Concert Documentation

Release 0.9.0

Matthias Vogelgesang, Tomas Farago

August 16, 2014

Contents

Welcome to the Concert experiment control system documentation. This is the first place to answer all your questions related to using Concert for an experiment and developing more modules.

You can take a *quick guided tutorial* to see how the system is effectively used or take a closer in-depth look for special topics in our *user manual*.

CHAPTER 1

User documentation

1.1 Installation

1.1.1 openSUSE packages

We use the openSUSE Build Service to provide packages for openSUSE 12.2 until openSUSE 13.1. Add the repository first, e.g.:

```
$ sudo zypper ar http://download.opensuse.org/repositories/home:/ufo-kit/openSUSE_12.2/ concert-repo
```

and update and install the packages. Note, that you have to install IPython on your own, if you intend to use the `concert` binary for execution:

```
$ sudo zypper update
$ sudo zypper in python-concert
```

1.1.2 Install from PyPI

It is recommended to use pip for installing Concert. The fastest way to install it is from PyPI:

```
$ sudo pip install concert
```

This will install the latest stable version. If you prefer an earlier stable version, you can fetch a tarball and install with:

```
$ sudo pip install concert-x.y.z.tar.gz
```

If you haven't have pip available, you can extract the tarball and install using the supplied `setup.py` script:

```
$ tar xfz concert-x.y.z.tar.gz
$ cd concert-x.y.z
$ sudo python setup.py install
```

More information on installing Concert using the `setup.py` script, can be found in the official Python documentation.

To install the Concert from the current source, follow the instructions given in the *developer documentation*.

Installing into a virtualenv

It is sometimes a good idea to install third-party Python modules independent of the system installation. This can be achieved easily using pip and virtualenv. When virtualenv is installed, create a new empty environment and activate that with

```
$ virtualenv my_new_environment
$ . my_new_environment/bin/activate
```

Now, you can install Concert's requirements and Concert itself

```
$ pip install -e path_to_concert/
```

As long as `my_new_environment` is active, you can use Concert.

1.2 Tutorial

Concert is primarily a user interface to control devices commonly found at a Synchrotron beamline. This guide will briefly show you how to use and extend it.

1.2.1 Running a session

In case you don't have a beamline at hand, you can fetch our sample sessions with the *fetch* command:

```
$ concert fetch --repo https://github.com/ufo-kit/concert-examples
```

Now *start* the tutorial session:

```
$ concert start tutorial
```

You will be greeted by an IPython shell loaded with pre-defined devices, processes and utilities like the pint package for unit calculation. Although, this package is primarily used for talking to devices, you can also use it to do simple calculations:

```
tutorial > a = 9.81 * q.m / q.s**2
tutorial > "Velocity after 5 seconds: {0}".format(5 * q.s * a)

'Velocity after 5 seconds: 49.05 meter / second'
```

You can get an overview of all defined devices by calling the `ddoc()` function:

```
tutorial > ddoc()

------------------------------------------------------------------------------
  Name          Description    Parameters
------------------------------------------------------------------------------
  motor         None            Name      Access   Unit   Description
                                position  rw       m      Position of the motor
------------------------------------------------------------------------------
  ...
```

Now, by typing just the name of a device, you can see it's currently set parameter values:

```
tutorial > motor

<concert.devices.motors.dummy.LinearMotor object at 0x9419f0c>
 Parameter  Value
 position   12.729455653 millimeter
```

To get an overview of all devices' parameter values, use the `dstate()` function:

```
tutorial > dstate()

---------------------------------------------
  Name          Parameters
---------------------------------------------
  motor         position  99.382 millimeter
---------------------------------------------
  ...
```

To change the value of a parameter, you simply assign a new value to it:

```
tutorial > motor.position = 2 * q.mm
```

Now, check the position to verify that the motor reached the target position:

```
tutorial > motor.position
<Quantity(2.0, 'millimeter')>
```

Depending on the device, changing a parameter will block as long as the device has not yet reached the final target state. You can read more about asynchronous execution in the *Device control* chapter.

Note: A parameter value is always checked for the correct unit and soft limit condition. If you get an error, check twice that you are using a compatible unit (setting two seconds on a motor position is obviously not) and are within the allowed parameter range.

`pdoc()` displays information about currently defined functions and processes and may look like this:

```
tutorial > pdoc()
------------------------------------------------------------------------------
Name                       Description
------------------------------------------------------------------------------
save_exposure_scan         Run an exposure scan and save the result as a NeXus
                           compliant file. This requires that libnexus and NexPy
                           are installed.
------------------------------------------------------------------------------
```

In case you are interested in the implementation of a function, you can use `code_of()`. For example:

```
tutorial > code_of(code_of)
def code_of(func):
    """Show implementation of *func*."""
    source = inspect.getsource(func)

    try:
        ...
```

Note: Because we are actually running an IPython shell, you can _always_ tab-complete objects and attributes. For example, to change the motor position, you could simply type `mo<Tab>.po<Tab> = q.me<Tab>`.

1.2.2 Creating a session

First of all, *initialize* a new session:

```
$ concert init new-session
```

and *start* the default editor with

```
$ concert edit new-session
```

At the top of the file, you can see a string enclosed in three `"`. This should changed to something descriptive as it will be shown each time you start the session.

Adding devices

To create a device suited for your experiment you have to import it first. Concert uses the following packaging scheme to separate device classes and device implementations: `concert.devices.[class].[implementation]`. Thus if you want to create a dummy ring from the storage ring class, you would add this line to your session:

```
from concert.devices.storagerings.dummy import StorageRing
```

Once imported, you can create the device and give it a name that will be accessible from the command line shell:

```
from concert.devices.motors.dummy import LinearMotor

ring = StorageRing()
motor = LinearMotor()
```

Importing other sessions

To specify experiments that share a common set of devices, you can define a base session and import it from each sub-session:

```
from base import *
```

Now everything that was defined will be present when you start up the new session.

1.3 User manual

1.3.1 Command line shell

Concert comes with a command line interface that is launched by typing `concert` into a shell. Several subcommands define the action of the tool.

Session commands

The `concert` tool is run from the command line. Without any arguments, its help is shown:

```
$ concert
usage: concert [-h] [--version]  ...

optional arguments:
  -h, --help  show this help message and exit
  --version   show program's version number and exit

Concert commands:

    start     Start a session
    init      Create a new session
    mv        Move session *source* to *target*
    log       Show session logs
    show      Show available sessions or details of a given *session*
    edit      Edit a session
    rm        Remove one or more sessions
    fetch     Import an existing *session*
```

The tool is command-driven, that means you call it with a command as its first argument. To read command-specific help, use:

```
$ concert [command] -h
```

Note: When Concert is installed system-wide, a bash completion for the `concert` tool is installed too. This means, that commands and options will be completed when pressing the `Tab` key.

init

Create a new session with the given name:

```
concert init experiment
```

If such a session already exists, Concert will warn you.

--force
Create the session even if one already exists with this name.

--imports
List of module names that are added to the import list.

Note: The location of the session files depends on the chosen installation method. If you installed into a virtual environment `venv`, the files will be stored in `/path/to/venv/share/concert`. If you have installed Concert system-wide our without using a virtual environment, it is installed into `$XDG_DATA_HOME/concert` or `$HOME/.local/share/concert` if the former is not set. See the XDG Base Directory Specification for further information. It is probably a *very* good idea to put the session directory under version control.

edit

Edit the session file by launching `$EDITOR` with the associated Python module file:

```
concert edit session-name
```

This file can contain any kind of Python code, but you will most likely just add device definitions and import processes that you want to use in a session. If the `session-name` doesn't exist it is created.

log

Show log of session:

```
concert log session-name
```

If a session is not given, the log command shows entries from all sessions.

--follow
Instead of showing the past log, update as changes come in. This is the same operation as if the log file was viewed with `tail -f`.

By default, logs are gathered in `$XDG_DATA_HOME/concert/concert.log`. To change this, you can pass the `--logto` and `--logfile` options to the `start` command. For example, if you want to output log to `stderr` use

```
concert --logto=stderr start experiment
```

or if you want to get rid of any log data use

```
concert --logto=file --logfile=/dev/null start experiment
```

show

Show all available sessions or details of a given session:

```
concert show [session-name]
```

mv

Rename a session:

```
concert mv old-session new-session
```

cp

Copy a session:

```
concert cp session session-copy
```

rm

Remove one or more sessions:

```
concert rm session-1 session-2
```

Warning: Be careful. The session file is unlinked from the file system and no backup is made.

fetch

Import an existing session from a Python file:

```
concert fetch some-session.py
```

Concert will warn you if you try to import a session with a name that already exists.

`--force`
Overwrite session if it already exists.

`--repo`
The URL denotes a Git repository from which the sessions are imported.

Warning: The server certificates are *not* verified when specifying an HTTPS connection!

start

Load the session file and launch an IPython shell:

```
concert start session-name
```

The quantities package is already loaded and named `q`.

