developer.skatelescope.org Documentation

Release 4.2.0

Marco Bartolini

May 15, 2024

CONTENTS

The PSS Product testing framework (ProTest) consists of a set of "product" tests, supported by a number of python-based backend applications, for verifying that the Square Kilometre Array's Pulsar Searching Sub-system (PSS) conforms to SKA requirements.

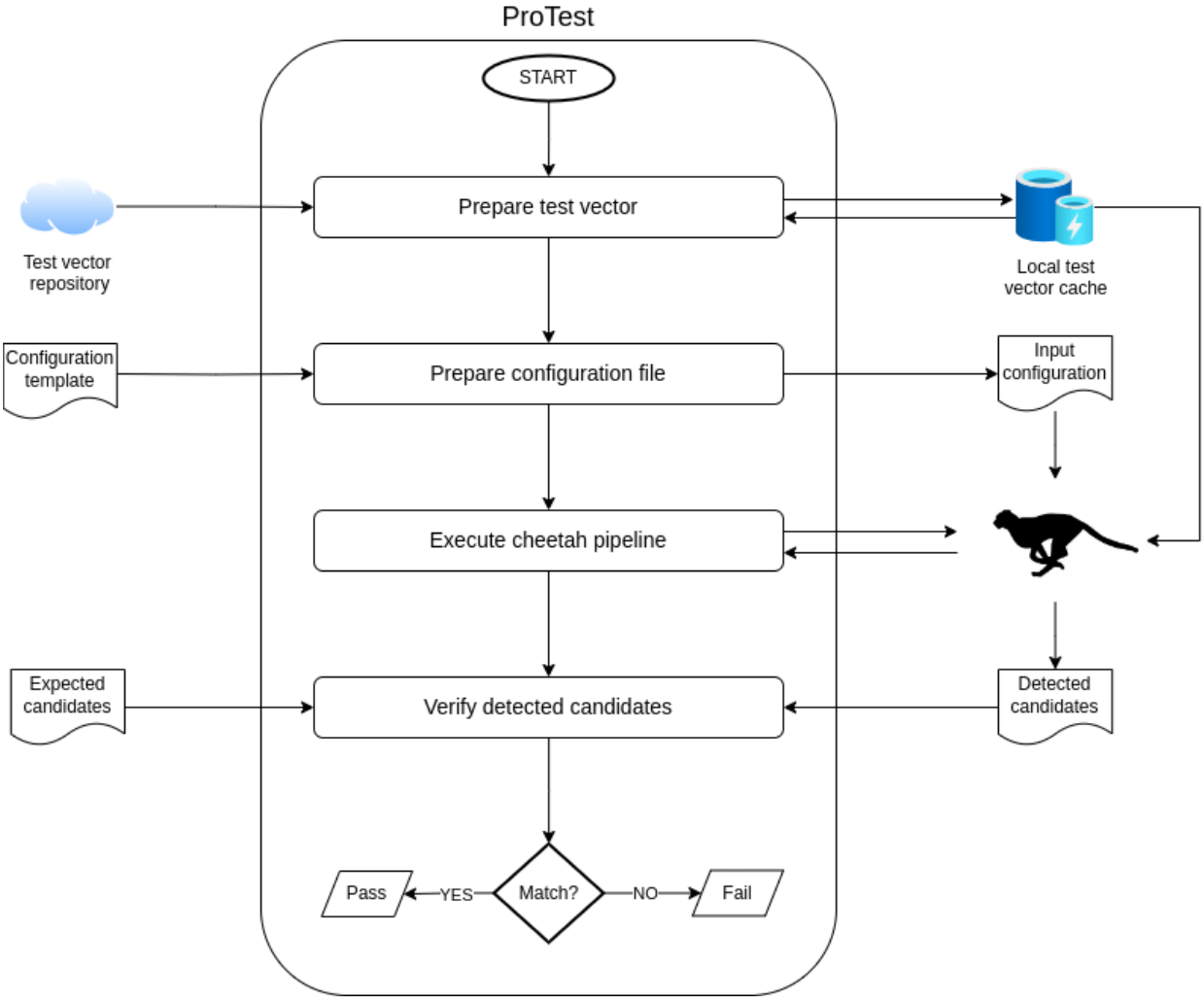
Pulsar and fast-transient searching is carried out using an application called cheetah (see https://ska-tdt.gitlab.io/cheetah/). Cheetah comprises a number of modules which are added together and configured to form a pipeline. Whilst each of these modules is supported by its own set of unit tests, ProTest's purpose is to execute the pipeline as a whole, and verify that it performs as designed. In this sense, ProTest considers cheetah to be a "black box".

