developer.skatelescope.org Documentation

Release 7.4.0

Marco Bartolini

Jun 21, 2023

HOME

CHAPTER

ONE

WHAT IS AN OBSERVING SCRIPT?

An observing script is some Python code whose purpose is to perform the observation defined in a Scheduling Block (SB). In practice, there may be several observing scripts that process different parts of the SB; for example, a script to allocate resources, a script to perform the observation; a script to deallocate resources, etc.

The SKA Project Data Model (PDM) defines the structure of Scheduling Blocks (SBs), and how an SB can be serialised as a JSON file. The syntax of the commands and configurations required by the telescope are described in the SKA Control Data Model (CDM). A major function of an observing script is to convert the SB PDM entity to the appropriate set of CDM allocation and configuration entities, which form the arguments for telescope control commands.

To achieve its aims the script must send a series of commands in the correct order. The typical sequence for 'executing' an SB from start to finish is:

1. Command the CentralNode to allocate the required resources and instantiate the SubArrayNode. The observing script parses the SB JSON, reading what resources are required and constructing the equivalent CDM JSON which it then issues to the control system.
2. Using the SubArrayNode, loop through the scans required by the SB:
    a. Command the SubArrayNode to configure for the scan. The script reads the subarray configurations, calculates the CDM JSON required for the scan configuration, and configures the subarray accordingly.
    b. Command the SubArrayNode to scan (that is, take data).
3. Tell the SubArrayNode and CentralNode that the SB is complete.
4. Command the CentralNode to release the resources in the SubArrayNode.

CHAPTER

TWO

CONTEXT

The relationship between SBs, observing scripts, and the devices that the scripts control is shown in the figure below.

Fig. 1: Observing scripts in the context of SBs, the OET, and SKA Tango devices.

Scheduling Blocks (SBs) are the atomic units of SKA observations. Each SB defines the required resources, configurations, scan sequences, timing constraints, and data processing that is required for that observation. Each SB also references an *observing script* - a set of Python instructions that will process the SB, translating the SB to the series of lower-level commands and JSON configuration strings that will control the SKA telescopes and take the required data.

During standard operations, the SKA will have an observing queue with a pool of pending SBs awaiting execution. SBs will be selected by the SKA Scheduler and sent to the Observation Execution Tool (OET) for execution. The OET is responsible for loading and executing the scripts, retrieving the script referenced by an SB, installing any dependencies into a Python environment and then executing the script.

The telescope appears to the script author as two Tango devices: the CentralNode is the target for commands to allocate resources to form a SubArrayNode, and release them again when the SubArrayNode is no longer needed after the observation; the SubArrayNode is the target for commands to configure and scan to take data during the observation.

CHAPTER

THREE

WRITING SCRIPTS FOR THE OET

The Observation Execution Tool (OET) can run observing scripts in a headless non-interactive manner. For efficiency, OET script execution is split into two phases: an initialisation phase and an execution phase. Scripts that are expected to be run by the OET should be structured to have two entry points corresponding to these two phases, as the template below:

Listing 1: Observing script template

```
def init(subarray: int, *args, **kwargs):
    # Called by the OET when the script is loaded and initialised by someone
    # calling 'oet prepare'. Add your script initialisation code here. Note that
    # the target subarray is supplied to this function as the first argument.
    pass

def main(*args, **kwargs):
    # Called by the OET when the prepared script is told to run by someone
    # calling 'oet start'. Add the main body of your script to this function.
    pass
```

The initialisation phase occurs when the script is loaded and the script's `init` function is called (if defined) to perform any preparation and/or initialisation. Expensive and slow operations that can be performed ahead of the main body of script execution can be run in the initialisation phase. Typical actions performed in init are I/O intensive operations, e.g., cloning a git repository, creating multiple Tango device proxies, subscribing to Tango events, etc. When run by the Observation Execution Tool (OET), the `init` function is passed an integer subarray ID declaring which subarray the control script is intended to control.

