Instructions for test_fault.py

Abdullah Alsafar

August 2025

Contents

1 Notes

1. The **test_fault.py** file can be found here:
   https://github.com/AbdullahAGit/microgrid-fault-test

2. It is meant to be used with the Typhoon HIL Schematic named
   "microgrid_Data generation.tse"

3. Please feel free to contact me at abdullah.alsafar@torontomu.ca if you have any questions

2 Introduction

Typhoon HIL provides an testing software "TyphoonTest" which can be used in conjunction with Typhoon schematic files. TyphoonTest uses a mix of the pytest framework as well as their own APIs which has documentation and can be found at:

- https://www.typhoon-hil.com/documentation/typhoon-hil-api-documentation/typhoon_api.html
- https://www.typhoon-hil.com/documentation/typhoon-hil-typhoontest-library/

These links are also available in the "Help" section on the Typhoon HIL Control Center.

This document will serve as an explanation for some beginner pytest features such as scopes, fixtures and test cases as well as go in to detail about how the **test_fault.py** file works. For more information on Typhoon HIL and TyphoonTest, the learning modules provided by HIL Academy are an excellent source.

You can also find the pytest documentation here:

- https://docs.pytest.org/en/stable/index.html

3 Pytest Basics

The **test_fault.py** file makes use of 2 simple pytest features: test cases and fixtures.

- Test cases are usually parameterized and dictate what happens while the test is occuring. In the TyphoonTest IDE, the runtime of tests is shown under the "Test" Section.
- Fixtures on the other hand are commonly used to setup and teardown i.e before a test starts and after a test finishes. This, in the context of Typhoon HIL, can include things such as loading the model, starting the simulation, setting fixed variables (that is, they won't change in the middle of a test) and stopping the simulation.

Fixtures contain a scope which can define when you want them to run. For example, a fixture with the *'function'* scope occurs for each and every test case. However, a *'module'* scope occurs only once for a module which usually means that the fixture setup occurs before the first test and the fixture teardown occurs after the last test.

Fixtures are activated upon being called by other tests in their parameters. In fact, fixtures can even be by called other fixtures in their parameters, which is useful because you can "chain" fixtures together, as in:

- The test calls a fixture that runs for every test with the *function* scope, and that fixture calls another fixture that once for all tests with the *module* scope.

4 test_fault.py

4.1 load_model Fixture

For test_fault.py, there are 4 fixtures and 1 test case. All 4 fixtures have the *'function'* scope and so they occur for each and every test. The **load_model** fixture is responsible for:

1. Loading the schematic model .tse file
2. Setting the fault_resistance and fault_type property values on the enabled faults
   - Some fault types require a connection to GND and some others do not. So, the fixture is also responsible for connecting the enabled faults to GND if required, which means creating a "GND" object near the position of the fault in the schematic. This is what these lines are for:

```
if "GND" in set_fault_type:
    fault_gnd = mdl.create_component(
            'core/Ground',
            name='fault_gnd',
            position = mdl.get_position(faultHandle)
            )
    mdl.create_connection(mdl.term(faultHandle, 'GND'), mdl.term(fault_gnd, 'node'))
```

3. Saving the changes and compiling the model
4. Loading the compiled .cpd model
5. Setting all fixed inputs such as:
   - The operation mode of the Diesel Genset and Battery ESS
   - The average, max and min wind speed for the Wind Turbine
   - The grid vrms_cmd and grid freq_cmd to 1
   - The rates of change for all subsystems
   - Enabling all subsystems and microgrid controller
6. And finally, starting the simulation and, during fixture teardown, stopping the simulation

All of the steps listed occur during the "setup" stage of the fixture, except for stopping the simulation, which occurs during the "teardown" stage of the fixture which can be seen here:

```
hil.start_simulation()

yield
#Fixture Teardown

hil.stop_simulation()
```

As a note, the *yield* keyword determines the boundary between the setup and teardown of a fixture. However, it can also return various variables which are helpful when it is called by other fixtures and tests, which will be a vital detail later.

4.2 test_faults Test Case

The test_faults test case has 3 parameters:

- **set_fault_resistance**
- **set_fault_type**
- **set_fault**

Other values can be added to **set_fault_resistance** but for **set_fault_type** the fault type must be available in the drop-down menu for faults in the schematic. For **set_fault** the fault name must be an existing fault in the schematic.

For these parameters, every unique combination is run by **test_faults** The order is decided by the order they are called by the **test_faults** function:

```
def test_faults(load_model, set_fault_resistance, set_fault_type, set_fault):
```

According to the order, all resistances will be tested for each fault type, and all fault types will be tested for each fault that has been enabled. Therefore, while testing, the next fault will only begin when all other combinations for the test before that have ran.

At the beginning of the test, there are many print statements. These are present just for testing purposes and ensuring the desired output is achieved. Because the fixture setup has already ran by this point, the simulation is on.

