ska-tango-testing

Release 0.5.2

Drew Devereux <drew.devereux@csiro.au>

Jun 08, 2023

CONTENTS:

This project provides test harness elements for testing of Tango devices within the Square Kilometre Array.

CHAPTER

ONE

USER GUIDE

The *ska_tango_testing* package provides test harness elements for SKA Tango devices.

1.1 Tango test contexts

A test context provides a consistent interface to different deployments of Tango. This allows you to write your test against that interface, and know that it will work the same, regardless of whether you are testing in a lightweight context or against a fully deployed Tango system.

The advantage of this is, you do not need to develop tests against a full Tango deployment, which is very slow to develop against, because it needs to be reset after every test run. Instead, you can develop your tests against a lightweight Tango test context, with a much faster development cycle. Once test development is complete, and all tests are passing in that lightweight Tango test context, the tests can be run against a full Tango deployment, *without changing the tests*.

Two context managers are provided:

- *TrueTangoContextManager* supports testing against a full Tango system that has already been deployed. Because Tango is assumed already to be fully deployed, there is little for this manager to do.

```
with TrueTangoContextManager() as context:
    signal_generator = context.get_device("lab/signalgenerator/1")
    spectrum_analyser = context.get_device("lab/spectrumanalyser/1")

    # test your devices...
```

- *ThreadedTestTangoContextManager* supports testing in a lightweight Tango test context, using threads for asynchrony. (Using threads instead of processes allows tests to make use of testing strategies that assume shared memory, such as mocks, patches and dependency injection.)

  Because this context manager has to launch the devices under test in a lightweight Tango test context before the tests can be run, we need to tell it about the devices that it should deploy. For each device, we must provide the device name, the device class, and any device properties. This is done with the *add_device()* method. This method must be called *before* the *with* syntax is used to enter the context.

```
context_manager = ThreadedTestTangoContextManager()
context_manager.add_device(
    "lab/signalgenerator/1",
    SignalGeneratorDevice,
    Host="siggen.lab.example.com",
    Port=5024,
)
context_manager.add_device(
```

(continues on next page)

(continued from previous page)

```
    "lab/spectrumanalyser/1",
    SpectrumAnalyserDevice,
    Host="specan.lab.example.com",
    Port=5024,
)
with context_manager as context:
    signal_generator = context.get_device("lab/signalgenerator/1")
    spectrum_analyser = context.get_device("lab/spectrumanalyser/1")

    # test your devices...
```

Unfortunately, there is a known bug in the underlying `tango.test_context.MultiDeviceTestContext` that this class uses: it cannot service `tango.DeviceProxy` requests that are specified with a device name; such requests must been specified with a Tango resource locator. (For more information see pytango issue https://gitlab.com/tango-controls/pytango/-/issues/459.)

This bug makes it necessary to patch `tango.DeviceProxy`, to convert device names into resource locators. To achieve this, a drop-in replacement *`ska_tango_testing.context.DeviceProxy`* is provided. Until the bug is fixed, production code should use this class instead of `tango.DeviceProxy`.

`ThreadedTestTangoContextManager` also supports mock devices. This is done with the *`add_mock_device()`* method:

```
context_manager = ThreadedTestTangoContextManager()
context_manager.add_device(
    "lab/controller/1",
    LabControllerDevice,
    SignalGeneratorName="lab/siggen/1",
    SpectrumAnalyserName="lab/spectana/1",
)

context_manager.add_mock_device(
    "lab/siggen/1",
    unittest.mock.Mock(**signal_generator_mock_config)
)
context_manager.add_mock_device(
    "lab/spectana/1",
    unittest.mock.Mock(**spectrum_analyser_mock_config)
)
with context_manager as context:
    controller = context.get_device("lab/controller/1")
    signal_generator = context.get_device("lab/siggen/1")
    spectrum_analyser = context.get_device("lab/spectana/1")

    # Test that when we tell the lab controller to turn everything on,
    # the signal generator and spectrum analyser are told to turn on.
    controller.On()
    signal_generator.On.assert_called_once_with()
    spectrum_analyser.On.assert_called_once_with()
```

