WHEN HARDWARE AND SOFTWARE WORK IN CONCERT

M. Vogelgesang, A. Kopmann,
Institute for Data Processing and Electronics, Karlsruhe Institute of Technology, Germany
T. Faragó, T. dos Santos Rolo, T. Baumbach,
Institute for Photon Science and Synchrotron Radiation, Karlsruhe Institute of Technology, Germany

Abstract

Integration of control and data processing is required to operate an X-ray beam line in the most efficient way. Although control systems such as TANGO address the distributed nature of experiment instrumentation, standardized APIs that provide a uniform and asynchronous device access are still missing. Moreover, process control and data analysis are not yet integrated in the most efficient way. In this paper we present concepts and implementation details of *Concert*, a Python-based framework to integrate device control, experiment processes and data analysis which we used to control calibration procedures and high-speed tomography scans.

INTRODUCTION

Modern synchrotron beamlines require precise control of a large range of diverse hardware devices. To increase modularity and improve fail safety, most devices are accessed via network APIs. Unfortunately, it is virtually impossible to describe semantically correct device hierarchies with the existing solutions due to the low-level details. However, a device hierarchy, comprising an API that all devices of one device class adhere to, is necessary to build high-level process abstractions.

Another reason for a new high-level control system is a potential integration of process control and data processing. Process control allows us to build experiments that take the sample under investigation into account, e.g. triggering data acquisition. Integrated data processing can reduce wasted time by allowing on-site investigation of the preliminary results. Current control systems do not integrate control and processing at all.

In this paper, we present a Python-based high-level control system called *Concert*. It is primarily designed for high-speed radiography and tomography but is general enough for all kinds of experiments. The main requirements and constraints during its development were:

1. *Concert* should provide a standardized device hierarchy and API independent of the underlying device access, suitable for rapid development of automation processes.

2. Asynchronous device control is mandatory to reduce the time spent on synchronization and hence improve throughput. It should be as transparent to the user as possible.

3. Existing technologies should be used wherever possible to reduce development time. This also requires open interfaces and easy extensibility.

4. Additionally, modern development paradigms and tools should be employed to maximize quality and robustness.

With our system in place, not only do we reduce the duration of an experiment but also enable process control and high-volume data processing for novel experiments such as on-line reconstruction of tomographic data sets and image-based data acquisition.

RELATED WORK

TANGO [1] and EPICS [2] are the most common control systems used in the synchrotron community. Both systems provide hardware-independent and distributed access of a variety of devices. However, neither of them provide a high-level device and process control API. In the TANGO camp, Sardana provide a higher-level layer TANGO [3]. However, due to its tight integration it is not (yet) as platform-independent as we need it.

SPEC is a commercial X-ray diffraction control system that has been extended for use with other types of experiments as well. Due to a custom scripting language and SPEC-specific idioms, interoperability is slightly reduced. Moreover, being a closed product, SPEC requires changes from the vendor to include new core features.

A fully automated software-controlled tomography setup is used at the TOMCAT beamline at the Swiss Light Source for high-throughput X-ray tomography experiments [4]. To achieve long intervention-free scan times, the system is built statically and can neither be easily extended nor re-used for other experiment types.

CONCEPTS AND IMPLEMENTATION

Fundamentally, a control system maps real hardware device access to some kind of software abstraction. This abstraction exposes certain parameters and operations specific to a device. In *Concert*, every device *type* (e.g. motor, detector, etc.) derives from a base `Device` class. This device class provides basic functionality shared by all devices: a parameter interface, locks and state information.

By deriving device type classes from the `Device` class, we obtain a class hierarchy as shown in Fig. 1. This hierarchy *guarantees* reusability of top-level interfaces due Python's polymorphism property. Hence, a higher-level process which uses such an interface will work with all