--logto={stderr, **file**}
Specify a method for logging events. If this flag is not specified, `file` is used and assumed to be `$XDG_DATA_HOME/concert/concert.log`.

--logfile=<filename>
Specify a log file if `--logto` is set to `file`.

--loglevel={debug, **info**, **warning**, **error**, **critical**}
Specify lowest log level that is logged.

--non-interactive
Run the session as a script and do not launch a shell.

Extensions

Spyder

If Spyder is installed, start the session within the Spyder GUI.

1.3.2 Device control

Parameters

In Concert, a *device* is a software abstraction for a piece of hardware that can be controlled. Each device consists of a set of named `Parameter` instances and device-specific methods. If you know the parameter name, you can get a reference to the parameter object by using the index operator:

```
pos_parameter = motor['position']
```

To set and get parameters explicitly , you can use the `Parameter.get()` and `Parameter.set()` methods:

```
pos_parameter.set(1 * q.mm)
print (pos_parameter.get().result())
```

Both methods will return a *Future*. A future is a promise that a result will be delivered when asked for. In the mean time other things can and should happen concurrently. As you can see, to get the result of a future you call its `result()` method.

An easier way to set and get parameter values are properties via the dot-name-notation:

```
motor.position = 1 * q.mm
print (motor.position)
```

As you can see, accessing parameters this way will *always be synchronous* and *block* execution until the value is set or fetched.

Parameter objects are not only used to communicate with a device but also carry meta data information about the parameter. The most important ones are `Parameter.name`, `Parameter.unit` and

`Parameter.in_hard_limit` as well as the doc string describing the parameter. Moreover, parameters can be queried for access rights using `Parameter.is_readable()` and `Parameter.is_writable()`.

To get all parameters of an object, you can iterate over the device itself

```
for param in motor:
    print("{0} => {1}".format(param.unit, param.name))
```

Saving state

In some scenarios you would like to come back to a certain state. Let's suppose, you have a motor that you want to check if it moves. If it does, you want it to go back to the same place it came from. For these cases you can use `Device.stash()` to store the current state of a device and `Device.restore()` to go back. Because this is done in a stacked fashion, you can, for example, model local coordinate pretty easily:

```
motor.stash()

# Do movements aka modify the "local" coordinate system
motor.move(1 * q.mm)

# Go back to the original state
motor.restore()
```

Locking parameters

In case you want to prevent a parameter from being written you can use `ParameterValue.lock()`. If you specify a *permanent* parameter to be True the parameter cannot be unlocked anymore. In case you want to unlock a parameter you can use `ParameterValue.unlock()`, to get the state you can check the attribute `ParameterValue.locked`. All the parameters within a device can be locked and unlocked at once, for example one can do:

```
motor['position'].lock()
motor.position = 10 * q.mm
# Does not work, you will get a LockError
motor.position.locked
True

motor['position'].unlock()

# Works as expected
motor.position = 10 * q.mm

# Lock the whole device (all parameters)
motor.lock(permanent=True)

# This will not work anymore
motor.unlock()
# You will get a LockError
```

1.3.3 Process control

Scanning

`scan()` is used to scan a device parameter and start a feedback action. For instance, to set 10 motor positions between 5 and 12 millimeter and acquire the flow rate of a pump could be written like:

```
from concert.processes import scan

# Assume motor and pump are already defined

def get_flow_rate():
    return pump.flow_rate

x, y = scan(motor['position'], get_flow_rate,
            5*q.mm, 12*q.mm, 10).result()
```

As you can see `scan()` always yields a future that needs to be resolved when you need the result.

`ascan()` and `dscan()` are used to scan multiple parameters in a similar way as SPEC:

```
from concert.quantities import q
from concert.processes import ascan

def do_something(parameters):
    for each parameter in parameters:
        print(parameter)

ascan([(motor1['position'], 0 * q.mm, 25 * q.mm),
       (motor2['position'], -2 * q.cm, 4 * q.cm)],
       n_intervals=10, handler=do_something)
```

Focusing

To adjust the focal plane of a camera, you use `focus()` like this:

```
from concert.processes import focus
from concert.cameras.uca import Camera
from concert.motors.dummy import LinearMotor

motor = LinearMotor()
camera = Camera('mock')
focus(camera, motor)
```

1.3.4 Data processing

Coroutines

Coroutines provide a way to process data and yield execution until more data is produced. *Generators* represent the source of data and can used as normal iterators, e.g. in a `for` loop. Coroutines can use the output of a generator to either process data and output a new result item in a *filter* fashion or process the data without further results in a *sink* fashion.

Coroutines are simple functions that get their input by calling yield on the right side or as an argument. Because they need to be started in a particular way, it is useful to decorate a coroutine with the `coroutine()` decorator:

```
from concert.coroutines.base import coroutine

@coroutine
def printer():
    while True:
        item = yield
        print(item)
```

This coroutine fetches data items and prints them one by one. Because no data is produced, this coroutine falls into the sink category. Concert provides some common pre-defined sinks in the `sinks` module.

Filters hook into the data stream and process the input to produce some output. For example, to generate a stream of squared input, you would write:

```
@coroutine
def square(consumer):
    while True:
        item = yield
        consumer.send(item**2)
```

You can find a variety of pre-defined filters in the `filters` module.

Connecting data sources with coroutines

In order to connect a *generator* that `yields` data to a *filter* or a *sink* it is necessary to bootstrap the pipeline by using the `inject()` function, which forwards generated data to a coroutine:

```
from concert.coroutines.base import inject

def generator(n):
    for i in range(n):
        yield i

# Use the output of generator to feed into printer
inject(generator(5), printer())
```

To fan out a single input stream to multiple consumers, you can use the `broadcast()` like this:

```
from concert.coroutines.base import broadcast

source(5, broadcast(printer(),
                    square(printer())))
```

High-performance processing

The generators and coroutines yield execution, but if the data production should not be stalled by data consumption the coroutine should only provide data buffering and delegate the real consumption to a separate thread or process. The same can be achieved by first buffering the data and then yielding them by a generator. It comes from the fact that a generator will not produce a new value until the old one has been consumed.

High-performance computing

The `ufo` module provides classes to process data from an experiment with the UFO data processing framework. The simplest example could look like this:

```
from concert.ext.ufo import InjectProcess
from gi.repository import Ufo
import numpy as np
import scipy.misc

pm = Ufo.PluginManager()
writer = pm.get_task('writer')
writer.props.filename = 'foo-%05i.tif'

proc = InjectProcess(writer)

proc.run()
proc.push(scipy.misc.lena())
proc.join()
```

To save yourself some time, the `ufo` module provides a wrapper around the raw `UfoPluginManager`:

```
from concert.ext.ufo import PluginManager

pm = PluginManager()
writer = pm.get_task('writer', filename='foo-%05i.tif')
```

Viewing processed data

Concert has a Matplotlib integration to simplify viewing 1D time series with the `PyplotViewer` and 2D image data with the `PyplotImageViewer`:

```
from concert.devices.cameras.dummy import Camera
from concert.ext.viewers import PyplotImageViewer

# Create a camera and execute something with it in recording state
camera = Camera()
with camera.recording():
    # Create a viewer and show one frame
    viewer = PyplotImageViewer()
    viewer.show(camera.grab())
```

1.3.5 Experiments

Experiments connect data acquisition and processing. They can be run multiple times by the `base.Experiment.run()`, they take care of proper file structure and logging output.

Acquisition

Experiments consist of `Acquisition` objects which encapsulate data generator and consumers for a particular experiment part (dark fields, radiographs, ...). This way the experiments can be broken up into smaller logical pieces. A single acquisition object needs to be reproducible in order to repeat an experiment more times, thus we specify its generator and consumers as callables which return the actual generator or consumer. We need to do this because generators cannot be “restarted”. An example of an acquisition could look like this:

```
from concert.coroutines.base import coroutine
from concert.experiments import Acquisition

# This is a real generator, num_items is provided somewhere in our session
```

```
def produce():
    for i in range(num_items):
        yield i

# A simple data forwarder filter, next_consumer has to be already defined
@coroutine
def consumer():
    while True:
        item = yield
        next_consumer.send(item)

acquisition = Acquisition('foo', produce, consumer_callers=[consumer])
# Now we can run the acquisition
acquisition()
```

class `concert.experiments.base.`**`Acquisition`**(*name*, *generator_caller*, *consumer_callers=None*)

An acquisition object connects data generator to consumers.

`generator_caller`
a callable which returns a generator once called

`consumer_callers`
a list of callables which return a coroutine once started

Base

Base `base.Experiment` makes sure a directory for each run is created and logger output goes to that directory.

class `concert.experiments.base.`**`Experiment`**(*acquisitions*, *walker*, *name_fmt='scan_{:>04}'*)

Experiment base class. An experiment can be run multiple times with the output data and log stored on disk. You can prepare every run by `prepare()` and finsh the run by `finish()`. These methods do nothing by default. They can be useful e.g. if you need to reinitialize some experiment parts or want to attach some logging output.