1.2 Mock consumers

The *MockConsumerGroup* class addresses the problem of testing production code that produces items asynchronously. It is low-level, powerful and flexible, but takes a bit to set up. It requires

- a *producer*. This is a callable that is called with a timeout, and either returns an item once it becomes available, or raises `queue.Empty` if no item has been produced at the end of the timeout period. The producer is the interface to the production code. The production code under test might actually contain something that can serve as a producer (for example, if the production code drops items onto a queue, then that queue's *get* method will serve). Alternatively, your test harness might have to wrap the production code with something that provides this *producer* interface.
- a *categorizer*. This is a callable that sorts items into categories that can be asserted on individually.
- *characterizers*. By default, assertions are made against a dictionary with two entries: an *item* entry contains the item that has been produced, and the *category* entry contains the category that it has been sorted into. Thus, we can assert what the item is, and we can assert what category it belongs to.

  If the item is complex and/or non-deterministic, however, we might not be able to construct an item to assert with. For example, suppose the item is an *Event*, with fields *name*, *value* and *timestamp*. We generally cannot predict the timestamp values, so we cannot construct an equivalent *item* that would let us `assert_item(item)`.

  A *characterizer* addresses this by modifying the dictionary that assertions are made against. In our example, we might provide a characterizer that inserts "name" and "value" items into the dictionary, thus allowing us to `assert_item(name="foo", value="bah")` and hence asserting against the bits that matter, while ignoring the timestamp.

With these things in place, here are some of the things that you can do in your tests:

- `group.assert_no_item()` – assert that no item at all is produced within the timeout period.
- `group.assert_item()` – assert that an item is produced. (This call would consume an item without really asserting anything about it, so wouldn't be used much.)
- `group.assert_item(item)` – assert that the next item produced (across the whole group) is equal to `item`.
- `group.assert_item(category="voltage")` – assert that the next item produced belongs to category "voltage".
- `group.assert_item(item, category="voltage")` – assert that the next item produced (across the whole group) is equal to item and belongs to category "voltage".
- `group.assert_item(name="voltage", value=pytest.approx(15.0))` – assert that the next item has a "name" characteristic equal to "voltage", and a "value" characteristic approximately equal to 15.0. (This assertion would require a characterizer to extract the "name" and "value" attributes from the item.)
- `group.assert_item(item, lookahead=2)` – assert that one of the next two items produced is equal to `item`.
- `group["voltage"].assert_item()` – assert that an item has been produced in the "voltage" category.
- `group["voltage"].assert_item(item)` – assert that the next item in category "voltage" is equal to `item`.
- `group["voltage"].assert_item(value=pytest.approx(15.0))` – assert that the next item in category "voltage" has a "value" characteristic approximately equal to 15.0. (This assertion would require a characterizer to extract the "value" attribute from the item.)
- `group["voltage"].assert_item(item, lookahead=2)` – assert that one of the next two items in the "voltage" category are equal to `item`.

1.3 Mock callables

Mock callables build on mock consumers to addresses the problem of testing production code that makes asynchronous calls to callables.

1.3.1 An example

Consider this example:

```
def do_asynchronous_work(
    status_callback: Callable[[str], None],
    letter_callback: Callable[[str], None],
    number_callback: Callable[[int], None],
) -> None:
    def call_letters() -> None:
        for letter in ["a", "b", "c", "d"]:
            time.sleep(0.1)
            letter_callback(letter)

    letter_thread = threading.Thread(target=call_letters)

    def call_numbers() -> None:
        for number in [1, 2, 3, 4]:
            time.sleep(0.1)
            number_callback(number)

    number_thread = threading.Thread(target=call_numbers)

    def run() -> None:
        status_callback("IN_PROGRESS")

        letter_thread.start()
        number_thread.start()

        letter_thread.join()
        number_thread.join()

        status_callback("COMPLETED")

    work_thread = threading.Thread(target=run)
    work_thread.start()
```

We can test this example by testing that callbacks are called in the order expected. What we expect is that:

- The first call will be a call of “IN_PROGRESS” to the status callback
- The numbers callback will be called consecutively with “1”, “2”, “3” and “4”.
- The letters callback will be called consecutively with “a”, “b”, “c” and “d”.
- The global order in which the number and letter callbacks are called is nondeterministic. One possible ordering is “1”, “a”, “2”, “b”, “3”, “c”, “d”, “4”; but there are many other possibilities.
- The final call will be a call of “COMPLETED” to the status callback.