ISBN 978-3-95450-139-7

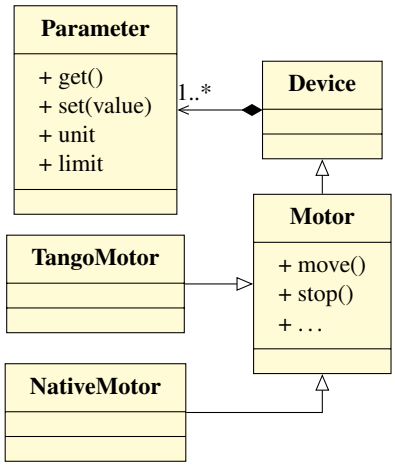


Figure 1: Simplified class hierarchy. `Parameter` and `Device` classes form the basis to build device types (a `Motor` class in this case) and subsequent concrete implementations.

implementations of that interface. For instance, the base `Camera` class provides a `grab` method to acquire one frame. This interface has many hardware-specific implementations which a higher-level process can use without knowing the hardware-specific details just by calling `grab`.

Device Parameters

Each device exposes an arbitrary amount of parameters which are uniquely identified by a name and map to a `Parameter` object. The user can read and – depending on the access rights – write values into that `Parameter` object. Each access is logged and the messaging system notifies interested parties about a change to the device. Before the value is actually passed to the hardware device, it is validated against the Parameter's soft limit and checked for physical unit compatibility.

Concert provides both, an enumeration and a dictionary-like API to fetch parameters in a programmatic way. This is used to query values for displaying information in user interfaces:

```
for p in device:
    value = p.get().result()
    print("{0} -> {1}".format(p, value))

p = device['position']
p.set(2 * q.mm)
```

Each `Parameter` can also be accessed by its name via attribute lookup. Hence, one can set the position also with:

```
device.position = 2 * q.mm
```

Device classes not only provide property-like attributes but can also implement device-manipulating methods that change the state internally (for example starting a continuous motion). To provide a valid interface for all devices of the same type, each base class of that device type either implements its methods directly or calls device-specific methods. The latter is implemented by having the public method (e.g. `Motor.home`) call a private method (`Motor._home`) that is in turn implemented by the concrete device class. This way the interface contains the required `home` method regardless of the concrete device knowledge.

Unit Validation

Parameter value is *always* associated with a physical unit as provided by the `pint` module. `pint` includes all base SI units, SI-prefixes (e.g. *kilo*meters) and derived units (e.g. meters per second). It is also used to calculate quantities (numerical values bound to a unit) and to convert the result to a final target base unit.

We enforce the usage throughout the system, from the base parameter class to the user interface to reduce the chance of invalid input. Whereas in most systems, the user can only input what the device accepts (which they also must know!), any of the following equivalent statements is possible in *Concert*:

```
cam.exposure_time = 0.002 * q.s
cam.exposure_time = 2 * q.milliseconds
cam.exposure_time = q.minute / 120000.0
```

Asynchronous Operation

To reduce the total amount of time to control several independent devices, device accesses must be parallelized wherever possible. *Concert* provides asynchronous device access by encapsulating parameter access and device methods in *future* objects. A future represents a value produced by an asynchronous operation that is ready at some time in the future. It provides methods to query its state (`running()`, `cancelled()`, `done()`), to get the final result (`result()`) and to attach callbacks that are called when the future is done (`add_done_callback()`). To produce a future for a task, it is internally submitted to a `ThreadPoolExecutor`. Although Python's global interpreter lock (GIL) prevents real multi-threading, device operations will still run in parallel because I/O operations yield from execution.

By decoupling the parameter access from de-referencing the value, we can access several devices at the same time. The future objects themselves can then be used to synchronize with other asynchronous operations by chaining callbacks or waiting explicitly for a future to finish. Contrarily, attribute-like parameter access is always executed synchronously because setting the parameter cannot return a future:

```
# Synchronous access
motor.position = 1 * q.mm

# Asynchronous access returns a future
future = motor.set_position(1 * q.mm)
future.wait()
```

Because we cannot foresee the methods that are attached to derived device classes, we provide a Python decorator `@async` that wraps device methods to provide a similar asynchronous interface. This means that device developers

ISBN 978-3-95450-139-7

do not need to care about *how* parallelism is implemented. All device base class methods are already wrapped in this way:

```
# Partial class definition of Motor
class Motor(Device):
    @async
    def move(self, delta):
        self.position += delta

# Moving the motor asynchronously
f = motor.move(-2 * q.mm)
f.wait()
```

With concurrent operations, there is always the danger that two independent code paths access the same device asynchronously, thus leading to a race condition. To solve the problem, each device has a lock that is activated by the `with` statement:

```
with motor, detector:
    # Other processes cannot access
    # neither motor nor detector
    motor.move(1 * q.mm)
    data = detector.grab().result()
```

Because the `with` statement is executed atomically from Python's point of view, deadlocks with two devices and two processes are not possible.