Subsequently, at some point a user may call `oet start`, requesting that the initialised script begin the main body of its execution. When this occurs, the OET calls the script's `main` function, which should performs the main function of the script. For an observing script, this would involve the configuration and control of a subarray.

Below are two short but real example scripts. Further examples can be found in the `scripts` folder of this project.

3.1 Telescope Startup

Telescope startup does not require a sub-array ID so the subarray ID argument is ignored when the script is initialised.

Listing 2: Telescope start-up script

```
"""
Example script for telescope startup
"""
```

(continues on next page)

(continued from previous page)

```
import logging
import os

from ska_oso_scripting.objects import Telescope

LOG = logging.getLogger(__name__)
FORMAT = '%(asctime)-15s %(message)s'

logging.basicConfig(level=logging.INFO, format=FORMAT)


def init(subarray_id):
    pass


def main(*args, **kwargs):
    """
    Start up telescope.
    """
    LOG.info(f'Running telescope start-up script in OS process {os.getpid()}')

    if args:
        LOG.warning('Got unexpected positional args: %s', args)
    if kwargs:
        LOG.warning('Got unexpected named args: %s', kwargs)

    telescope = Telescope()

    LOG.info(f'Starting telescope...')
    telescope.on()

    LOG.info('Telescope start-up script complete')
```

3.2 SKA-MID : Allocate Resources

Allocating resources requires communication with a TMC CentralNode and TMC SubarrayNode and targets a specific subarray. This script's `init` function pre-applies the subarray ID argument to the main function. Note that the script does not perform any Tango calls, with the `assign_resources_from_cdm()` library function called to perform all the required Tango interactions (command invocation; event subscriptions; event monitoring). Direct Tango calls could be performed in the script if the ska-oso-scripting library does not provide the helper functions needed.

Listing 3: Resource allocation script for an SKA MID subarray

```
"""
Example script for SB-driven observation resource allocation from file.
On default CDM file values (if CDM file is provided) will be overwritten
by values from the SB file. CDM file should be included if values are
added to the AssignResourcesRequest that are not available from the SB
"""
import functools
```

(continues on next page)

(continued from previous page)

```
import logging
import os

from ska_oso_oet.event import topics
from ska_tmc_cdm.messages.central_node.assign_resources import AssignResourcesRequest
from ska_tmc_cdm.messages.central_node.common import DishAllocation as cdm_DishAllocation
from ska_tmc_cdm.schemas import CODEC as cdm_CODEC
from ska_oso_pdm.entities.dish.dish_allocation import DishAllocation as pdm_
↪DishAllocation
from ska_oso_pdm.entities.common.sb_definition import SBDefinition

from ska_oso_pdm.schemas import CODEC as pdm_CODEC

from ska_oso_scripting.functions import devicecontrol, messages, environment
from ska_oso_scripting.functions import pdm_transforms

LOG = logging.getLogger(__name__)
FORMAT = '%(asctime)-15s %(message)s'

logging.basicConfig(level=logging.INFO, format=FORMAT)


def init(subarray_id: int):
    """
    Initialise the script, binding the sub-array ID to the script.
    """
    if environment.is_ska_low_environment():
        raise environment.ExecutionEnvironmentError(expected_env="SKA-mid")
    global main
    main = functools.partial(_main, subarray_id)
    LOG.info(f'Script bound to sub-array {subarray_id}')


def _main(subarray_id: int, sb_json, allocate_json=None):
    """
    Allocate resources to a target sub-array using a Scheduling Block (SB).

    :param subarray_id: numeric subarray ID
    :param sb_json: file containing SB in JSON format
    :param allocate_json: name of configuration file
    :return:
    """
    LOG.info(f'Running allocate script in OS process {os.getpid()}')
    LOG.info(
        f'Called with main(sb_json={sb_json}, configuration={allocate_json}, subarray_id=
↪{subarray_id})')

    if not os.path.isfile(sb_json):
        msg = f'SB file not found: {sb_json}'
        LOG.error(msg)
        raise IOError(msg)