1.3.2 Testing with a `unittest.mock`

It is extremely hard to test asynchronous code like this using a standard `unittest.mock.Mock`. A test might look something like this:

```
def test_do_asynchronous_work_using_unittest_mock() -> None:
    status_callback = unittest.mock.Mock()
    letters_callback = unittest.mock.Mock()
    numbers_callback = unittest.mock.Mock()

    do_asynchronous_work(
        status_callback,
        letters_callback,
        numbers_callback,
    )

    time.sleep(0.05)

    status_callback.assert_called_once_with("IN_PROGRESS")
    status_callback.reset_mock()

    time.sleep(0.1)
    letters_callback.assert_called_once_with("a")
    letters_callback.reset_mock()
    numbers_callback.assert_called_once_with(1)
    numbers_callback.reset_mock()

    time.sleep(0.1)
    letters_callback.assert_called_once_with("b")
    letters_callback.reset_mock()
    numbers_callback.assert_called_once_with(2)
    numbers_callback.reset_mock()

    time.sleep(0.1)
    letters_callback.assert_called_once_with("c")
    letters_callback.reset_mock()
    numbers_callback.assert_called_once_with(3)
    numbers_callback.reset_mock()

    time.sleep(0.1)
    letters_callback.assert_called_once_with("d")
    numbers_callback.assert_called_once_with(4)

    status_callback.assert_called_once_with("COMPLETED")
```

Note that we start by sleeping for 0.05 seconds: long enough to make it unlikely that the test code will outrun the code under test, and assert a call before it has been made… but not so long that a callback will have been called more than once.

We then sleep for 0.1 seconds in the test, whenever the code under test sleeps for 0.1 seconds. It's easy to do this when you know the exact code timings. However real-world code won't contain sleeps of known duration. Rather, they will do things like file I/O, network I/O, or waiting for a lock, which have unknown and variable time costs. In such cases, it is difficult or even impossible to tune the sleeps in your test so that the test passes reliably. One tends to err on the side of caution by sleeping for longer than necessary.

In short, tests like this one are extremely brittle, and often very slow.

1.3.3 Testing with mock callables

The *MockCallable* and *MockCallableGroup* classes simplify testing behaviour like this, removing the need for tuned sleeps, and ensuring that the test takes no longer than necessary to run:

```
def test_do_asynchronous_work_using_mock_callback_group() -> None:
    callback_group = MockCallableGroup()

    do_asynchronous_work(
        callback_group["status"],
        callback_group["letters"],
        callback_group["numbers"],
    )

    callback_group.assert_call("status", "IN_PROGRESS")

    for letter in ["a", "b", "c", "d"]:
        callback_group["letters"].assert_call(letter)

    for number in [1, 2, 3, 4]:
        callback_group["numbers"].assert_call(number)

    callback_group.assert_call("status", "COMPLETED")
```

We now have a clean, readable test, with no sleeps.

Note that we can

- make assertions against the entire group, in which case we are asserting that the next call will be a specific call to a specific callback.
- use syntax like `callback_group["letters"]` to extract a particular callback, and then make assertions against that callback alone.

1.4 Callable behaviour

Sometimes when the production code calls a callable, it expects some action to be performed or a value to be returned. Thus, if we simply replace that callable with a *MockCallable*, the expectations of of the caller will not be met, and thus problems may arise in testing.

Two mechanisms are provided to address this:

1. The *MockCallable* class has a *configure_mock()* method that can be used to configure the behaviour of an underlying `unittest.mock.Mock`. For example, if the callable is supposed to return an integer, we can configure the underlying mock so that it always returns *1*:

   ```
   mock_callable = MockCallable()
   mock_callable.configure_mock(return_value=1)
   ```

   The arguments to *configure_mock* are passed straight through to the `unittest.mock.Mock.configure_mock()` method of the underlying `unittest.mock.Mock`; so see that method for documentation.