Process Control

The basic device and parameter abstractions can be used to control devices manually. One could use these mechanisms to write simple scripts that perform certain tasks. However, these tasks often have a very similar functionality and differ mostly only in a few parameters. *Concert* reduces re-inventing the same procedures with a high-level process control module, that is the result of decomposing the recurrent logic from hardware-specific operations.

A very common procedure is a scan of a parameter (e.g. motor position, camera exposure time) and evaluation of a measure (e.g. pressure, image response) at each scan point. The `Scanner` object from the process module provides this in an abstract way. By passing `Parameter` objects instead of `Devices`, we can model every type of scan, for example a tomographic scan could look like this:

```
def acquire():
    return detector.grab()

scan = Scanner(stage['angle'], acquire)
future = scan.run()
positions, frames = future.result()
```

In this example, the scanner manipulates the rotation stage's angle and acquires one frame at each set position. When the process is finished, the positions and the actual frame data are returned. Although scanning can already process the measured data, it is not suitable for *feedback-based control* due to the missing feedback loop.

Feedback-based control is necessary for beamline tasks that need to evaluate a measure and act upon the result. In tomographic environments, the control algorithms often require an image-based feedback, e.g. focusing, sample alignment, etc. This logic can be decoupled into image-based metrics, control algorithms and feedback mechanisms.

We can find different metrics for diverse problems, or even for a single one like focusing, where one can use gradient information, variance or some other metric. Data assessment based on such metric is then used by a control algorithm which optimizes the parameters in order to achieve better results. Computed parameters are then projected onto the hardware by a high-level device API, which closes the feedback loop.

For example, a simple focusing process is nothing else but an optimization of a `Parameter` object (e.g. a motor position) with respect to a measured value (e.g. a metric denoting the sharpness of the current detector frame):

```
def measure():
    return np.std(detector.grab())

def on_finish()
    detector.stop_recording()

maximizer = Maximizer(motor['position'],
                      measure, bfgs)
detector.start_recording()
f = maximizer.run()
f.add_done_callback(on_finish)
```

Although this is the most general way of focusing, *Concert* also provides pre-defined process functions which encapsulate these processes and use default parameters for even faster prototyping:

```
from concert.processes.alignment import *
f = focus(detector, motor)
```

Because of the asynchronous approach employed in *Concert*, we are able to use the feedback loops in a *continuous* mode, i.e. we do not have to wait for the above-mentioned steps to finish one before each other, but let them run in parallel.

Data Processing

Process abstractions can employ basic data processing to enhance the control outcome (e.g. with NumPy). Up to now, control and data processing are two independent entities in control systems, with data processing commonly moved to a later offline stage. However, to improve the user's beam time, it is necessary to analyze the results right after or during the acquisition.

For this, we integrated our GPU-based data processing framework UFO [5] within *Concert*. With this framework, a user describes their processing workflow as a graph of processing tasks. The result is the transformation of data going from the roots to the leaves of the graph. The graph itself is transformed before execution to utilize all processing units (CPU cores, GPUs and remote nodes) as good as possible.

To integrate this framework in *Concert*, we used the process abstraction described before and export scalar properties in the same way as device parameters but flagged

ISBN 978-3-95450-139-7

with read-only status. Thus, a user can scan along a node's property "axis" to see the effects of a parameter change.

Session Management

User workflows center around the notion of a session. Sessions encapsulate different types of experiments, devices, data and processes associated with them. They can be nested to build "inheritance" relationships between those with common devices and processes. For example, one would define a base session for the whole beam line that contains every device that is installed. For each experiment, the necessary devices and processes are imported.