`acquisitions`
A list of acquisitions this experiment is composed of

`walker`
A `concert.storage.Walker` stores experimental data and logging output

`name_fmt`
Since experiment can be run multiple times each iteration will have a separate entry on the disk. The entry consists of a name and a number of the current iteration, so the parameter is a formattable string.

`acquire`()
Acquire data by running the acquisitions. This is the method which implements the data acquisition and should be overriden if more functionality is required, unlike `run()`.

`acquisitions`
Acquisitions is a read-only attribute which has to be manipulated by explicit methods provided by this class.

`add`(*acquisition*)
Add *acquisition* to the acquisition list and make it accessible as attribute, e.g.:

frames = Acquisition(...) experiment.add(frames) # This is possible experiment.frames

`finish`()
Gets executed after every experiment run.

get_acquisition (*name*)
Get acquisition by its *name*. In case there are more like it, the first one is returned.

prepare ()
Gets executed before every experiment run.

remove (*acquisition*)
Remove *acquisition* from experiment.

run ()
Compute the next iteration and run the `acquire()`.

swap (*first*, *second*)
Swap acquisition *first* with *second*. If there are more occurences of either of them then the ones which are found first in the acquisitions list are swapped.

Imaging

Imaging experiments all subclass `imaging.Experiment`, which makes sure all the acquired frames are written to disk.

class `concert.experiments.imaging.`**Experiment** (*acquisitions*, *walker*, *name_fmt='scan_{:>04}'*)
Imaging experiment stores images acquired in acquisitions on disk automatically.

acquire ()
Run the experiment. Add writers to acquisitions dynamically.

A basic frame acquisition generator which triggers the camera itself is provided by `frames()`

`concert.experiments.imaging.`**frames** (*num_frames*, *camera*, *callback=None*)
A generator which takes *num_frames* using *camera*. *callback* is called after every taken frame.

There are tomography helper functions which make it easier to define the proper settings for conducting a tomographic experiment.

`concert.experiments.imaging.`**tomo_angular_step** (*frame_width*)
Get the angular step required for tomography so that every pixel of the frame rotates no more than one pixel per rotation step. *frame_width* is frame size in the direction perpendicular to the axis of rotation.

`concert.experiments.imaging.`**tomo_projections_number** (*frame_width*)
Get the minimum number of projections required by a tomographic scan in order to provide enough data points for every distance from the axis of rotation. The minimum angular step is considered to be needed smaller than one pixel in the direction perpendicular to the axis of rotation. The number of pixels in this direction is given by *frame_width*.

`concert.experiments.imaging.`**tomo_max_speed** (*frame_width*, *frame_rate*)
Get the maximum rotation speed which introduces motion blur less than one pixel. *frame_width* is the width of the frame in the direction perpendicular to the rotation and *frame_rate* defines the time required for recording one frame.

Note: frame rate is required instead of exposure time because the exposure time is usually shorter due to the camera chip readout time. We need to make sure that by the next exposure the sample hasn't moved more than one pixel from the previous frame, thus we need to take into account the whole frame taking procedure (exposure + readout).

Control

Experiment automation based on on-line data analysis.

class `concert.experiments.control.`**`ClosedLoop`**

An abstract feedback loop which acquires data, analyzes it on-line and provides feedback to the experiment. The data acquisition procedure is done iteratively until the result of some metric converges to a satisfactory value. Schematically, the class is doing the following in an iterative way:

```
initialize -> measure -> compare -> OK -> success
                ^            |
                |           NOK
                |            |
                -- control <--
```

`compare``()`

Return True if the metric is satisfied, False otherwise. This is the decision making process.

`control``()`

React on the result of a measurement.

`initialize``()`

Bring the experimental setup to some defined initial (reference) state.

`measure``()`

Conduct a measurement from data acquisition to analysis.

`run``(`*self*, *max_iterations=10*`)`

Run the loop until the metric is satisfied, if we don't converge in *max_iterations* then the run is considered unsuccessful and False is returned, otherwise True.

class `concert.experiments.control.`**`DummyLoop`**

A dummy optimization loop.

CHAPTER 2

Developer documentation

2.1 Development

2.1.1 Writing devices

Get the code

Concert is developed using Git on the popular GitHub platform. To clone the repository call:

```
$ git clone https://github.com/ufo-kit/concert
```

To get started you are encouraged to install the *development* dependencies via pip:

```
$ cd concert
$ pip install -r requirements.txt
```

After that you can simply install the development source with

```
$ make install
```

Run the tests

The core of Concert is tested using Python's standard library `unittest` module and nose. To run all tests, you can call nose directly in the root directory or run make with the `check` argument

```
$ make check
```

Some tests take a lot of time to complete and are marked with the `@slow` decorator. To skip them during regular development cycles, you can run

```
$ make check-fast
```

You are highly encouraged to add new tests when you are adding a new feature to the core or fixing a known bug.

Basic concepts

The core abstraction of Concert is a `Parameter`. A parameter has at least a name but most likely also associated setter and getter callables. Moreover, a parameter can have units and limiters associated with it.

The modules related to device creation are found here

```
concert/
|-- base.py
`-- devices
    |-- base.py
    |-- cameras
    |   |-- base.py
    |   `-- ...
    |-- __init__.py
    |-- motors
    |   |-- base.py
    |   `-- ...
    `-- storagerings
        |-- base.py
        `-- ...
```

Adding a new device

To add a new device to an existing device class (such as motor, pump, monochromator etc.), a new module has to be added to the corresponding device class package. Inside the new module, the concrete device class must then import the base class, inherit from it and implement all abstract method stubs.

Let's assume we want to add a new motor called `FancyMotor`. We first create a new module called `fancy.py` in the `concert/devices/motors` directory package. In the `fancy.py` module, we first import the base class

```
from concert.devices.motors.base import LinearMotor
```

Our motor will be a linear one, let's sub-class `LinearMotor`:

```
class FancyMotor(LinearMotor):
    """This is a docstring that can be looked up at run-time by the `ddoc`
    tool."""
```

In order to install all required parameters, we have to call the base constructor. Now, all that's left to do, is implementing the abstract methods that would raise a `AccessorNotImplementedError`:

```
def _get_position(self):
    # the returned value must have units compatible with units set in
    # the Quantity this getter implements
    return self.position

def _set_position(self, position):
    # position is guaranteed to be in the units set by the respective
    # Quantity
    self.position = position
```

We guarantee that in setters which implement a `Quantity`, like the `_set_position()` above, obtain the value in the exact same units as they were specified in the respective `Quantity` they implement. E.g. if the above `_set_position()` implemented a quantity with units set in kilometers, the `position` of the `_set_position()` will also be in kilometers. On the other hand the getters do not need to return the exact same quantity but the value must be compatible, so the above `_get_position()` could return millimeters and the user would get the value in kilometers, as defined in the respective `Quantity`.

Creating a device class

Defining a new device class involves adding a new package to the `concert/devices` directory and adding a new `base.py` class that inherits from `Device` and defines necessary `Parameter` and `Quantity` objects.

In this exercise, we will add a new pump device class. From an abstract point of view, a pump is characterized and manipulated in terms of the volumetric flow rate, e.g. how many cubic millimeters per second of a medium is desired.

First, we create a new `base.py` into the new `concert/devices/pumps` directory and import everything that we need:

```
import quantities as q
from concert.base import Quantity
from concert.devices.base import Device
```

The `Device` handles the nitty-gritty details of messaging and parameter handling, so our base pump device must inherit from it. Furthermore, we have to specify which kind of parameters we want to expose and how we get the values for the parameters (by tying them to getter and setter callables):

```
class Pump(Device):

    flow_rate = Quantity(q.m**3 / q.s,
                         lower=0 * q.m**3 / q.s, upper=1 * q.m**3 / q.s,
                         help="Flow rate of the pump")

    def __init__(self):
        super(Pump, self).__init__()
```

The *flow_rate* parameter can only receive values from zero to one cubic meter per second.

We didn't specify explicit *fget* and *fset* functions, which is why implicit setters and getters called *_set_flow_rate* and *_get_flow_rate* are installed. The real devices then need to implement these. You can however, also specify explicit setters and getters in order to hook into the get and set process:

```
class Pump(Device):

    def __init__(self):
        super(Pump, self).__init__()

    def _intercept_get_flow_rate(self):
        return self._get_flow_rate() * 10

    flow_rate = Parameter(unit=q.m**3 / q.s,
                          fget=_intercept_get_flow_rate)
```

Be aware, that in this case you have to list the parameter *after* the functions that you want to refer to.