2. For cases where a callable performs essential behaviour that cannot be mocked out, there is the option of *wrapping* the callable with a *MockCallable*. When we wrap, the underlying callable still gets called, but the call passes through the *MockCallable* on the way, allowing us to assert against calls in the usual way:

```
mock_callable = MockCallable(wraps=essential_callable)
mock_callable.assert_call(0.0)
```

or

```
mock_callable = MockCallable()
mock_callable.wraps(essential_callable)
mock_callable.assert_call(0.0)
```

or

```
mock_callables = MockCallableGroup("foo", "bah")
mock_callable["bah"].wraps(essential_callable)
mock_callable.assert_call(bah, 0.0)
```

Note that these two options are mutually exclusive: a *MockCallable* always wraps something: it is just a question of whether the thing wrapped is a vanilla mock, a configured mock, or a user-provided callable. Thus, each call to *configure_mock* or *wraps* replaces any previous call.

1.5 Mock Tango event callbacks

A common use case for testing against callbacks in SKA is the callbacks that are called when Tango events are received. We can effectively test Tango device simply by using these callbacks to monitor changes in device state.

The *MockTangoEventCallbackGroup* class is a subclass of *MockConsumerGroup* with built-in characterizers that extract the key information from tango.EventData instances. Specifically, it extracts the attribute name, value and quality, and stores them under keys "attribute_name", "attribute_value" and "attribute_quality" respectively.

```
device_under_test.On()
callbacks.assert_change_event("command_status", "QUEUED")

# We can't be completely sure which of these two will arrive first,
# so lets give the first one a lookahead of 2.
callbacks.assert_change_event("command_status", "IN_PROGRESS", lookahead=2)
callbacks.assert_change_event("command_progress", "33")
callbacks.assert_change_event("command_progress", "66")

callbacks.assert_change_event("device_state", DevState.ON)
callbacks.assert_change_event(
    "device_status", "The device is in ON state."
)

callbacks.assert_change_event("command_status", "COMPLETED")
callbacks.assert_not_called()
```

For spectrum and image attributes, the values in Tango change events are numpy arrays. However assertions should be expressed using python lists:

```
# The change event will actually contain a numpy array,
# but this assertion will still pass if the elements are the same
callbacks.assert_change_event("levels", [1, 2, 1, 3, 2])
```

1.6 Return values

All methods that assert the presence of an item, such as *assert_item()*, *assert_call()*, *assert_against_call()* and *assert_change_event()*, return the matched item. This is useful as a diagnostic tool when developing tests. Suppose, for example, that you are writing a test, and the assertion

```
callback.assert_call(power=PowerState.ON)
```

fails unexpectedly. *Why* has it failed? Did the call not arrive? Is the value wrong? Was the value provided as a position argument rather than a keyword argument? Are there additional arguments?

The assertion made by `assert_call` is quite strict; in our example, it asserts that the call arguments are *exactly (power=PowerState.ON)*. We can relax this assertion to make it pass. For example,

```
callback.assert_against_call(power=PowerState.ON)
```

asserts only that the call *contains* the keyword argument *power=PowerState.ON*. Assuming that this more relaxed assertion passes, we can review the details of the match:

```
>>> call_details = callback.assert_against_call(power=PowerState.ON)
>>> print(call_details)
{'call_args': (,), 'call_kwargs': {'power': PowerState.ON, 'fault': False}}
```

Thus we see why our original assertion failed: the call also had a `fault` keyword argument. If this is not an bug in the production code, then we can now tighten up our test assertion again:

```
callback.assert_call(power=PowerState.ON, fault=False)
```

1.7 Logging

The *ska_tango_testing.mock* subpackage logs to the "ska_tango_testing.mock" logger. These logs exist to allow diagnosis of issues within *ska_tango_testing* itself, but may also assist with diagnosis of test failures.