```

(continues on next page)

(continued from previous page)

```
    if allocate_json:
        if not os.path.isfile(allocate_json):
            msg = f'CDM file not found: {allocate_json}'
            LOG.error(msg)
            raise IOError(msg)

        cdm_allocation_request: AssignResourcesRequest = cdm_CODEC.load_from_file(AssignResourcesRequest, allocate_json)
    else:
        cdm_allocation_request = AssignResourcesRequest(subarray_id, None, None)

    pdm_allocation_request: SBDefinition = pdm_CODEC.load_from_file(SBDefinition, sb_json)

    # Configure PDM DishAllocation to the equivalent CDM DishAllocation
    pdm_dish = pdm_allocation_request.dish_allocations
    cdm_dish = convert_dishallocation(pdm_dish)
    LOG.info(f'Setting dish : {cdm_dish.receptor_ids} ')

    # Configure PDM SDPConfiguration to the equivalent CDM SDPConfiguration
    pdm_sdp_config = pdm_allocation_request.sdp_configuration
    cdm_sdp_config = pdm_transforms.convert_sdpconfiguration_centralnode(pdm_sdp_config,
                                                                         pdm_allocation_request.targets)
    LOG.info(f'Setting SDP configuration for EB: {cdm_sdp_config.execution_block.eb_id}')

    cdm_allocation_request.dish = cdm_dish
    cdm_allocation_request.sdp_config = cdm_sdp_config

    response = devicecontrol.assign_resources_from_cdm(subarray_id, cdm_allocation_request)
    LOG.info(f'Resources Allocated: {response}')

    messages.send_message(topics.sb.lifecycle.allocated, sb_id=pdm_allocation_request.sbd_id)

    LOG.info('Allocation script complete')


def convert_dishallocation(pdm_config: pdm_DishAllocation) -> cdm_DishAllocation:
    """
    Convert a PDM DishAllocation to the equivalent CDM DishAllocation.
    """

    return cdm_DishAllocation(
        receptor_ids=pdm_config.receptor_ids
    )

```

CHAPTER

FOUR

CONTROLLING SUBARRAYS WITHOUT SBS

At the highest level, SKA subarrays are configured and controlled by a handful of commands, some of which require JSON control strings. Scripted control of subarrays can be achieved by creating these JSON control strings - either directly or by using the ska-cdm-library to create the Python equivalent of the JSON control strings - and then sending these to the subarray via `SubArray` methods or the appropriate ska-oso-scripting functions.

Consult the API documentation for `ska_oso_scripting.objects.SubArray` for details on the methods used to assign resources to a subarray, configure a subarray, perform a scan, etc.

See `ska_oso_scripting.objects.Telescope` for methods used to turn the telescope on and off.

See `ska_oso_scripting.functions.devicecontrol` for functions you can use for high-level telescope and subarray control.

4.1 Tweaking configuration strings

Resource allocation and configuration require lengthy JSON strings as input. Modifying these JSON payloads is most easily achieved by using classes from the ska-tmc-cdm project. Use of the Control Data Model (CDM) objects allows JSON payloads to be modified via Python rather than by modifying JSON strings directly and reading/writing those changes to a file.

In the absence of an SB, the recommended way to control the telescope is to:

1. use the ska-oso-cdm library to convert JSON contained in a file or string to the equivalent Python objects
2. perform any required modifications to the CDM objects
3. relay the instructions to the control system using the appropriate methods (`assign_from_cdm()`, `configure_from_cdm()`, etc.)