In case you want to specify the name of the accessor function yourself and rely on implementation by subclasses, you have to raise an `AccessorNotImplementedError`:

```
class Pump(Device):

    ...

    def _set_flow_rate(self):
        raise AccessorNotImplementedError
```

State machine

A formally defined finite state machine is necessary to ensure and reason about correct behaviour. Concert provides an implicitly defined, decorator-based state machine. The machine can be used to model devices which support hardware state reading but also the ones which don't thanks to the possibility to store the state in the device itself. To use the state machine you need to declare a `State` object in the base device class and apply the `check()` decorator on each method that changes the state of a device. If you are implementing a device which can read the hardware state you need to define the `_get_state` method. If you are implementing a device which does not support hardware state reading then you need to redefine the `State` in such a way that it has a default value (see the code below) and you can ensure it is changed by respective methods by using the `transition()` decorator on such methods, so that you can keep track of state changes at least in software and comply with transitioning. Examples of such devices could look as follows:

```python
from concert.base import Quantity, State, transition, check


class BaseMotor(Device):

    """A base motor class."""

    state = State()
    position = Quantity(unit=q.m)

    @check(source='standby', target='moving')
    def start(self):
        ...

    def _start(self):
        # the actual implementation of starting something
        ...


class Motor(BaseMotor):

    """A motor with hardware state reading support."""

    ...

    def _start(self):
        # Implementation communicates with hardware
        ...

    def _get_state(self):
        # Get the state from the hardware
        ...


class StatelessMotor(BaseMotor):

    """A motor which doesn't support state reading from hardware."""

    # we have to specify a default value since we cannot get it from
    # hardware
    state = State(default='standby')

    ...

    @transition(target='moving')
    def _start(self):
        ...
```

The example above explains two devices with the same functionality, however, one supports hardware state reading and the other does not. When they want to `start` the state is checked before the method is executed and afterwards. By checking we mean the current state is checked against the one specified by `source` and the state after the execution is checked against `target`. The `Motor` represents a device which supports hardware state reading. That means all we have to do is to implement `_get_state`. The `StatelessMotor`, on the other hand, has no way of determining the hardware state, thus we need to keep track of it in software. That is achieved by the `transition()` which sets the device state after the execution of the decorated function to `target`. This way the `start` method can look the same for both devices.

Besides single state strings you can also add lists of strings and a catch-all `*` state that matches all states.

There is no explicit error handling implemented for devices which support hardware state reading but it can be easily modeled by adding error states and reset functions that transition out of them. In case the device does not support state reading and it runs into an error state all you need to do is to raise a `StateError` exception, which has a parameter `error_state`. The exception is caught by `transition()` and the `error_state` parameter is used for setting the device state.

Parameters In case changing a parameter value causes a state transition, add a `transition()` to the `Parameter` object:

```
class Motor(Device):

    state = State(default='standby')

    velocity = Parameter(unit=q.m / q.s,
                         transition(source='*', target='moving'))
```

2.1.2 Asynchronous execution

Concurrency

Every user defined function or method *must* be synchronous (blocking). To define a function as asynchronous, use the `async()` decorator:

```
from concert.async import async

@async
def synchronous_function():
    # long running operation
    return 1
```

Every asynchronous function returns a *Future* that can be used for explicit synchronization:

```
future = synchronous_function()
print(future.done())
result = future.result()
```

Every future that is returned by Concert, has an additional method `join` that will block until execution finished and raise the exception that might have been raised in the wrapped function. It will also return the future to gather the result:

```
try:
    future = synchronous_function().join()
    result = future.result()
except:
    print("synchronous_function raised an exception")
```

The asynchronous execution provided by Concert deals with concurrency. If the user wants to employ real parallelism they should make use of the multiprocessing module which provides functionality not limited by Python's global interpreter lock.

Synchronization

When using the asynchronous getters and setters of `Device` and `Parameter`, processes can not be sure if other processes or the user manipulate the device during the execution. To lock devices or specific parameters, processes can use them as context managers:

```
with motor, pump['foo']:
    motor.position = 2 * q.mm
    pump.foo = 1 * q.s
```

Inside the `with` environment, the process has exclusive access to the devices and parameters.

Disable asynchronous execution

Testing and debugging asynchronous code can be difficult at times because the real source of an error is hidden behind calls from different places. To disable asynchronous execution (but still keeping the illusion of having Futures returned), you can import `DISABLE_ASYNC` and set it to `True` *before* importing anything else from Concert.

Concert already provides a Nose plugin that adds a `--disable-async` flag to the test runner which in turn sets `DISABLE_ASYNC` to `True`.

2.1.3 Helpers

Messaging

The backbone of the local messaging system is a dispatching mechanism based on the publish-subscribe analogy. Once a dispatcher object is created, objects can `Dispatcher.subscribe()` to messages from other objects and be notified when other objects `Dispatcher.send()` a message to the dispatcher:

```
from concert.helpers import Dispatcher

def handle_message(sender):
    print("{0} send me a message".format(sender))

dispatcher = Dispatcher()

obj = {}
dispatcher.subscribe(obj, 'foo', handle_message)
dispatcher.send(obj, 'foo')
```

If not stated otherwise, users should use the global `dispatcher` for sending and receiving messages.

`concert.helpers.`**`dispatcher`**
    A global `Dispatcher` instance used by all devices.

2.1.4 Contributing

Reporting bugs

Any bugs concerning the Concert core library and script should be reported as an issue on the GitHub issue tracker.

Fixing bugs or adding features

Bug fixes and new features **must** be in pull request form. Pull request commits should consist of single logical changes and bear a clear message respecting common commit message conventions. Before the change is merged eventually it must be rebased against master.

Bug fixes must come with a unit test that will fail on the bug and pass with the fix. If an issue exists reference it in the branch name and commit message, e.g. `fix-92-remove-foo` and "Fix #92: Remove foo".

New features **must** follow PEP 8 and must be documented thoroughly.

2.2 API reference

2.2.1 Core objects

Parameters

class `concert.base.`**`Parameter`**(*fget=None*, *fset=None*, *data=None*, *check=None*, *help=None*)

A parameter with getter and setter.

Parameters are similar to normal Python properties and can additionally trigger state checks. If *fget* or *fset* is not given, you must implement the accessor functions named *_set_name* and *_get_name*:

```
from concert.base import Parameter, State

class SomeClass(object):

    state = State(default='standby')

    def actual(self):
        return 'moving'

    param = Parameter(check=check(source='standby',
                                  target=['standby', 'moving'],
                                  check=actual))

    def _set_param(self, value):
        pass

    def _get_param(self):
        pass
```

When a `Parameter` is attached to a class, you can modify it by accessing its associated `ParameterValue` with a dictionary access:

```
obj = SomeClass()
print(obj['param'])
```

fget is a callable that is called when reading the parameter. *fset* is called when the parameter is written to.

data is passed to the state check function.

check is a `check()` that changes states when a value is written to the parameter.

help is a string describing the parameter in more detail.

class `concert.base.`**`ParameterValue`**(*instance*, *parameter*)

Value object of a `Parameter`.

get (**args*, ***kwargs*)
Get concrete *value* of this object.

If *wait_on* is not None, it must be a future on which this method joins.

lock (*permanent=False*)
Lock parameter for writing. If *permament* is True the parameter cannot be unlocked anymore.

locked
Return True if the parameter is locked for writing.

restore ()
Restore the last value saved with `ParameterValue.stash()`.

If the parameter can only be read or no value has been saved, this operation does nothing.

set (**args*, ***kwargs*)
Set concrete *value* on the object.

If *wait_on* is not None, it must be a future on which this method joins.

stash (**args*, ***kwargs*)
Save the current value internally on a growing stack.

If the parameter is writable the current value is saved on a stack and to be later retrieved with `ParameterValue.restore()`.

unlock ()
Unlock parameter for writing.

wait (*value*, *sleep_time=<Quantity(0.1, 'second')>*, *timeout=None*)
Wait until the parameter value is *value*. *sleep_time* is the time to sleep between consecutive checks. *timeout* specifies the maximum waiting time.

writable
Return True if the parameter is writable.

class concert.base.**Quantity** (*unit*, *fget=None*, *fset=None*, *lower=None*, *upper=None*, *data=None*, *check=None*, *help=None*)
Bases: `concert.base.Parameter`

A `Parameter` associated with a unit.

fget, *fset*, *data*, *check* and *help* are identical to the `Parameter` constructor arguments.

unit is a Pint quantity. *lower* and *upper* denote soft limits between the `Quantity` values can lie.

class concert.base.**QuantityValue** (*instance*, *quantity*)
Bases: `concert.base.ParameterValue`

lock_limits (*permanent=False*)
Lock limits, if *permanent* is True the limits cannot be unlocker anymore.

unlock_limits ()
Unlock limits.

wait (*value*, *eps=None*, *sleep_time=<Quantity(0.1, 'second')>*, *timeout=None*)
Wait until the parameter value is *value*. *eps* is the allowed discrepancy between the actual value and *value*. *sleep_time* is the time to sleep between consecutive checks. *timeout* specifies the maximum waiting time.

Collection of parameters

class `concert.base.`**`Parameterizable`**

Collection of parameters.

For each class of type `Parameterizable`, `Parameter` can be set as class attributes

```
class Device(Parameterizable):

    def get_something(self):
        return 'something'

    something = Parameter(get_something)
```

There is a simple `Parameter` and a parameter which models a physical quantity `Quantity`.