Consider again the example above, of a test that fails on the line

```
callback.assert_call(power=PowerState.ON)
```

where `callback` is a *MockCallable*. To diagnose this failure, we can inspect the logs of the "ska_tango_testing.mock" logger. In pytest, this is done via the `caplog` fixture:

```
caplog.set_level(logging.DEBUG, logger="ska_tango_testing.mock")
callback.assert_call(power=PowerState.ON)
```

Running this test will now produce the following logs:

```
DEBUG    ska_tango_testing.mock:consumer.py:470 assert_item: Asserting item within next␣
↪1 item(s), with characteristics {'category': 'component_state', 'call_args': (), 'call_
↪kwargs': {'power': <PowerState.ON: 4>}}.
DEBUG    ska_tango_testing.mock:consumer.py:496 assert_item: 'call_kwargs'␣
↪characteristic is not '{'power': <PowerState.ON: 4>}' in item '{'category': 'component_
↪state', 'call_args': (), 'call_kwargs': {'power': <PowerState.ON: 4>, 'fault': False}}
↪'.
DEBUG    ska_tango_testing.mock:consumer.py:510 assert_item failed: no matching item␣
↪within the first 1 items
```

Thus we see why our assertion failed: the call also had a *fault* keyword argument. If this is not an bug in the production code, then we can now tighten up our test assertion again:

```
callback.assert_call(power=PowerState.ON, fault=False)
```

1.8 Assertion placeholders

Placeholders allow for flexibility in assertions by broadening the range of items that an assertion can match. So far, the only placeholder available is *Anything*. This matches any item at all.

For example, suppose we want to assert a call with keyword arguments *name*, *value* and *timestamp*, but we don't know exactly what the value of the *timestamp* will be. One way to make such an assertion is

```
from ska_tango_testing.mock.placeholders import Anything

mock_callback.assert_call(
    name="voltage",
    value=0.0,
    timestamp=Anything,
)
```

and this assertion will match irrespective of the actual value of the *timestamp* keyword.

Placeholders can be used in assertions anywhere that a specific item can by used:

- In a *MockConsumer* assertion, it can replace an item or a category or any characteristics
- In a *MockCallable* or *MockCallableGroup* assertion, it can substitute for a positional or keyword argument
- In a *MockTangoEventCallbackGroup*, it can substitute for an entire event, or for an event characteristic.

CHAPTER

TWO

API

2.1 ska_tango_testing

This package provides test harness for testing of SKA Tango devices.

2.2 ska_tango_testing.context

This module provides support for tango testing contexts.

ska_tango_testing.context.**DeviceProxy = <ska_tango_testing.context._DeviceProxyFactory object>**

A drop-in replacement for tango.DeviceProxy.

There is a known bug in tango.test_context.MultiDeviceTestContext for which the workaround is a patch to tango.DeviceProxy. This drop-in replacement makes it possible for *ThreadedTestTangoContextManager* to apply this patch. Until the bug is fixed, all production code that will be tested in that context must use this class instead of tango.DeviceProxy.

(For more information, see https://gitlab.com/tango-controls/pytango/-/issues/459.)

class ska_tango_testing.context.**TangoContextProtocol**(**args*, ***kwds*)

Protocol for a tango context.

__init__(**args*, ***kwargs*)

get_device(*device_name*)

Return a proxy to a specified device.

Parameters
device_name (str) – name of the device

Return type
DeviceProxy

Returns
a proxy to the device

class ska_tango_testing.context.**ThreadedTestTangoContextManager**

A lightweight context for testing Tango devices.

__init__()

Initialise a new instance.

add_device(*device_name*, *device_class*, ***properties*)

Add a device to the context managed by this manager.

Parameters

- **device_name** (str) – name of the device to be added
- **device_class** (Union[str, Type[Device]]) – the class of the device to be added. This can be the class itself, or its name.
- **properties** (Any) – a dictionary of device properties

Return type
None

add_mock_device(*device_name*, *device_mock*)

Register a mock at a given device name.

Registering this mock means that when an attempts is made to create a *tango.DeviceProxy* to that device name, this mock is returned instead.

Parameters

- **device_name** (str) – name of the device for which the mock is to be registered.
- **device_mock** (DeviceProxy) – the mock to be registered at this name.

