: textinput

...
```

Figure 5: YAML file for simDetector driver Block.

These YAML files define what collection of Controller and Parts make up a specific Block. They define the parameters that should be passed to them, and how those parameters should be used to create Controller and Part instances. They are also used to create other Block instances that are defined in other YAML files as well as

instantiating include files that contain commonly used Parts.

This whole system is made possible because Parts, Controllers, Blocks and includes define the type and description of the arguments that should be passed to them. This yields several benefits:

- Arguments specified in YAML are validated and sensible error messages output when configuring an entire Malcolm system.
- Documentation for the arguments to be passed to each Part, Controller and Block is automatically generated.
- It paves the way for a future tabular GUI editor for these YAML files.

Loading and saving definitions

Another benefit to the split between logic and interface is that it gives an ideal place to load and save definitions of Malcolm Blocks. For example, on a detector Block, there might be a number of parameters like the trigger mode that Malcolm does not control, but needs to be set to a particular value in order for a scan to work. Malcolm has the concept of saving and loading a design, and all configuration Attributes by default are included in this design. This means that when a scan is working, the detector design can be saved, and then the design will be checked and restored if necessary before each scan. It also allows an Attribute that warns if the design has been modified and needs to be saved again.

This is useful because it allows end users like beamline scientists to use this ability to turn on and off features of Malcolm and create their own saved designs for custom scans without having to restart or reconfigure Malcolm.

Asynchronous helper methods

To make continuous scanning fast, care needs to be taken about which Attributes can be set at the same time and whether completion needs to be waited for before more Attributes are set. For example, let us consider setting 3 Attributes on a Block. Exposure and period need to be set first, then when they have completed the xml Attribute can be set with a value that has to be calculated in a time consuming manner. Figure 6 shows the inefficient but readable synchronous style:

```
def configure(self, block, params):
    block.exposure.put_value(
        params.exposure)
    block.period.put_value(
        params.exposure)
    xml = self.time_consuming_f(params)
    block.xml.put_value(xml)
```

Figure 6: Synchronous style configure method.

This code will put and wait for exposure, then put and wait for period, then calculate a value, then put and wait for xml. This can be made more efficient by doing both initial puts at the same time, then doing the calculation before waiting for completion of the two initial puts. Figure 7 shows this code written in an efficient but less readable callback style:

```
def cb(self, value):
    self.q.put(value)

def configure(self, block, params):
    self.q = Queue()
    block.exposure.put_value_async(
        params.exposure, self.cb)
    block.period.put_value_async(
        params.exposure, self.cb)
    xml = self.time_consuming_f(params)
    for i in range(2):
        assert not isinstance(
            q.get(), Exception)
    block.xml.put_value(xml)
```

Figure 7: Callback style configure method.

This is much more efficient because the time where the synchronous code was waiting for puts to complete is now spent calculating a value. Unfortunately the code is now less readable because you have to read the callback function to follow the control flow. Figure 8 shows how this can be improved by using a futures style.