A `Parameterizable` is iterable and returns its parameters of type `ParameterValue` or its subclasses

```
for param in device:
    print("name={}".format(param.name))
```

To access a single name parameter object, you can use the `[]` operator:

```
param = device['position']
print param.is_readable()
```

If the parameter name does not exist, a `ParameterError` is raised.

Each parameter value is accessible as a property. If a device has a position it can be read and written with:

```
param.position = 0 * q.mm
print param.position
```

`install_parameters`(*params*)

Install parameters at run-time.

params is a dictionary mapping parameter names to `Parameter` objects.

`lock`(*permanent=False*)

Lock all the parameters for writing. If *permanent* is True, the parameters cannot be unlocked anymore.

`restore`(**args*, ***kwargs*)

Restore all parameters saved with `Parameterizable.stash()`.

`stash`(**args*, ***kwargs*)

Save all writable parameters that can be restored with `Parameterizable.restore()`.

The values are stored on a stacked, hence subsequent saved states can be restored one by one.

`unlock`()

Unlock all the parameters for writing.

State machine

class `concert.base.`**`State`**(*default=None*, *fget=None*, *fset=None*, *data=None*, *check=None*, *help=None*)

Finite state machine.

Use this on a class, to keep some sort of known state. In order to enforce restrictions, you would decorate methods on the class with `check()`:

```
class SomeObject(object):

    state = State(default='standby')

    @check(source='*', target='moving')
    def move(self):
        pass
```

In case your device doesn't provide information on its state you can use the `transition()` to store the state in an instance of your device:

```
@transition(immediate='moving', target='standby')
def _set_some_param(self, param_value):
    # when the method starts device state is set to *immediate*
    # long operation goes here
    pass
    # the state is set to *target* in the end
```

Accessing the state variable will return the current state value, i.e.:

```
obj = SomeObject()
assert obj.state == 'standby'
```

The state cannot be set explicitly by:

```
obj.state = 'some_state'
```

but the object needs to provide methods which transition out of states, the same holds for transitioning out of error states. If the `_get_state()` method is implemented in the device it is always used to get the state, otherwise the state is stored in software.

`concert.base.`**`check`**`(`*source='*', target=None*`)`

Decorates a method for checking the device state.

source denotes the source state that must be present at the time of invoking the decorated method. *target* is the state that the state object will be after successful completion of the method or a list of possible target states.

`concert.base.`**`transition`**`(`*immediate=None, target=None*`)`

Change software state of a device to *immediate*. After the function execution finishes change the state to *target*.

Devices

class `concert.devices.base.`**`Device`**

Bases: `concert.base.Parameterizable`

A `Device` provides locked access to a real-world device.

It implements the context protocol to provide locking:

```
with device:
    # device is locked
    device.parameter = 1 * q.m
    ...

# device is unlocked again
```

Asynchronous execution

exception `concert.async.`**`KillException`**

Exception that may be thrown during the execution of an `async()` decorated function. The function may run cleanup code.

`concert.async.`**`async`**`()`

A decorator for functions which are executed asynchronously.

`concert.async.`**`threaded`**`()`

Threaded execution of a function *func*.

class `concert.async.`**`Dispatcher`**

Core dispatcher

`send``(`*sender*, *message*`)`

Send message from sender.

`subscribe``(`*sender*, *message*, *handler*`)`

Subscribe to a message sent by sender.

When message is sent by sender, handler is called with sender as the only argument.

`unsubscribe``(`*sender*, *message*, *handler*`)`

Remove *handler* from the subscribers to *(sender, message)*.

`concert.async.`**`resolve`**`(`*result*`)`

Return a list of tuples *(x, y, ...)* from a process that returns a list of futures each returning a single tuple *(x, y, ...)*.

`concert.async.`**`wait`**`(`*futures*`)`

Wait for the list of *futures* to finish and raise exceptions if happened.

Exceptions

class `concert.base.`**`UnitError`**

Raised when an operation is passed value with an incompatible unit.

class `concert.base.`**`LimitError`**

Raised when an operation is passed a value that exceeds a limit.

class `concert.base.`**`ParameterError`**`(`*parameter*`)`

Raised when a parameter is accessed that does not exists.

class `concert.base.`**`AccessorNotImplementedError`**

Raised when a setter or getter is not implemented.

class `concert.base.`**`ReadAccessError`**`(`*parameter*`)`

Raised when user tries to change a parameter that cannot be written.

class `concert.base.`**`WriteAccessError`**`(`*parameter*`)`

Raised when user tries to read a parameter that cannot be read.

class `concert.base.`**`StateError`**`(`*error_state*, *msg=None*`)`

Raised in state check functions of devices.

Configuration

`concert.config.`**`DISABLE_ASYNC`**

Disable asynchronous execution by returning a dummy future which is not executed synchronously.

`concert.config.`**`DISABLE_GEVENT`**
Turn of gevent support and fall back to ThreadPoolExecutor approach.

2.2.2 Sessions

`concert.session.utils.`**`code_of`**`(`*func*`)`
Show implementation of *func*.

`concert.session.utils.`**`ddoc`**`()`
Render device documentation.

`concert.session.utils.`**`dstate`**`()`
Render device state in a table.

`concert.session.utils.`**`get_default_table`**`(`*field_names*, *widths=None*`)`
Return a prettytable styled for use in the shell. *field_names* is a list of table header strings.

`concert.session.utils.`**`pdoc`**`(`*hide_blacklisted=True*`)`
Render process documentation.

2.2.3 Networking

Networking package facilitates all network connections, e.g. sockets and Tango.

Socket Connections

class `concert.networking.base.`**`SocketConnection`**`(`*host*, *port*, *return_sequence='n'*`)`
A two-way socket connection. *return_sequence* is a string appended after every command indicating the end of it, the default value is a newline (n).

`execute``(`*data*`)`
Execute command and wait for response (thread safe).

`recv``()`
Read data from the socket. The result is first stripped from the trailing return sequence characters and then returned.

`send``(`*data*`)`
Send *data* to the peer. The return sequence characters are appended to the data before it is sent.

class `concert.networking.aerotech.`**`Connection`**`(`*host*, *port*`)`
Aerotech socket connection.

`recv``()`
Return properly interpreted answer from the controller.

TANGO

Tango devices are interfaced by PyTango, one can obtain the DeviceProxy by the `get_tango_device()` function.

`concert.networking.base.`**`get_tango_device`**`(`*uri*, *peer=None*`)`
Get a Tango device by specifying its *uri*. If *peer* is given change the tango_host specifying which database to connect to. Format is host:port as a string.

2.2.4 Helpers

class `concert.helpers.`**`Bunch`**(*values*)

Encapsulate a list or dictionary to provide attribute-like access.

Common use cases look like this:

```
d = {'foo': 123, 'bar': 'baz'}
b = Bunch(d)
print(b.foo)
>>> 123

l = ['foo', 'bar']
b = Bunch(l)
print(b.foo)
>>> 'foo'
```

class `concert.helpers.`**`Command`**(*name*, *opts*)

Command class for the CLI script

Command objects are loaded at run-time and injected into Concert's command parser.

name denotes the name of the sub-command parser, e.g. "mv" for the MoveCommand. *opts* must be an argparse-compatible dictionary of command options.

`run`(**args*, ***kwargs*)

Run the command

exception `concert.helpers.`**`WaitError`**

Raised on busy waiting timeouts

`concert.helpers.`**`busy_wait`**(*condition*, *sleep_time=<Quantity(0.1, 'second')>*, *timeout=None*)

Busy wait until a callable *condition* returns True. *sleep_time* is the time to sleep between consecutive checks of *condition*. If *timeout* is given and the *condition* doesn't return True within the time specified by it a `WaitingError` is raised.

class `concert.helpers.`**`expects`**(**args*, ***kwargs*)

Decorator which determines expected arguments for the function and also check correctness of given arguments. If input arguments differ from expected ones, exception *TypeError* will be raised.

For numeric arguments use *Numeric* class with 2 parameters: dimension of the array and units (optional). E.g. "Numeric (1)" means function expects one number or "Numeric (2, q.mm)" means function expects expression like [4,5]*q.mm

Common use case looks like this:

@expects (Camera, LinearMotor, pixelsize = Numeric(2, q.mm)) def foo(camera, motor, pixelsize = None):

pass

`concert.helpers.`**`memoize`**(*func*)

Memoize the result of *func*.

Remember the result of *func* depending on its arguments. Note, that this requires that the function is free from any side effects, e.g. returns the same value given the same arguments.

2.2.5 Device classes

Cameras

class `concert.devices.cameras.base.`**`Camera`**

Bases: `concert.devices.base.Device`

Base class for remotely controllable cameras.

`frame-rate`
Frame rate of acquisition in q.count per time unit.

`grab``()`
Return a NumPy array with data of the current frame.

`recording``(`**args*, ***kwds*`)`
A context manager for starting and stopping the camera.

In general it is used with the `with` keyword like this:

```
with camera.recording():
    frame = camera.grab()
```

`start_recording``(`*instance*, **args*, ***kwargs*`)`
Start recording frames.