Return type
None

class ska_tango_testing.context.**TrueTangoContextManager**

A Tango context in which the device has already been deployed.

For example, Tango has been deployed into a k8s cluster, and now we want to run tests against it.

get_device(*device_name*)

Return a proxy to a specified device.

Parameters
device_name (str) – name of the device

Return type
DeviceProxy

Returns
a proxy to the device

2.3 ska_tango_testing.mock

This subpackage provides mocks for testing of SKA Tango devices.

class ska_tango_testing.mock.**MockCallable**(*timeout=1.0*, *wraps=None*)

A class for a single mock callable.

__call__(**args*, ***kwargs*)

Register a call on this callable.

Parameters

- **args** (Any) – positional arguments in the call
- **kwargs** (Any) – keyword arguments in the call

Return type
Any

Returns
whatever this callable is configured to return

__init__(*timeout=1.0*, *wraps=None*)

Initialise a new instance.

Parameters

- **timeout** (Optional[float]) – how long to wait for the call, in seconds, or None to wait forever. The default is 1 second.
- **wraps** (Optional[Callable]) – a callable to be wrapped by this one. See *wraps()* for details.

assert_against_call(*lookahead=None*, ***kwargs*)

Assert that this callable has been called as characterised.

Parameters

- **lookahead** (Optional[int]) – The number of calls to examine in search of a matching call. The default is 1, which means we are asserting against the *next* call.
- **kwargs** (Any) – the characteristics that we are asserting the call to have. For details see *MockCallableGroup.assert_against_call()*.

Return type
Dict[str, Any]

Returns
details of the call

Raises
AssertionError – if the asserted call has not occurred within the timeout period

assert_call(**args*, ***kwargs*)

Assert that this callable has been called as specified.

For example, *assert_call("b", c=1, lookahead=2)* asserts that one of the next 2 calls to this callable will have call signature *("b", c=1)*.

This is syntactic sugar, which simplifies the expression of assertions, but also muddles up the arguments to *assert_call* with the arguments that we are asserting the call to have. It is equivalent to the more principled and flexible, but long-winded:

```
assert_against_call(
    call_args=("b",),
    call_kwargs={"c": 1},
    lookahead=2,
)
```

Parameters

- **args** (Any) – positional arguments asserted to be in the call.
- **kwargs** (Any) – If a "lookahead" keyword argument is provided, this specifies the number of calls to examine in search of a matching call. The default is 1, in which case we are asserting against the *next* call.

  All other keyword arguments are keyword arguments asserted to be in the call.

Return type
Dict[str, Any]

Returns
details of the call

Raises
AssertionError – if the asserted call has not occurred within the timeout period

assert_not_called()

Assert that this callable has not been called.

Raises
AssertionError – if this callable has been called.

Return type
None

configure_mock(**configuration)

Configure the underlying mock.

Parameters
configuration (Any) – keyword arguments to be passed to the underlying mock.

Return type
None

wraps(wrapped)

Specify a callable for this mock callable to wrap.

This allows use of this class as a shim between a caller and a called method. For example, suppose we need to provide an *important_callable* that we expect to be called with specific arguments. When called, this callable will do some very important work that cannot be mocked out.

In testing, instead of mocking out the callable, we can wrap it in a shim:

```
important_shim = MockCallable()
important_shim.wraps(important_callable)
```

This way, *important_callable* will still be called, but we can also assert on the call(s) as they pass through the shim:

```
important_shim.assert_call(0.0)
```

Note: calls to this method override any previous call to *configure_mock()*.

Parameters
wrapped (Callable) – a callable for this mock callable to wrap

Return type
None

class ska_tango_testing.mock.**MockCallableGroup**(**callables*, *timeout=1.0*, ***special_callables*)

This class implements a group of callables.

__init__(**callables*, *timeout=1.0*, ***special_callables*)

Initialise a new instance.