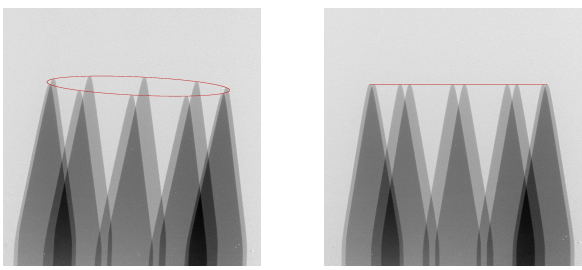
APPLICATION

We employed *Concert* in a typical tomographic beamline experiment to prove its versatility. In the following two sections, we show how we used *Concert* to calibrate the detector system and scanned a sample.

Pre-experiment Calibration

We have used *Concert* to focus a camera on the sample. The scintillator was moved along the beam direction and perpendicular to the focal plane with a linear motor. Based on the standard deviation of the image, the optimization process moved the motor towards or away from the sample until the metric was maximized.

After focusing, we used *Concert* to align the axis of rotation for the subsequent tomographic scan. An image sequence of a rotated and off-centered reference sample was recorded as shown in Fig. 2a. Reference points were determined and fit to an ellipse which was then used to align the center of rotation with the vertical axis, as shown in Fig. 2b. All these steps were performed without any user intervention by pre-defined *Concert* processes written in less than 50 lines of code.



(a) The elliptic path is caused by the misaligned axis.

(b) Aligned axis.

Figure 2: Result of rotation axis alignment.

High-speed Tomography Scan

We controlled a high-speed tomography experiment with *Concert* and our high-speed detector system [6] to investigate a living movement. The user only specified the desired frame rate and number of acquired projections. From this information, functions implemented using *Concert* calculated the required exposure time to avoid motion blur, opened and closed shutters, moved the sample in and out of the field of view, took dark and flat fields, scanned the sample and initiated the flat field correction. Using the center of rotation obtained from the calibration procedure before, we reconstructed the volume shown in Fig. 3 using the UFO framework and the filtered backprojection algorithm.

Figure 3: A projection and tomographic reconstruction of *Sitophius granarius*.

CONCLUSION

In this paper we presented *Concert*, a high-level control system that integrates TANGO, process control and data processing within a simple Python interface. It is an open source project hosted on GitHub

`https://github.com/ufo-kit/concert`

with up-to-date documentation at

`https://concert.readthedocs.org`.

Concert provides asynchronous, parallel operation and integration with our high performance computing framework for low latencies as well high-level process abstractions for rapid experiment development. In a complex beamline environment, *Concert* realizes smart instrumentation for fast tomography and high data rates but still being compatible with common beamline standards.

REFERENCES

[1] A. Götz, E. Taurel, J. Pons, P. Verdier, J. Chaize, J. Meyer, F. Poncet, G. Heunen, E. Götz, A. Buteau, *et al.*, "Tango a corba based control system," in *ICALEPCS*, 2003.

[2] L. R. Dalesio, M. Kraimer, and A. Kozubal, "Epics architecture," in *ICALEPCS*, vol. 91, pp. 92–15, 1991.

[3] "Sardana website." `www.tango-controls.org/static/sardana/`. Accessed: September 10th, 2013.

[4] K. Mader, F. Marone, C. Hintermüller, G. Mikuljan, A. Isenegger, and M. Stampanoni, "High-throughput full-automatic synchrotron-based tomographic microscopy," *Journal of Synchrotron Radiation*, vol. 18, pp. 117–124, Mar 2011.

[5] M. Vogelgesang, S. Chilingaryan, T. d. S. Rolo, and A. Kopmann, "Ufo: A scalable gpu-based image processing framework for on-line monitoring," in *Proceedings of The 14th IEEE Conference on High Performance Computing and Communication & The 9th IEEE International Conference on Embedded Software and Systems (HPCC-ICESS)*, HPCC '12, pp. 824–829, IEEE Computer Society, 6 2012.

[6] M. Caselle, S. Chilingaryan, A. Herth, A. Kopmann, U. Stevanovic, M. Vogelgesang, M. Balzer, and M. Weber, "Ultrafast streaming camera platform for scientific applications," *Nuclear Science, IEEE Transactions on*, vol. PP, no. 99, p. in print, 2013.

ISBN 978-3-95450-139-7