`stop_recording``(`*instance*, **args*, ***kwargs*`)`
Stop recording frames.

`stream``(`**args*, ***kwargs*`)`
Grab frames continuously and send them to *consumer*, which is a coroutine.

`trigger``()`
Trigger a frame if possible.

I/O

class `concert.devices.io.base.`**`IO`**

Bases: `concert.devices.base.Device`

The IO device consists of ports which can be readable, writable or both.

`ports`
Port IDs used by `read_port()` and `write_port()`

`read_port``(`*port*`)`
Read a *port*.

`write_port``(`*port*, *value*`)`
Write a *value* to the *port*.

Lightsources

class `concert.devices.lightsources.base.`**`LightSource`**

Bases: `concert.devices.base.Device`

A base LightSource class.

Monochromators

class `concert.devices.monochromators.base.`**`Monochromator`**

Bases: `concert.devices.base.Device`

Monochromator device which is used to filter the beam in order to get a very narrow energy bandwidth.

`energy`

Monochromatic energy in electron volts.

`wavelength`

Monochromatic wavelength in meters.

Motors

Linear

Linear motors are characterized by moving along a straight line.

class `concert.devices.motors.base.`**`LinearMotor`**

Bases: `concert.devices.motors.base._PositionMixin`

One-dimensional linear motor.

`position`

Position of the motor in length units.

class `concert.devices.motors.base.`**`ContinuousLinearMotor`**

Bases: `concert.devices.motors.base.LinearMotor`

One-dimensional linear motor with adjustable velocity.

`velocity`

Current velocity in length per time unit.

Rotational

Rotational motors are characterized by rotating around an axis.

class `concert.devices.motors.base.`**`RotationMotor`**

Bases: `concert.devices.motors.base._PositionMixin`

One-dimensional rotational motor.

`position`

Position of the motor in angular units.

class `concert.devices.motors.base.`**`ContinuousRotationMotor`**

Bases: `concert.devices.motors.base.RotationMotor`

One-dimensional rotational motor with adjustable velocity.

`velocity`

Current velocity in angle per time unit.

Axes

An axis is a coordinate system axis which can realize either translation or rotation, depending by which type of motor it is realized.

class `concert.devices.positioners.base.`**`Axis`**(*coordinate*, *motor*, *direction=1*, *position=None*)

Bases: `object`

An axis represents a Euclidean axis along which one can translate or around which one can rotate. The axis *coordinate* is a string representing the Euclidean axis, i.e. 'x' or 'y' or 'z'. Movement is realized by a *motor*. An additional *position* argument is necessary for calculatin more complicated motion types, e.g. rotation around arbitrary point in space. It is the local position with respect to a `concert.devices.positioners.base.Positioner` in which it is placed.

`get_position`()

Get position asynchronously with respect to axis direction.

`set_position`(*position*)

Set the *position* asynchronously with respect to axis direction.

Photodiodes

Photodiodes measure light intensity.

class `concert.devices.photodiodes.base.`**`PhotoDiode`**

Bases: `concert.devices.base.Device`

Impementation of photo diode with V output signal

Positioners

Positioner is a device consisting of more `concert.devices.positioners.base.Axis` instances which make it possible to specify a 3D position and orientation of some object.

class `concert.devices.positioners.base.`**`Positioner`**(*axes*, *position=None*)

Bases: `concert.devices.base.Device`

Combines more motors which move to form a complex motion. *axes* is a list of `Axis` instances. *position* is a 3D vector of coordinates specifying the global position of the positioner.

If a certain coordinate in the positioner is missing, then when we set the position or orientation we can specify the respective vector position to be zero or numpy.nan.

`back`(*value*)

Move back by *value*.

`down`(*value*)

Move down by *value*.

`forward`(*value*)

Move forward by *value*.

`left`(*value*)

Move left by *value*.

`move`(*position*)

Move by specified *position*.

`right`(*value*)

Move right by *value*.

rotate(*angles*)
Rotate by *angles*.

up(*value*)
Move up by *value*.

Imaging Positioners

Imaging positioner is a positioner capable of moving in *x* and *y* directions by the given amount of pixels.

class concert.devices.positioners.imaging.**Positioner**(*axes*, *detector*, *position=None*)
Bases: concert.devices.positioners.base.Positioner

A positioner which takes into account a detector with some pixel size. This way the user can specify the movement in pixels.

move(*position*)
Move by specified *position* which can be given in meters or pixels.

Pumps

class concert.devices.pumps.base.**Pump**
Bases: concert.devices.base.Device

A pumping device.

start()
Start pumping.

stop()
Stop pumping.

Scales

class concert.devices.scales.base.**Scales**
Bases: concert.devices.base.Device

Base scales class.

class concert.devices.scales.base.**TarableScales**
Bases: concert.devices.scales.base.Scales

Scales which can be tared.

tare(**args*, ***kwargs*)
Tare the scales.

Shutters

class concert.devices.shutters.base.**Shutter**
Bases: concert.devices.base.Device

Shutter device class implementation.

close()
Close the shutter.

open()
Open the shutter.

Storage rings

class concert.devices.storagerings.base.**StorageRing**
Bases: `concert.devices.base.Device`

Read-only access to storage ring information.

current
Ring current

energy
Ring energy

lifetime
Ring lifetime in hours

2.2.6 Processes

Scanning

concert.processes.**scan**(*param*, *feedback*, *minimum=None*, *maximum=None*, *intervals=64*, *convert=<function <lambda> at 0x7f134d67f848>*)
Scan the parameter object in *intervals* steps between *minimum* and *maximum* and call *feedback* at each step. If *minimum* or *maximum* is `None`, `ParameterValue.lower` or `ParameterValue.upper` is used.

Set *convert* to a callable that transforms the parameter value prior to setting it.

Generates futures which resolve to tuples containing the set and feedback values *(x, y)*.

concert.processes.**ascan**(*param_list*, *n_intervals*, *handler*, *initial_values=None*)
For each of the *n_intervals* and for each of the *(parameter, start, stop)* tuples in *param_list*, calculate a set value from *(stop - start) / n_intervals* and set *parameter* to it:

```
ascan([(motor['position'], 0 * q.mm, 2 * q.mm)], 5, handler)
```

When all devices have reached the set point *handler* is called with a list of the parameters as its first argument.

If *initial_values* is given, it must be a list with the same length as *devices* containing start values from where each device is scanned.

concert.processes.**dscan**(*parameter_list*, *n_intervals*, *handler*)
For each of the *n_intervals* and for each of the *(parameter, start, stop)* tuples in *param_list*, calculate a set value from *(stop - start) / n_intervals* and set *parameter*.

concert.processes.**scan_param_feedback**(*scan_param*, *feedback_param*, *minimum=None*, *maximum=None*, *intervals=64*, *convert=<function <lambda> at 0x7f134d67f938>*)
Convenience function to scan one parameter and measure another.

Scan the *scan_param* object and measure *feedback_param* at each of the *intervals* steps between *minimum* and *maximum*.

Returns a tuple *(x, y)* with scanned parameter and measured values.

Focusing

`concert.processes.`**`focus`**`()`

Alignment

`concert.processes.`**`align_rotation_axis`**`(`**args*, ***kwargs*`)`

`concert.processes.`**`center_to_beam`**`(`**args*, ***kwargs*`)`

Tries to center the camera *cam* to the beam by moving with the motors *xmotor* and *zmotor*. It starts by searching the beam inside the search-area defined by *xborder* and *zborder*. Argument *pixelsize* is needed to convert pixelcoordinates into realworld-coordinates of the motors. Exceptions are raised on fail.

Optional arguments *xstep*, *zstep*, *thres*, *tolerance* and *max_iterations* are passed to the functions 'find_beam(...)' and 'center2beam(...)'.

`concert.processes.`**`drift_to_beam`**`(`*cam*, *xmotor*, *zmotor*, *pixelsize*, *tolerance=5*, *max_iterations=100*`)`

Moves the camera *cam* with motors *xmotor* and *zmotor* until the center of mass is nearer than *tolerance*-pixels to the center of the frame or *max_iterations* is reached.

To convert pixelcoordinates to realworld-coordinates of the motors the *pixelsize* (scalar or 2-element array-like, e.g. [4*q.um, 5*q.um]) is needed.

`concert.processes.`**`find_beam`**`()`

2.2.7 Coroutines

`concert.coroutines.base.`**`broadcast`**`(`**consumers*`)`

Forward data to all *consumers*.

`concert.coroutines.base.`**`coroutine`**`(`*func*`)`

Start a coroutine automatically without the need to call next() or send(None) first.

`concert.coroutines.base.`**`inject`**`(`*generator*, *consumer*`)`

Let a *generator* produce a value and forward it to *consumer*.

Sinks

class `concert.coroutines.sinks.`**`Accumulate`**

Accumulate items in a list.

class `concert.coroutines.sinks.`**`Result`**

The object is callable and when called it becomes a coroutine which accepts items and stores them in a variable which allows the user to obtain the last stored item at any time point.