Parameters

- **callables** (str) – names of simple callables in this group; that is, callables that do not need a special characterizer.
- **timeout** (Optional[float]) – number of seconds to wait for the callable to be called, or None to wait forever. The default is 1.0 seconds.
- **special_callables** (Callable[[Dict[str, Any]], Dict[str, Any]]) – keyword argument for special callables that need a special characterizer. Each argument is of the form *callable_name=characterizer*.

assert_against_call(*callable_name*, *lookahead=None*, ***kwargs*)

Assert that the specified callable has been called as characterised.

Parameters

- **callable_name** (str) – name of the callable that we are asserting to have been called
- **lookahead** (Optional[int]) – The number of calls to examine in search of a matching call. The default is 1, which means we are asserting against the *next* call.
- **kwargs** (Any) – the characteristics that we are asserting the call to have. All call have
  - *call_args* and *call_kwargs* characteristics. For example,

```
example_callback("a", b=1)
example_callback("c", d=1)

assert_against_call(
    "example",
    lookahead=2,
    call_args=("c",),
    call_kwargs={"d": 1},
)
```

    asserts that one of the next two calls to callback "a" will have the signature ("b", c=1).

  - *argN* characteristics for *N* up to the number of positional arguments. For example, if a callable was called with three positional arguments, the captured call will have characteristics *arg0*, *arg1* and *arg2*:

```
assert_against_call(
    "example",
    lookahead=2,
    arg0="c",
)
```

  - a characteristic for each keyword argument. For example, if a callable was called with keyword argument *power=PowerState.ON*, then the captured call will have characteristic *power* with value *PowerState.ON*:

```
assert_against_call(
    "a",
    lookahead=2,
    d=1,
)
```

    If a characterizer was provided for the callback in this group's constructor, then there may be other characteristics that this method can assert against. For example, suppose we expect callable "a" to have been called with signature

```
callable_a(
    named_tuple(name="a", value=2, timestamp=1234567890)
)
```

but the timestamp is unknown. If we don't know the timestamp then we can't construct an equal object to assert:

```
assert_against_call(
    "a",
    arg0=named_tuple(name="a", value=2, timestamp=UNKNOWN)
)
```

Instead we can provide a characterizer that unpacks the "name" and "value" arguments for us, and then

```
assert_against_call(
    "a",
    lookahead=2,
    name="a",
    value=2,
)
```

Return type
Dict[str, Any]

Returns
details of the call

Raises
AssertionError – if the asserted call has not occurred within the timeout period

assert_call(*callable_name*, **args*, ***kwargs*)

Assert that the specified callable has been called as specified.

For example, *assert_call("a", "b", c=1, lookahead=2)* will assert that one of the next 2 calls to callable "a" will have call signature *("b", c=1)*.

This is syntactic sugar, which simplifies the expression of assertions, but also muddles up the arguments to *assert_call* with the arguments that we are asserting the call to have. It is equivalent to the more principled and flexible, but long-winded:

```
assert_against_call(
    "a",
    call_args=("b",),
    call_kwargs={"c": 1},
    lookahead=2,
)
```

Parameters

- **callable_name** (str) – name of the callable that we are asserting to have been called
- **args** (Any) – positional arguments asserted to be in the call.
- **kwargs** (Any) – If a "lookahead" keyword argument is provided, this specifies the number of calls to examine in search of a matching call. The default is 1, in which case we are asserting against the *next* call.

All other keyword arguments are keyword arguments asserted to be in the call.

Return type
Dict[str, Any]

Returns
details of the call

Raises
AssertionError – if the asserted call has not occurred within the timeout period

assert_not_called()

Assert that no callable in this group has been called.

Raises
AssertionError – if one of the callables in this group has been called.

Return type
None

class ska_tango_testing.mock.**MockConsumerGroup**(*producer*, *categorizer*, *timeout*, **consumers*, ***special_consumers*)

A group of consumers of items from a single producer.

__init__(*producer*, *categorizer*, *timeout*, **consumers*, ***special_consumers*)

Initialise a new instance.