Cheetah is a command line application which requires a number of command line arguments. ProTest prepares cheetah for execution, ensuring all inputs and command line arguments are provided and make sense. It then executes cheetah as a child process, in most cases producing a set of candidate detections. ProTest then evaluates these candidates and uses them to determine whether or not cheetah is behaving as designed.

Examples of command line arguments required by cheetah are as follows.

- The pipeline type (i.e., what are we searching for? Pulsars and/or single pulses).
- The data source (e.g., a test vector (filterbank file) or a UDP data stream).
- A configuration. This is usually an XML file which enables/disables and tunes each of cheetah's modules so that they can pieced together to form a pipeline.

The basic workflow of ProTest, and how it executes cheetah, is summarised in the following diagram.



CHAPTER

ONE

INSTALLING PROTEST

A python>=3.10 virtual environment is recommended

1.1 Install with pip

```
pip install --index-url https://artefact.skao.int/repository/pypi-internal/simple --
↪extra-index-url https://pypi.org/simple ska-pss-protest
```

Verify that protest has successfully installed.

```
protest -h
```

1.2 Install from source

Ensure poetry is installed first

```
which poetry
```

If so, proceed as follows.

```
git clone --recursive https://gitlab.com/ska-telescope/pss/ska-pss-protest.git
cd ska-pss-protest
export PYTHON_KEYRING_BACKEND=keyring.backends.null.Keyring
poetry install --without dev
```

1.3 Install from source (for developers)

If you are planning to contribute to ProTest, run the git clone command above. If you have an existing clone, you can check whether you have the correct submodules (made available by –recursive) by running

```
git submodule status
```

and if nothing is returned, use

```
git submodule init
git submodule update
```

To install a developer version of ProTest, run

```
poetry install
```

Start virtual environment

```
poetry shell
```

Verify that protest has successfully installed.

```
protest -h
```

If required, you can verify the install further by executing the unit tests

```
make python-test
```

CHAPTER

TWO

RUNNING PSS PRODUCT TESTS WITH PROTEST

ProTest is a command line application. It is essentially a wrapper around pytest, that provides everything pytest needs to run tests of the PSS (see *ProTest - the PSS Product Testing Framework*). If you've followed the steps in *Installing ProTest* then a protest executable should be in your system path. Test this running

```
protest -h
```

This should produce the following output.

```
usage: protest [-h] [-H] [-p PATH] [-i INCLUDE [INCLUDE ...]] [-e EXCLUDE [EXCLUDE ...]]
               [--cache CACHE] [--outdir OUTDIR] [--keep] [--reduce]

Run PSS Product Tests

options:
 -h, --help            show this help message and exit
 -H, --show_help       Show detailed help on test options
 -p PATH, --path PATH  Path to cheetah build tree
 -i INCLUDE [INCLUDE ...], --include INCLUDE [INCLUDE ...]
                       Include the following test types (def=product)
 -e EXCLUDE [EXCLUDE ...], --exclude EXCLUDE [EXCLUDE ...]
                       Exclude the following test types
 --cache CACHE         Directory containing locally stored test vectors
 --outdir OUTDIR       Directory to store candidate data products
 --keep                Preserve the post-test data products (e.g, candidates, cheetah␣
↪logs, configs, etc)
 --reduce              Store only header information from SPS candidate filterbanks
```

ProTest takes a number of optional arguments. The first (-p PATH) is the path to a cheetah build. This can be either of the following

- a build tree (generated by running *make*)
- a bin directory (which contains the cheetah executables in one place, generated by running *make install*).

ProTest will determine which of these options you are providing under the hood.

If `-p` is not provided, ProTest will assume you have the cheetah executable(s) in your system $PATH and will look there for the relevant one for each of its tests. If it cannot find cheetah in your $PATH, then an exception will be raised and ProTest will exit.

The `-i` INCLUDE option tells ProTest which group(s) of tests you wish to run according to how they are *marked*. ProTest tests PSS pipelines which rely on different processing resources that may not necessarily be available on all hardware that it runs on. For example, tests of PSS pipelines which required GPUs will not execute successfully on a machine which only has CPUs available. Furthermore, as a tester, you may only wish to test one pipeline type (e.g.,

Single Pulse Search pipelines but not Acceleration Search pipelines). You may wish to run tests only associated with SKA-LOW but not SKA-MID. For this reason we utilise the marker functionality provided by pytest. Each test is *decorated* with a set of markers which classify tests into groups, thereby allowing the tester to run only the subset of tests that are of interest to them. The user can specify as many markers as required to enable a specific combination of test types.