The script below illustrates how this can be done.

```
from datetime import timedelta
from ska_tmc_cdm.messages.central_node.assign_resources import AssignResourcesRequest
from ska_tmc_cdm.messages.subarray_node.configure import ConfigureRequest
from ska_tmc_cdm.schemas import CODEC
from ska_oso_scripting.objects import SubArray, Telescope

# Create telescope object to control telescope start-up and shut-down
telescope = Telescope()
# Create sub-array object which will form the target for subsequent instructions
subarray = SubArray(1)

# Turn the telescope on
```

(continues on next page)

(continued from previous page)

```
telescope.on()

# Create a CDM AssignResourcesRequest object. This example loads JSON
# from file but a request can also be formed from a JSON string using
# CODEC.loads(AssignResourcesRequest, allocation_json_string)
request = CODEC.load_from_file(AssignResourcesRequest, file_path, timeout=None)
# Modify request object here if necessary, e.g.
request.dish.receptor_ids = ["SKA001", "SKA002", "SKA003", "SKA004"]
# issue resource allocation request
subarray.assign_from_cdm(request, timeout=None)

# Similarly, create a CDM ConfigureRequest object. Again, this could
# also be formed from a JSON string using
# CODEC.loads(ConfigureRequest, configuration_json_string)
request = CODEC.load_from_file(ConfigureRequest, file_path)
# Modify request object here if necessary, e.g.
request.tmc.scan_duration = timedelta(seconds=10.0)
# issue sub-array configuration request
subarray.configure_from_cdm(request)

# Execute scan
subarray.scan(timeout=None)

# End the Scheduling Block
subarray.end()

# release all sub-array resources
subarray.release(timeout=None)

# Set telescope to standby
telescope.off()
```

4.2 Control using static JSON

For an interaction where no modifications to the CDM are required, you can also use the `assign_from_file()` and `configure_from_file()` methods, which will relay the JSON directly to the control system. The JSON will be validated against the required JSON schema and any elements that are required to be unique from observation to observation, such as scheduling block ID and processing block ID, will managed as necessary.

Note: You can also send the raw JSON directly to the control system without performing any validation or ID updates by setting `with_processing=False` for these methods. However, it is then your responsibility to ensure that the CDM payloads are valid!

```
from ska_oso_scripting.objects import SubArray, Telescope

# Create telescope object to control telescope start-up and shut-down
telescope = Telescope()
```

(continues on next page)

(continued from previous page)

```
# Turn the telescope on
telescope.on()

# Create domain object for the sub-array the commands will be sent to
subarray = SubArray(1)

# Allocate resources, provide a path to a file with allocation JSON
subarray.assign_from_file(path_to_allocation_json_file, timeout=None)

# Configure sub-array, provide a path to a file with configuration JSON
subarray.configure_from_file(configuration_json_file, scan_duration=10.0, timeout=None)

# Execute scan sub-array was configured for
subarray.scan()

# End the Scheduling Block
subarray.end()

# Set telescope to standby
telescope.off()
```

CHAPTER

FIVE

EXECUTION BLOCKS

5.1 What is an Execution Block?

Execution Blocks (EBs) are a record of the requests to and responses from the telescope during an observing session, and provide a way to link the data to this observing session. For more information and a sample of an EB, see the documentation for the PDM project, which is where the EB is defined.

EBs are stored in the OSO Data Archive (ODA), and the ODA application provides custom API resources for creating and updating EBs. The ODA also provides a client with functions to call this create API and a decorator which will send the decorated function request and response to the endpoint to update the EB. See the documentation for the ODA project for more details.

The client is implemented in a way that minimises the visibility of the EB lifecycle, however the script author or notebook user should be aware of the purpose of an EB and should follow the instructions below so ensure the session is properly captured in an EB.

5.2 Capturing an observing session in an Execution Block

An observing session typically means either the Scheduling Block driven execution of observing scripts defined in this project or elsewhere, or a Jupyter notebook session where a user is interacting with the telescope, either through the functions in this project or at a lower level.