```
def configure(self, block, params):
    fs = block.put_values_async(
        exposure=params.exposure,
        period=params.exposure)
    xml = self.time_consuming_f(params)
    block.wait_futures(fs)
    block.xml.put_value(xml)
```

Figure 8: Futures style configure method.

This does exactly the same as the callback style, but wraps up the callback functions in Future objects that can be waited on. This restores the linear nature of the code and makes it readable again. Malcolm exposes the helper methods needed to allow this futures style to be written side by side with the synchronous style to allow code to retain its readability.

COMMUNICATING WITH MALCOLM

Although Malcolm can be run standalone as a library, the most common use case is to add some communications modules to it to allow it to be communicated with from the outside. The two used at Diamond are websockets [13] to allow a web GUI configuration view and pvAccess for control system access from GDA.

Websockets

This communications module exposes the structure of Malcolm Blocks via a JSON [14] protocol over websockets. There is a client MalcolmJS [15] library that is used to create a web GUI to allow configuration of the underlying Blocks from a web browser. It is generally used to wire Hardware Blocks together and load/save configurations. Figure 9 shows its use in wiring together Blocks to configure a PandABox [16].

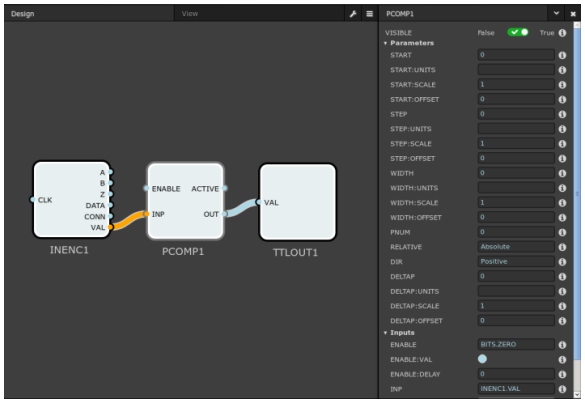

Figure 9: Malcolm web GUI configuring a PandABox.

pvAccess

GDA communicates with the top level scan Block, configuring it with a set of parameters then telling it to run. These are done using the pvaPy [17] Python bindings to pvAccess in Malcolm, and the pvAccessJava [18] bindings on the GDA side. Methods support Get and Monitor for introspection of requested arguments and RPC for calling them. Attributes support Get and Monitor for introspection of value, timestamp, alarm status and descriptive metadata, and Put for changing value.

Serialization of Blocks in Malcolm

To allow the web GUI to introspect the Blocks that can be connected and which parameters can be set on them and to allow GDA to introspect configure arguments, Block structures can be serialized. All data classes in Malcolm have `to_dict()` and `from_dict()` methods allowing their structure to be exposed to the outside world. Figure 10 shows the serialized version of a Block as described in pvData Meta Language [19].

```
Block :=

malcolm:core/Block:1.0
    BlockMeta   meta
    Attribute   health
    {Attribute  <attribute-name>}0+
    {Method     <method-name>}0+
```

Figure 10: Block structure.

Every Block has some metadata about itself, an Attribute displaying its health, and optional additional Attributes and Methods.

Figure 11 shows the serialized version of an Attribute.

```
Attribute := Scalar | ScalarArray | …

Scalar :=

epics:nt/NTScalar:1.0
    scalar_t    value
    alarm_t     alarm       :opt
    time_t      timeStamp   :opt
    ScalarMeta  meta        :opt
```

Figure 11: Attribute structure.

This is conformant to an EPICS V4 Normative Types [20] NTScalar because the value, alarm and timeStamp fields are present, but the metadata like descriptor, display and control have been moved to a child ScalarMeta object. This allows metadata to be specified separately to transient fields like value and timeStamp, allowing the same Meta objects to be reused to specify the arguments that should be passed to a Method.

Figure 12 demonstrates the serialized version of a Meta object.