If `-i` is not provided, ProTest will execute all of the tests that it has (as specified by the *product* marker). Available markers can be found in pytest.ini or by running *protest -H*

The `-e` EXCLUDE option tells ProTest which group(s) of tests should not be executed. The user can specify as many markers as required to exclude a specific combination of test types.

`--cache` is a path to a local repository of PSS test vectors. This may be an existing cache, or a new cache into which new vectors will be pulled and stored. If the vectors required by the tests are not found in the cache directory provided on the command line, then they will be downloaded from the PSS test vector server. –cache is an optional argument. If it is not provided, ProTest will create and use a directory in the user's $HOME area. ProTest will check there is sufficient space on the drive to download a test vector before doing so.

`--outdir` is the path to a directory into which data products which result from the execution of cheetah are written. This could include lists of candidate metadata, filterbank files, folded archives, etc. For each test, a directory with a randomly generated name is created under the directory specified by –outdir, and the results of that test are written to that directory. If –outdir is not provided, then ProTest will use "/tmp" to store cheetah data products.

`--keep` allows the output data products that are created by test runs (written to <outdir>) to be preserved. By default ProTest completely cleans up after itself leaving no files behind. However, for diagnostic puporses, one may wish to keep configuration files, candidates, etc.

`--reduce` is used to prevent candidate files (e.g., filterbanks) from occupying large amounts of disk space (i.e., in CI pipelines). If `--reduce` is enabled, SPS candidate filterbanks are removed, even if `--keep` is enabled, and are replaced by a json file (placed in <outdir>) that contains information from the header of each filterbank.

To demonstrate, let's run all of the available SPS (Single Pulse Search) tests. We'll use a build tree in this case, and we will leave –outdir and –cache empty.

```
protest -i sps -p <path/to/cheetah/build_dir>/cheetah/
```

For clarity the <build_dir> refers to the directory in which *make* was executed, whereafter a subdirectory called *cheetah* is generated. At the time of writing, this command will produce the following output

```
 Running pytest -m sps -c /home/bshaw/.venvs/protest/lib/python3.8/site-packages/ska_pss_
↪protest/pytest.ini --path=/raid/bshaw/cheetah_builds/release_basic_cheetah_panda/
↪cheetah /home/bshaw/.venvs/protest/lib/python3.8/site-packages/ska_pss_protest
============================================================================================␣
↪test session starts␣
↪============================================================================================
platform linux -- Python 3.8.13, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/bshaw/.venvs/protest/lib/python3.8/site-packages/ska_pss_protest
configfile: pytest.ini
plugins: repeat-0.9.1, mock-3.11.1, metadata-3.0.0, bdd-6.1.1, html-3.2.0
collected 3 items / 2 deselected / 1 selected

../../../../home/bshaw/.venvs/protest/lib/python3.8/site-packages/ska_pss_protest/test_
↪sps_emulator.py::test_detecting_fake_single_pulses
--------------------------------------------------------------------------------------------
↪--------- live log call ----------------------------------------------------------------
↪-----------------------------------
INFO     root:requester.py:132 Cache location: /home/bshaw/.cache/SKA/test_vectors
```

(continues on next page)

(continued from previous page)

```
INFO     root:requester.py:195 SPS-MID_747e95f_0.2_0.0002_2950.0_0.0_Gaussian_50.0_
↪123123123.fil in local cache
INFO     root:_config.py:138 Located cheetah executable: /raid/bshaw/cheetah_builds/
↪release_basic_cheetah_panda/cheetah/pipelines/search_pipeline/cheetah_pipeline
INFO     root:pipeline.py:144 Command is: /raid/bshaw/cheetah_builds/release_basic_
↪cheetah_panda/cheetah/pipelines/search_pipeline/cheetah_pipeline --config=/tmp/
↪yrkajb0u -p SinglePulse -s sigproc
INFO     root:pipeline.py:171 Return code is: 0
INFO     root:candlist.py:158 Detected candidates found at: /tmp/tmprr0_aij7/2012_03_14_
↪00:00:00.spccl
INFO     root:candlist.py:173 Located 60 candidates
PASSED                                                                                ␣
↪                                                                                     ␣
↪                              [100%]

=============================================================================================␣
↪1 passed, 2 deselected in 49.42s␣
↪=============================================================================================
```

CHAPTER

THREE

WRITING NEW PRODUCT TESTS

Product tests can be written in any form that is understandable and executable by pytest. The current preferred approach is to adopt pytest-bdd. This is a pytest plugin which faciliates the use of the Gherkin standard to provide natural language descriptions of product tests. For general guidance for working with BDD tests for SKA projects see this link.