In either case, the same process for capturing data in an Execution Block should be followed:

1. Set the ODA_URI environment variable to the location of a running instance of the ODA, eg ODA_URI=https://k8s.stfc.skao.int/button-dev-ska-db-oda/api/v1/
2. At the start of the session or script, call the create_eb from the ODA EB client discussed above

```
from ska-oso-scripting import create_eb

create_eb()
```

3. Functions that are decorated with `@capture_request_response` will send the request_responses to the ODA with the relevant eb_id. The public functions in ska_oso_scripting.functions.devicecontrol are already decorated. To capture custom function calls, the decorator can either imported and added to the function definition, or functions calls can be decorated on the fly during the execution in the session:

```
from ska-oso-scripting import capture_request_response

# Decorate the function in the source code
```

(continues on next page)

(continued from previous page)

```
@capture_request_response
def my_function_to_record(args):
    ...

# OR 'decorate' the function at runtime when calling
my_response = capture_request_response(my_function_to_record)(args)
```

CHAPTER

SIX

ENVIRONMENT VARIABLES

Environment variables are used by ska-oso-scripting to define the execution environment and which Tango devices should be controlled. This in turn modifies the behaviour of the code, giving different behaviour depending on whether the code is running in an SKA MID environment (default) or an SKA LOW environment. For example, when configured for SKA MID, connections to a subarray will connect to an SKA MID subarray, and validation code will reject CDM payloads intended for SKA LOW.

Table 1: Environment variables recognised by ska-oso-scripting

Variable	Default value	Description
`SKA_TELESCOPE`	`skamid`	Controls the behaviour of telescope-specific functions to expect SKA LOW (`skalow`) or SKA MID (`skamid`). If the environment variable is not recognised, `skamid` will be assumed.
`CENTRALNODE_FQDN`	`ska_mid/tm_central/central_node`	The fully-qualified domain name (FQDN) of the TMC CentralNode Tango device. If left unset, an appropriate FQDN for SKA MID will be used.
`SUBARRAYNODE_FQDN_PREFIX`	`ska_mid/tm_subarray_node`	Prefix to use when constructing the FQDN for a TMC SubarrayNode Tango device. If left unset, an apprpriate FQDN prefix for SKA MID will be used.
`ODA_URI`	`null`	The base API location for an instance of the ODA, to be used to create and update Execution Blocks. For example, https://k8s.stfc.skao.int/button-dev-ska-db-oda/api/v1/

CHAPTER

SEVEN

SKA_OSO_SCRIPTING.OBJECTS

CHAPTER

EIGHT

SKA_OSO_SCRIPTING.FUNCTIONS.DEVICECONTROL

CHAPTER

NINE

SKA_OSO_SCRIPTING.FUNCTIONS.ENVIRONMENT

The ska_oso_scripting.functions.environment module contains code used to identify the execution environment and to verify commands are appropriate to that environment.

exception ska_oso_scripting.functions.environment.**ExecutionEnvironmentError**(*expected_env*, *msg='Telescope environment required by script does not match deployment'*)

Error raised when execution environment does not match the targeted telescope.

ska_oso_scripting.functions.environment.**check_environment_for_consistency**(*request_json: str*) → None

Confirm that the request is correct for the current environment, raising an ExecutionEnvironmentError if there is a mismatch.

Parameters
request_json – The JSON control string to be tested

ska_oso_scripting.functions.environment.**is_ska_low_environment**() → bool

Return True if execution environment is identified as SKA LOW.

Returns
True if SKA LOW

ska_oso_scripting.functions.environment.**is_ska_mid_environment**() → bool

Return True if execution environment is identified as SKA MID

Returns
True if SKA MID

CHAPTER

TEN

SKA_OSO_SCRIPTING.FUNCTIONS.PDM_TRANSFORMS

The pdm_transforms module contains code to transform Project Data Model (PDM) entities to Configuration Data Model (CDM) entities. The pdm_transforms code is called by observing scripts to convert the PDM Scheduling Block to the equivalent CDM configurations, which are then sent to TMC devices to control the telescope.

ska_oso_scripting.functions.pdm_transforms.pdm_transforms.**convert_cspconfiguration**(*pdm_config: ska_oso_pdm.entities.csp.c* *re-* *ceiver_band: ska_oso_pdm.entities.dish.* → ska_tmc_cdm.messages.su

Convert a PDM CSPConfiguration to the equivalent CDM CSPConfiguration.