```
ScalarMeta := NumberMeta | StringMeta …

NumberMeta :=

malcolm:core/NumberMeta:1.0
    string    dtype
    string    description
    string[]  tags       :opt
    bool      writeable  :opt
    string    label      :opt
    display_t display    :opt
    control_t control    :opt
```

Figure 12: ScalarMeta structure.

It contains some of the meta information that would normally appear in the NTScalar, with some additions for specific Meta objects like the dtype (e.g. uint32) for the NumberMeta.

Serialization of Blocks in GDA

To facilitate communication between GDA and Malcolm over the pvAccess channel, a serialization library was built in Java to convert Java objects into PVStructures and vice versa. The pvMarshaller [21] library takes any Java object, inspects its members using the Java Reflection API, and creates a representation of that object in a PVStructure as shown in Fig. 13.

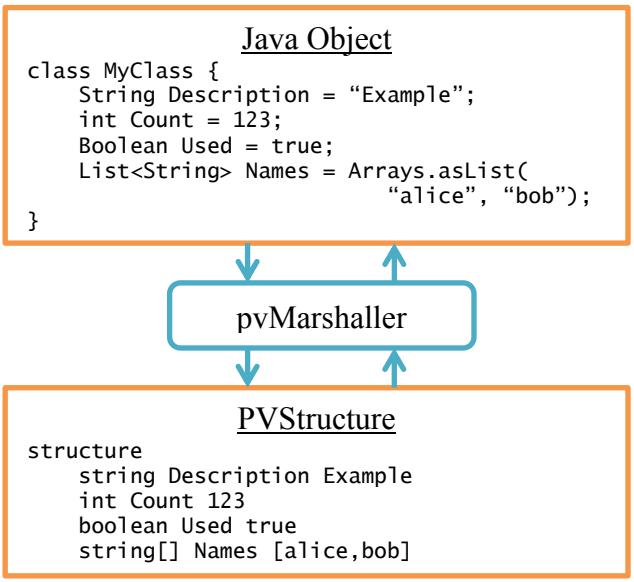

Figure 13: pvMarshaller serialization and deserialization.

When deserializing from a PVStructure object into a Java object, a new Java object of the target class is created and its members populated with the values from the source PVStructure. If the target class is not known, there are mechanisms for determining it from the PVStructure Key, and if this is not specified, a Java Map object is created consisting of the key-value pairs from the PVStructure. Advanced features of the library include the ability to specify members to exclude from serialization, and the ability for users to create custom serializers and deserializers for classes whose member structures don't map exactly to the desired PVStructure.

CONCLUSION

Malcolm was created as an application to support generic continuous scanning across multiple diverse beamlines at Diamond. It aims to produce flexible, concise and maintainable applications by providing tools to encourage reuse of code, reducing future support effort for the maintaining groups. Its pluggable communications modules allow it to be configured from a web GUI and to act as a service among the other elements that make up experimental control. A rollout project is underway to expand its use from the 5 initial beamlines to any Diamond beamline that could benefit from continuous scanning.

REFERENCES

[1] R. P. Walker *et al.*, "Commissioning and Status of the Diamond Storage Ring", in *Proc. IPAC 2017*, Copenhagen, Denmark.

[2] M. Basham, J. Filik, "Generic Mapping Scans at Diamond Light Source", in *Proc. NOBUGS 2016*, Copenhagen, Denmark.

[3] R. Walton, "Mapping Developments at Diamond", in *Proc. ICALEPCS 2015*, Melbourne, Australia.

[4] Delta Tau Geobrick, http://www.deltatau.com

[5] N. Rees, "Developing Hdf5 For The Synchrotron Community", in *Proc. ICALEPCS 2015*, Melbourne, Australia.

[6] E. P. Gibbons, M. T. Heron, N. P. Rees, "GDA and EPICS: working in unison for science driven data acquisition and control at Diamond light source", in *Proc. ICALEPCS 2011*, Grenoble, France

[7] M. Basham *et al.*, "Data Analysis WorkbeNch (DAWN)", *Journal of synchrotron radiation*, vol. 22, pp. 828-838.

[8] Malcolm, http://pymalcolm.readthedocs.io/en/latest/.

[9] EPICS, http://www.aps.anl.gov/epics/.

[10] pvAccess, http://epics-pvdata.sourceforge.net

[11] EPICS areaDetector, http://cars9.uchicago.edu/software/epics/areaDetector.html

[12] YAML, http://yaml.org

[13] websocket, https://www.websocket.org

[14] JSON, http://www.json.org

[15] I. Gillingham and T. Cobb, "MalcolmJS: a Browser-Based Graphical User Interface", presented at ICALEPCS'17, Barcelona, Spain, May 2016, paper THPHA184, this conference.

[16] S. Zhang et al, "PandABox: A Multipurpose Platform For Multi-technique Scanning and Feedback Applications", presented at ICALEPCS'17, Barcelona, Spain, May 2016, paper TUAPL05, this conference

[17] S. Veseli, "PvaPy: Python API for EPICS PV Access", in *Proc. ICALEPCS 2015*, Melboure, Australia.

[18] pvAccessJava, https://github.com/epics-base/pvAccessJava

[19] pvData Meta Language, http://epics-pvdata.sourceforge.net/docbuild/pvDataJava/tip/documentation/pvDataJava.html#pvdata_meta_language

[20] EPICS V4 Normative Types, http://epics-pvdata.sourceforge.net/alpha/normativeTypes/normativeTypes.html

[21] pvMarshaller, https://github.com/DiamondLightSource/pv-marshaller