`concert.coroutines.sinks.`**`null`**`()`

A black-hole.

Filters

class `concert.coroutines.filters.`**`PickSlice`**`(`*index*`)`

Pick a slice from a 3D volume.

class `concert.coroutines.filters.`**`Timer`**

Timer object measures execution times of coroutine-based workflows. It measures the time from when this object receives data until all the subsequent stages finish, e.g.:

```
acquire(timer(process()))
```

would measure only the time of *process*, no matter how complicated it is and whether it invokes subsequent coroutines. Everything what happens in *process* is taken into account. This timer does not treat asynchronous operations in a special way, i.e. if you use it like this:

```
def long_but_async_operation():
    @async
    def process(data):
        long_op(data)

    while True:
        item = yield
        process(item)

timer(long_but_async_operation())
```

the time you truly measure is only the time to forward the data to *long_but_async_operation* and the time to *start* the asynchronous operation (e.g. spawning a thread).

`duration`
All iterations summed up.

`mean`
Mean iteration execution time.

`reset``()`
Reset the timer.

`concert.coroutines.filters.`**`absorptivity`**`(`*consumer*`)`
Get the absorptivity from a flat corrected stream of images. The intensity after the object is defined as $I = I_0 \cdot e^{-\mu t}$ and we extract the absorptivity μt from the stream of flat corrected images I/I_0.

`concert.coroutines.filters.`**`average_images`**`(`*consumer*`)`
Average images as they come and send them to *consumer*.

`concert.coroutines.filters.`**`backproject`**`(`*center*, *consumer*`)`
Filtered backprojection filter. The filter receives a sinogram, filters it and based on *center* of rotation it backprojects it. The slice is then sent to *consumer*.

`concert.coroutines.filters.`**`downsize`**`(`*consumer*, *x_slice=None*, *y_slice=None*, *z_slice=None*`)`
Downsize images in 3D and send them to *consumer*. Every argument is either a tuple (start, stop, step). *x_slice* operates on image width, *y_slice* on its height and *z_slice* on the incoming images, i.e. it creates the third time dimension.

Note: the *start* index is included in the data and the *stop* index is excluded.

`concert.coroutines.filters.`**`flat_correct`**`(`*flat*, *consumer*, *dark=None*`)`
Flat correcting corounte, which takes a *flat* field, a *dark* field (if given), calculates a flat corrected radiograph and forwards it to *consumer*.

`concert.coroutines.filters.`**`queue`**`(`*consumer*`)`
Store the incoming data in a queue and dispatch in a separate thread which prevents the stalling on the "main" data stream.

`concert.coroutines.filters.`**`sinograms`**`(`*num_radiographs*, *consumer*, *sinograms_volume=None*`)`
Convert *num_radiographs* into sinograms and send them to *consumer*. The sinograms are sent every time a

new radiograph arrives. If there is more than *num_radiographs* radiographs, the sinograms are rewritten in a ring-buffer fashion. If *sinograms_volume* is given, it must be a 3D array and it is used to store the sinograms.

`concert.coroutines.filters.`**`stall`**(*consumer*, *per_shot=10*, *flush_at=None*)
Send items once enough is collected. Collect *per_shot* items and send them to *consumer*. The incoming data might represent a collection of some kind. If the last item is supposed to be sent regardless the current number of collected items, use *flush_at* by which you specify the collection size and every time the current item *counter % flush_at* == 0 the item is sent.

2.2.8 Optimization

Optimization is a procedure to iteratively find the best possible match to

$$y = f(x).$$

This module provides execution routines and algorithms for optimization.

`concert.optimization.`**`bfgs`**(*function*, *x_0*, ***kwargs*)
Broyde-Fletcher-Goldfarb-Shanno (BFGS) algorithm from `scipy.optimize.fmin_bfgs()`. Please refer to the scipy function for additional arguments information.

`concert.optimization.`**`down_hill`**(*function*, *x_0*, ***kwargs*)
Downhill simplex algorithm from `scipy.optimize.fmin()`. Please refer to the scipy function for additional arguments information.

`concert.optimization.`**`halver`**(*function*, *x_0*, *initial_step=None*, *epsilon=None*, *max_iterations=100*)
Halving the interval, evaluate *function* based on *param*. Use *initial_step*, *epsilon* precision and *max_iterations*.

`concert.optimization.`**`least_squares`**(*function*, *x_0*, ***kwargs*)
Least squares algorithm from `scipy.optimize.leastsq()`. Please refer to the scipy function for additional arguments information.

`concert.optimization.`**`nonlinear_conjugate`**(*function*, *x_0*, ***kwargs*)
Nonlinear conjugate gradient algorithm from `scipy.optimize.fmin_cg()`. Please refer to the scipy function for additional arguments information.

`concert.optimization.`**`optimize`**(**args*, ***kwargs*)
Optimize y = *function* (x), where *x_0* is the initial guess. *algorithm* is the optimization algorithm to be used:

```
algorithm(x_0, *alg_args, **alg_kwargs)
```

consumer receives all the (x, y) values as they are obtained.

`concert.optimization.`**`optimize_parameter`**(*parameter*, *feedback*, *x_0*, *algorithm*, *alg_args=()*, *alg_kwargs=None*, *consumer=None*)
Optimize *parameter* and use the *feedback* (a callable) as a result. Other arguments are the same as by `optimize()`. The function to be optimized is determined as follows:

```
parameter.set(x)
y = feedback()
```

consumer is the same as by `optimize()`.

`concert.optimization.`**`powell`**(*function*, *x_0*, ***kwargs*)
Powell's algorithm from `scipy.optimize.fmin_powell()`. Please refer to the scipy function for additional arguments information.

2.2.9 Extensions

Concert integrates third-party software in the `ext` package. Because the dependencies of these modules are not listed as Concert dependencies, you have to make sure, that the appropriate libraries and modules are installed.

UFO Processing

Base objects

class `concert.ext.ufo.`**`PluginManager`**
Plugin manager that initializes new tasks.

`get_task`(*name*, ***kwargs*)
Create a new task from plugin *name* and initialize with *kwargs*.

class `concert.ext.ufo.`**`InjectProcess`**(*graph*, *get_output=False*)
Process to inject NumPy data into a UFO processing graph.

`InjectProcess` can also be used as a context manager, in which case it will call `start()` on entering the manager and `wait()` on exiting it.

graph must either be a Ufo.TaskGraph or a Ufo.TaskNode object. If it is a graph the input tasks will be connected to the roots, otherwise a new graph will be created.

`insert`(*array*, *node=None*, *index=0*)
Insert *array* into the *node*'s *index* input.

Note: *array* must be a NumPy compatible array.

`start`()
Run the processing in a new thread.

Use `push()` to insert data into the processing chaing and `wait()` to wait until processing has finished.

`wait`()
Wait until processing has finished.

Coroutines

class `concert.ext.ufo.`**`Backproject`**(*axis_pos=None*)
Bases: `concert.ext.ufo.InjectProcess`

Coroutine to reconstruct slices from sinograms using filtered backprojection.

axis_pos specifies the center of rotation in pixels within the sinogram. If not specified, the center of the image is assumed to be the center of rotation.

Viewers

Opening images in external programs.

class `concert.ext.viewers.`**`PyplotImageViewer`**(*imshow_kwargs=None*, *colorbar=True*, *title=''*)
Dynamic image viewer using matplotlib.

show(*item*, *force=False*)
show *item* into the redrawing queue. The item is truly inserted only if the queue is empty in order to guarantee that the newest image is drawn or if the *force* is True.

class concert.ext.viewers.**PyplotViewer**(*style='o'*, *plot_kwargs=None*, *autoscale=True*, *title=''*)
Dynamic plot viewer using matplotlib.

style
One of matplotlib's linestyle format strings

plt_kwargs
Keyword arguments accepted by matplotlib's plot()

autoscale
If True, the axes limits will be expanded as needed by the new data, otherwise the user needs to rescale the axes

clear()
Clear the plotted data.

plot(*x*, *y=None*, *force=False*)
Plot *x* and *y*, if *y* is None and *x* is a scalar the real y is given by *x* and x is the current iteration of the plotting command, if *x* is an iterable then it is interpreted as y data array and x is a span [0, len(x)]. If both *x* and *y* are given, they are plotted as they are. If *force* is True the plotting is guaranteed, otherwise it might be skipped for the sake of plotting speed.

Note: if x is not given, the iteration starts at 0.

class concert.ext.viewers.**PyplotViewerBase**(*view_function*, *blit=False*)
A base class for data viewer which sends commands to a matplotlib updater which runs in a separate process.

view_function
The function which updates the figure based on the changed data. Its nomenclature has to be:

```
foo(data, force=False)
```

Where *force* determines whether the redrawing must be done or not. If it is False, the redrawing takes place if the data queue contains only the current data item. This prevents the actual drawer from being overwhelmed by the amount of incoming data.

blit
True if faster redrawing based on canvas blitting should be used.

pause()
Pause, no images are dispayed but image commands work.

resume()
Resume the viewer.

terminate()
Close all communication and terminate child process.

concert.ext.viewers.**imagej**(**args*, ***kwargs*)
Open *image* in ImageJ found by *path*. *writer* specifies the written image file type.