Parameters

- **pdm_config** – The PDM configuration to convert
- **receiver_band** – PDM receiver band to set for this configuration

Returns
the equivalent CDM configuration

ska_oso_scripting.functions.pdm_transforms.pdm_transforms.**convert_mccs_configuration**(*mccs_allocation: ska_oso_pdm.entities.m* *tar-* *get_beam_configuration List[ska_oso_pdm.entiti* *tar-* *gets: List[ska_oso_pdm.entiti* *sub-* *ar-* *ray_beam_configuration List[ska_oso_pdm.entiti* *sub-* *ar-* *ray_beam_map: Map-* *ping[ska_oso_pdm.entit* *int]*) → ska_tmc_cdm.messages

Convert PDM Low SB TargetBeamConfiguration list to a CDM MCCSConfiguration.

Other SB elements required are the Target list and SubarrayBeamConfiguration list, which are referenced by the TargetBeamConfigurations.

The MCCSAllocation is also needed for its list of station_ids.

Parameters

- **mccs_allocation** – The PDM MCCSAllocation
- **target_beam_configurations** – The PDM TargetBeamConfiguration list
- **targets** – The PDM Target list
- **subarray_beam_configurations** – The PDM SubarrayBeamConfiguration list
- **subarray_beam_map** – mapping of offline beam IDs to online beam IDs

Returns
the required CDM MCCSConfiguration

ska_oso_scripting.functions.pdm_transforms.pdm_transforms.**convert_mccsallocation**(*mccsallocation: ska_oso_pdm.entities.mccs.m subarray_beam_map: Mapping[ska_oso_pdm.entities.m int]*, *union: bool = False*) → List[ska_tmc_cdm.messages.

Convert a PDM Low MCCSAllocation to a list of CDM MCCSAllocate instances, one CDM instance per beam.

At the time of writing (PI10), MCCS requires subarray beams to be allocated one at a time, that is, one subarray beam per allocation request. This behaviour can be toggled by setting the union argument. When union is set to True, this function returns a list of CDM instances, each instance narrowed to configure a single beam.

Parameters

- **mccsallocation** – The PDM MCCSAllocation
- **subarray_beam_map** – mapping of offline beam IDs to online beam IDs
- **union** – True to create one multi-beam request, False to create n per-beam requests

Returns
equivalent CDM instances

ska_oso_scripting.functions.pdm_transforms.pdm_transforms.**convert_tmcconfiguration**(*scan_definition: ska_oso_pdm.entities.com* → ska_tmc_cdm.messages.su

Convert a PDM ScanDefinition to the equivalent TMC configuration

ska_oso_scripting.functions.pdm_transforms.pdm_transforms.**convert_pointingconfiguration**(*target: ska_oso_pdm.entitie* → ska_tmc_cdm.messa

Convert a PDM Target to the equivalent TMC configuration

ska_oso_scripting.functions.pdm_transforms.pdm_transforms.**convert_dishconfiguration**(*dish_configuration: ska_oso_pdm.entities.dis* → ska_tmc_cdm.messages.s

Convert a PDM Dish configuration to a CDM Dish Configuration

The pdm_transforms.sdp module contains code to transform SDP Project Data Model (PDM) entities to Configuration Data Model (CDM) entities.

ska_oso_scripting.functions.pdm_transforms.sdp.**convert_sdpconfiguration_centralnode**(*pdm_config: ska_oso_pdm.entities.sdp pdm_targets: List[ska_oso_pdm.entitie* → ska_tmc_cdm.messages.c

Convert a PDM SDPConfiguration to the equivalent CDM SDPConfiguration.

In a SchedulingBlockDefinition, Targets are recorded exactly once as PDM Targets separate and external to any SDPConfiguration. Targets to be inserted into the output SDPConfiguration should be passed to this function.