The first step in the development of a product test is to encode the behaviour that we wish to test, using the Gherkin format, into a **feature file**. These files should be placed in the src/ska-pss-protest/features directory, where currently implemented examples can be found. Tests should be marked on the first line using as many markers as are required to describe the test type. For example, a product test which executes a SPS pipeline which requires cuda would be tagged as follows:

```
@product @cuda @sps
```

A list of valid markers can be found by running

```
protest -H
```

or in pytest.ini. New markers may be declared in this file as required.

The test itself should be written in a separate file in src/ska-pss-protest. In order for the test to be discovered by ProTest, the file name must be in the form **test_<some_description>.py**. Should any new naming conventions be required, these too must be declared in pytest.ini.

To import the required BDD functionality required to write the test, import the following.

```
import pytest
from pytest_bdd import given, parsers, scenarios, then, when
```

To import ProTest, do

```
import ska_pss_protest
```

The test must then be linked to the feature file that was written in the previous step using the **scenarios** method as follows:

```
scenarios("path/to/feature/ticket.feature")
```

Following this, each Gherkin step described in the feature ticket can be implemented as a test function. An example of the first *given* step might look like this…

```
@given(parsers.parse("Some initial condition"))
def some_function():
    """
```

(continues on next page)

(continued from previous page)

```
    Write test code
    """
```

As variables cannot automatically be passed between test stages (i.e., a variable declared in the *given* function cannot be accessed in any other functions), it is useful to define a context *fixture* that contains a dictionary of variables we wish to pass around the different stages, e.g.,

```
@pytest.fixture(scope="function")
def context():
    """
    Return dictionary containing variables
    to be shared between test stages
    """
    return {}
```

This dictionary can then be accessed for the purpose of added or extracting shared variables in the function declaration of the step that wishes to use it, for example,

```
@given(parsers.parse("Some initial condition"))
def some_function(context):
    """
    shared_variable = context["<key>"] # access existing variable

    context["<key>"] = new_variable # Create a new shared variable
    """
```

3.1 Accessing command line arguments

ProTest allows a number of command line arguments to provide inputs that should be shared amongst all tests that it will execute. For example, to instruct ProTest to use a set of locally stored test vectors, the user would run,

```
protest --cache <path/to/cache/dir> .....
```

ProTest passes these to the tests, where required, via conftest.py which provides a pytestconfig fixture that contains the value of the argument. This is passed to a test function in the same way as the fixture described above. For example, to access the cache directory that we pass in at the command line, we would write…

```
@given(parsers.parse("Some initial condition"))
def some_function(pytestconfig):
    """
    cache_dir = pytestconfig.getoption("cache")
    """
```

3.2 Test execution

Tests can be executed as part of test development, assuming no default parameters are overridden (i.e., that the command line arguments are set to their default values, see *conftest.py*) simply by running

```
pytest /path/to/test.py
```

but to ensure that they run as part of ProTest, it's safest to update your local install of ProTest, to include your new tests. From the package root directory, run

```
poetry install
```

and then ProTest can be executed in the usual way

```
protest -i <marker> --cache </path/to/cache> --path </path/to/cheetah/build> --outdir </
↪path/to/output/directory>
```

Detailed instruction on how to run ProTest product tests can be found in *Running PSS product tests with ProTest*

CHAPTER

FOUR

CONTRIBUTING TO PROTEST

ProTest comprises a number of libraries that are intended to facilitate the development of product tests. The libraries are located at src/ska-pss-protest and each is repsonsible for a different piece of functionality that may be required by a test. Examples include

- The handling of test vectors - how they are provisioned from a remote server, and stored/accessed in the local cache.
- The preparation and execution of cheetah with all of its required input
- The extraction of header parameters from filterbank files
- The verification of candidate filterbanks
- The verification of candidate metadata

As ProTest (and cheetah) grows, more and more libraries will be required to verify PSS data products. New libraries can be added to ProTest by simply adding them to src/ska-pss-protest. New libraries and the classes within them should be added to init.py as follows

```
from ska_pss_protest.<new_library> import <new_class_a>, <new_class_b>
```

There are no specific naming conventions currently required of new libraries but they should at least succinctly describe what the library does. Each library must naturally be accompanied by a test of unit tests and these should be placed in tests/. Like product tests, unit test are tagged with markers which describe the library that they are testing. This allows the developer to execute their own new tests during development of their library. New markers for unit tests should be declared in pytest.ini. To execute unit tests of a specific library, one can run

```
pytest -m <marker> tests/
```

or to run all tests

```
make python-test
```