CHAPTER 3

Additional notes

3.1 Changelog

Here you can see the full list of changes between each Concert release.

3.1.1 Version 0.9

Released on August 15th 2014.

Improvements

- The state machine mechanism is not special anymore but directly inherits from *Parameter*.
- Added walker mechanism to write sequence data in hierarchical structures such as directories or HDF5 files.
- The long-standing gevent integration with IPython is finished at least for IPython >= 2.0.
- Added *@expects* decorator to annotate what a function can receive.
- Added *async.resolve()* to get result of future lists.
- Added *accumulate* sink and *timer* coroutines.
- Added *Timestamp* class for PCO cameras that decodes the BCD timestamp embedded in a frame.
- Added optional *wait_on* to getter and setter of a *ParameterValue*.
- We now raise an exception in if a uca frame is not available.
- Experiments have now hooks for preparation and cleanup tasks.
- Added basic control loop classes.
- Add binary signal device class.

API breaks

- *scan* yields futures instead of returning a list
- Moved specific pco cameras to *concert.devices.cameras.pco*.
- Moved *write_images* to *concert.storage*
- Removed *base.MultiContext* and *base.Process*

Fixes

- #198, #254, #271, #277, #280, #286, #293
- The pint dependency had to be raised to 0.5.2 in order to compute sums of quantities.

3.1.2 Version 0.8

Released on April 16th 2014.

Improvements

- `concert log` can now `--follow` the current operation.
- Soft limits and parameters can be locked both temporarily and permanently.
- Added new `@quantity` decorator for simple cases.
- The `concert`` binary can now be started without a session.
- Added cross-correlation tomographic axis finding.
- Added frame consumer to align_rotation_axis.
- Simplify file camera and allow resetting it
- Added ports property to the base IO device.
- Added Photodiode base device class.
- Added Fiber-Lite halogen lightsource.
- Added LEDs connected within the wago.
- Added stream coroutine to cameras.
- Added EdmundOptics photodiode.
- Added PCO.4000 camera.
- Added Wago input/output device.

API breaks

- Raise CameraError instead of ValueError
- Change Pco's freerun to stream

Fixes

- Fix FileCamera pixel units in grab
- Import GLib.GError correctly
- Make recording context exception-safe
- Fix quantity problem with recent Pint versions
- #200, #203, #206, #209, #228, #230, #245

3.1.3 Version 0.7

Released on February 17th 2014.

Improvements

- Added beam finding and centering
- `threaded` decorator uses daemonic threads
- Added `downsize`, `queue`, `stall`, `PickSlice` to coroutine filters
- Added reconstruction of the whole volume using UFO Framework
- Documentation was restructured significantly (split to usage/API)
- Added tomography helper functions
- Crio motor support continuous rotation
- `PyplotViewer` can be configured for faster drawing capabilities using `blit`
- Added dummy `Scales`
- Tests cover all devices (at least try to instantiate them)
- Added pixel units, `q.pixel` (shorthand `q.px`)
- Changed prompt color to terminal default
- Added `Positioner` device
- Added `Detector` device

API Breaks

- Finite state machine was reworked significantly
- Motors were cleaned from mixins and hard-limit was incorporated into them
- recording() context was added to cameras
- `backprojector` coroutine filter was significantly simplified
- `average_images` arguments changed
- Experiments were completely restructured based on usage of `Acquisition`
- `PyplotViewer` plotting signature changed
- Remove leftover beam line specific shutters
- Many getters/setters were replaced by properties, especially in the `concert.ext.viewers` module
- Appropriate `get_` `set_` functions were replaced by non-prefixed ones

Fixes

- #118, #128, #132, #133, #139, #148, #149, #150, #157, #159, #165, #169, #173, #174, #175, #176, #178, #179, #181, #184, #189, #192

3.1.4 Version 0.6

Released on December 10th 2013.

Improvements

- Concert now comes with an experimental gevent backend that will eventually replace the thread pool executor based asynchronous infrastructure.
- Each device can now have an explicit `State` object and `@transition` applied to function which will change the state depending on the successful outcome of the decorated function.
- 1D data plotting is implemented as `PyplotCurveViewer`.
- The `concert` binary now knows the `cp` command to make a copy of a session. The `start` command can receive a log level and with the `--non-interactive` option run a session as a script.
- Devices and parameters can store their current parameter values with `stash` and restore them later with `restore`.
- Changed the IPython prompt.
- Added the NewPort 74000 Monochromator.
- Provide a `require` function that will scream when the required Concert version is not installed.

API breaks

- `Motor` is renamed to `LinearMotor` for all devices.
- `Parameter` objects are now declared at class-level instead of at run-time within the class constructor.
- `concert.storage.create_folder` renamed to `concert.storage.create_directory`
- `concert.ext.viewers.PyplotViewer` substituted by 1D and 2D viewers `concert.ext.viewers.PyplotCurveViewer` and `concert.ext.viewers.PyplotImageViewer`
- To wait on a Future you have to call `.join` instead of `.wait`.
- Coroutine functions and decorators moved to `concert.coroutines[.base]`, asynchronous functions and decorators moved to `concert.async`.
- Removed `is_async`
- Configuration moved to `concert.config`
- Method names of `concert.ext.ufo.InjectProcess` changed.

Fixes

- #168, #166, #152, #147, #158, #150, #157, #95, #138
- Many more concerning the camera implementation.

3.1.5 Version 0.5

Released on October 31st 2013.

Improvements

- Python 3 is supported and can be tested with tox.
- Most imports are delayed in the concert binary to reduce startup time.
- We do not depend on Logbook anymore but use Python's logging module.
- Experiments can now be modelled with the `concert.experiments` module.
- `concert.ext.viewers.PyplotViewer` can be used to show 2D image data.
- Spyder command plugin is now available. That means if you have Spyder installed you can control Concert from an IDE instead of from IPython.
- Tests were restructured for easier access.

API breaks

- `concert.connections` package moved to `concert.networking` module
- Renamed `concert.helpers.multicast` to `broadcast` to reflect its true purpose.
- Session helpers such as `dstate` and `ddoc` have been moved to `concert.session.utils`.
- Frames grabbed with the libuca devices will return a copy instead of the same buffer.

Fixes:

- #106, #113 and many more which did not deserve an issue number.

3.1.6 Version 0.4

Released on October 7th 2013.

Improvements

- Tests and rotation axis alignment is faster now.
- Soft limits were added to the parameter (accessible with `.lower` and `.upper`)
- Cleaner inet connection implemention.
- Base pumps and scales were added.
- Concert no longer depends on testfixtures for running tests.
- Started work on flexible data processing schemes for light computation based on a coroutine approach.
- Integrated tifffile.py in case libtiff is not available.
- `concert mv` renames sessions.
- `@threaded` decorator can be used to run a function in its own thread.
- `Scanner` parameters can now be set in the constructor.
- Parameters can now be locked independently of the parent device. However, if done so, no one else can lock the device.
- Add `code_of` function to show the source of a function.
- Introduced coroutine based data processing facility.

API breaks

- Renamed `to_steps` to `to_device` and do not drop units
- `camera.grab` returns *None* if no data is available
- `uca.Camera` exposes the wrapped GObject camera as an attribute called `uca` instead of `camera`.
- `minimum`, `maximum` and `intervals` are now longer implemented as `Parameter` objects of `Scanner` but simple attributes.
- `asynchronous` module content has been moved to `helpers`
- Removed `Scanner` class in favor of `scan` function.

Fixes:

- Integration with all IPython releases works again.
- runtests.py returns 0 on success.
- #19, #55, #71, #78, #79

3.1.7 Version 0.3

Released on August 19th 2013.

Note: This release breaks Python 2.6 compatibility!

- `Calibration` classes moved to `concert.devices.calibration`
- Remove `concert.processes.focus` and reorganize `concert.optimization` package, the focusing can be implemented by Maximizer with a proper feedback.
- Add `--repo` parameter to the `fetch` command. With this flag, session files version controlled with Git can be imported.
- Use pint instead of quantities. pint is faster for smaller Numpy arrays, stricter and does not depend on Numpy.
- Things can now run serialized if `concert.asynchronous.DISABLE` is set to `True`.
- Restructured tests into separate directories.
- Fix PDF generation of the docs.
- Fix problem with IPython version >= 0.10.

3.1.8 Version 0.2

Released on July 14th 2013.

- Move third-party code to `concert.ext`. For example `get_tomo_scan_result` must be imported from `concert.ext.nexus`.
- Adds `concert fetch` to pull session files from remote locations.
- Code cleanup

3.1.9 Version 0.1.1

Bug fix release, released on May 25th 2013

- Fixes Python 3 support.
- Monochromator fix.

3.1.10 Version 0.1

First public release.