Parameters

- **pdm_config** – the SDPConfiguration to convert
- **pdm_targets** – Targets to inject into output SDP configuration

Raises

TypeError – if pdm_config is not an SDPConfiguration

ska_oso_scripting.functions.pdm_transforms.sdp.**convert_sdpconfiguration_subarraynode**(*scan_definition: ska_oso_pdm.entities.co* → ska_tmc_cdm.messages

Convert a PDM Scan Definition to an SDP Configuration aspect of a TMC SubArrayNode.Configure call

CHAPTER

ELEVEN

SKA_OSO_SCRIPTING.FUNCTIONS.MESSAGES

The ska_oso_scripting.functions.messages module contains functions that scripts can use to announce events and messages to the outside world.

ska_oso_scripting.functions.messages.**send_message**(*topic*, ***kwargs*)

Helper function to send messages via pypubsub.

Parameters

- **topic** – topic matching a topic in oet.event.topics
- **kwargs** – kwargs to be included in message

ska_oso_scripting.functions.messages.**publish_event_message**(*topic=ska_oso_oet.event.topics.user.script.announce*, ***kwargs*)

publish pypubsub event messages, OET scripts will be using this method to publish a freeform messages to an unknown listener.

Parameters

- **topic** – message topic
- **kwargs** – any metadata associated with pypubsub message

CHAPTER

TWELVE

SKA_OSO_SCRIPTING.FUNCTIONS.SB

ska_oso_scripting.functions.sb.**create_sbi**(*sbd: ska_oso_pdm.entities.common.sb_definition.SBDefinition*) → ska_oso_pdm.entities.common.sb_definition.SBDefinition

Create a Scheduling Block Instance from a Scheduling Block Definition.

Currently, an SBI is a snapshot of an SBD but with EB and PB IDs replaced.

ska_oso_scripting.functions.sb.**load_sbd**(*path: str*) → ska_oso_pdm.entities.common.sb_definition.SBDefinition

Load an SBDefinition from a JSON file on disk. :param path: path to SBD. :return: SBDefinition object

ska_oso_scripting.functions.sb.**save_sbi**(*sbi: ska_oso_pdm.entities.common.sb_definition.SBDefinition*, *path: str*)

Save an SBI to disk. Saves an SBI (really, an SBD but with fixed IDs) the specified path. :param sbi: SBI to serialise :param path: output file to write

CHAPTER

THIRTEEN

SKA-OSO-SCRIPTING

13.1 Overview

The ska-oso-scripting project provides a Python library intended to be useful for engineers and scientists writing *observing scripts* and engineering tests. The helper functions and classes contained in the library support the high-level configuration and control of an SKA subarray, hiding the low-level details of how the related Tango devices are controlled from the script author.

This library provides a simple object-oriented interface and a functional interface, located in `ska_oso_scripting.objects` and `ska_oso_scripting.functions` respectively. The objects are recommended as the most user-friendly option but the functions can also be called if preferred. Regardless, the same code is called at the lowest level: object methods call 'public' scripting functions, which then call 'private' lower-level scripting functions held in submodules of `devicecontrol`.

Observing scripts can be run interactively in a Jupyter notebook, or remotely executed by the Observation Execution Tool (OET). For documentation on how to use OET to run observing scripts, see the ska-oso-oet project documentation.

A major use case for the ska-oso-scripting library is to support the execution of Scheduling Blocks (SBs). Hence, in addition to Tango device control, much of the ska-oso-scripting library is concerned with translating SBs into the equivalent JSON configuration and control strings and with the issuing of commands to TMC Tango devices at the appropriate times.

13.2 Quickstart

Like all SKA projects, this project uses containers for development and testing so that the build environment, test environment and test results are reproducible and independent of the host environment. `make` is used to provide a consistent UI.

Build a new container image for the OET with:

```
make oci-build
```

Execute the test suite with:

```
make python-test
```

Format and lint the Python code with:

```
make python-format
make python-lint
```

PYTHON MODULE INDEX

S

INDEX