Parameters

- **producer** (Callable[[Optional[float]], TypeVar(ItemType)]) – the producer from which this consumer gets items
- **categorizer** (Callable[[Any], str]) – a callable that categorizes items.
- **timeout** (Optional[float]) – optional number of seconds to wait for an item. If omitted, the default is 1 second. If explicitly set to None, the wait is forever.
- **consumers** (str) – list of simple consumers in this group
- **special_consumers** (Optional[Callable[[Any], Dict]]) – keyword arguments specifying special consumers in this group. Consumers are special if they have their own characterizer. Here, each key-value pair is the name of the consumer and the characterizer that it uses.

assert_item(**args*, *lookahead=1*, ***kwargs*)

Assert that an item is available in any category.

Parameters

- **args** (Any) – a single optional positional argument is allowed. If provided, it is asserted that there is an item available that is equal to the argument.
- **lookahead** (int) – how many items to look through for the item that we are asserting. The default is 1, in which case we are asserting what the very next item will be. This will be the usual case in deterministic situations where we know the exact order in which items will arrive. In non-deterministic situations, we can provide a higher value. For example, a lookahead of 2 means that we are asserting the item will be one of the first two items.
- **kwargs** (Any) – characteristics that the item is expected to have

Return type
Dict[str, Any]

Returns
the matched item

`assert_no_item()`
Assert that no item is available in any category.

Return type
None

2.4 ska_tango_testing.mock.placeholders

This module provides some special cases for equality checking.

Two special cases are provided: *Anything* and *OneOf*:

- ***Anything* can be used as a placeholder for assertions, in situations**
  where any item should be matched.

  For example, suppose we want to assert a call with keyword arguments *name*, *value* and *timestamp*, but we don't know exactly what the value of the *timestamp* will be. One way to make such an assertion is

```
from ska_tango_testing.mock.placeholders import Anything

mock_callback.assert_call(
    name="voltage",
    value=0.0,
    timestamp=Anything,
)
```

  and this assertion will match irrespective of the actual value of the *timestamp* keyword.

- ***OneOf* can be used as a placeholder for assertions, in situations**
  where we want to assert that the item will be a member of a specified set. See below for details.

`class ska_tango_testing.mock.placeholders.OneOf(*options)`
Equality placeholder that is equal if any of its args is equal.

When first initialised, an object of this class is provided with some number of arguments. Whenever we check if this object is equal to some other object, it returns True if and only if the other object is equal to one of its arguments.

This can be thus used as an assertion placeholder in situations where one does not know precisely what value will be returned:

```
from ska_tango_testing.mock.placeholders import OneOf

mock_callback.assert_call(
    name="state",
    value=OneOf(DevState.ON, DevState.ALARM),
)
```

and this assertion will match as long as one of the arguments to *OneOf* is met.

`__init__(*options)`
Initialise a new instance.

Parameters
options (Any) – any number of options against which to check equality.

2.5 ska_tango_testing.mock.tango

This subpackage provides test harness for testing SKA Tango devices.

class ska_tango_testing.mock.tango.**MockTangoEventCallbackGroup**(**callables*, *timeout=1.0*, *assert_no_error=True*)

This class implements a group of Tango change event callbacks.

__init__(**callables*, *timeout=1.0*, *assert_no_error=True*)

Initialise a new instance.

Parameters

- **callables** (str) – positional arguments providing the names of callables in this group.
- **timeout** (Optional[float]) – number of seconds to wait for the callable to be called, or None to wait forever. The default is 1.0 seconds.
- **assert_no_error** (bool) – defaults to True, in which case this callback group will assert that each event to arrive is not an error event. Tests can then proceed on that assumption. If False, this callback group will not assert that events are not error events, but rather will return "err" and "errors" values. Tests then have to be written to check for error events.

assert_change_event(*callback_name*, *attribute_value*, *lookahead=None*)

Assert that the callback received a change event with the given value.

Parameters

- **callback_name** (str) – name of the callback that we are asserting to have been called
- **attribute_value** (Any) – new value of the attribute for which the change event has been sent
- **lookahead** (Optional[int]) – The number of calls to examine in search of a matching call. The default is 1, which means we are asserting against the *next* call.

Return type
Dict[str, Any]

Returns
details of the change event

Raises
AssertionError – if the asserted call has not occurred within the timeout period

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

C

D

G

M

O

S

T

W