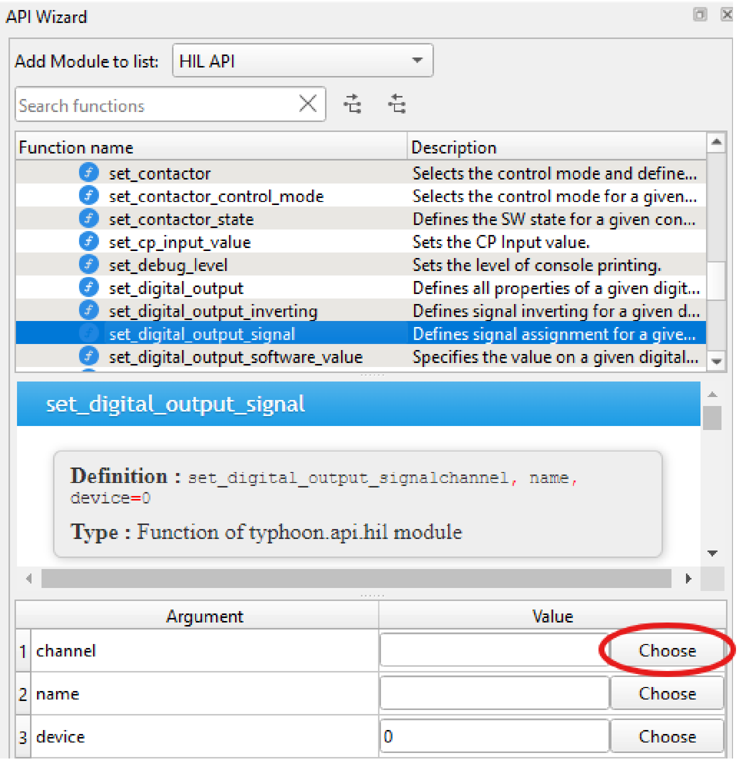
Before beginning capturing all desired signals, the simulation waits 20 seconds. This is important because the specific model **microgrid_Data generation.tse** requires a little less than 20 seconds to reach steady state. After the 20 seconds are finished, the capture begins.

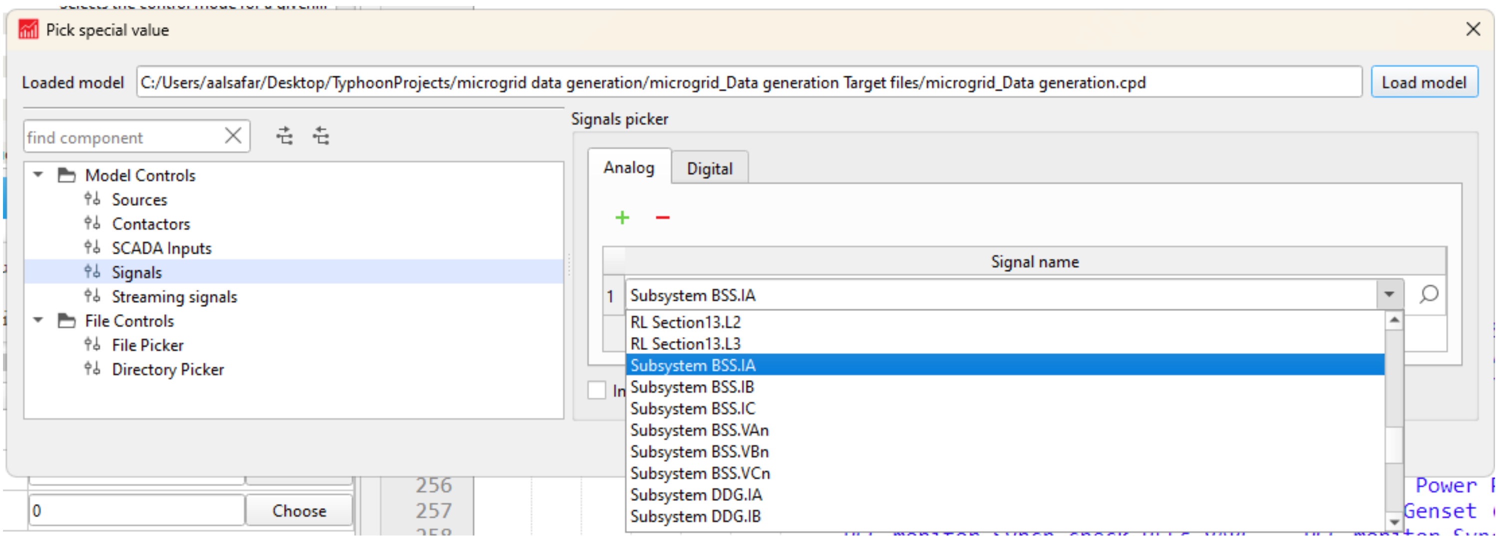
```
    cap_duration = 1
    time_before_fault = cap_duration/2
    cap.start_capture(duration=cap_duration,
                      rate=fs,
                      signals=[

                          'Grid1.Vc', 'Grid1.Vb', 'Grid1.Va',
                          'PCC_monitor.S1_fb',
                          'Grid1.Ic', 'Grid1.Ib', 'Grid1.Ia',
                          'Subsystem BSS.IA', 'Subsystem BSS.IB', 'Subsystem BSS.IC',
                          'Subsystem WT-B.IA', 'Subsystem WT-B.IB', 'Subsystem WT-B.IC',
                          'Subsystem WT-B.VAn', 'Subsystem WT-B.VBn', 'Subsystem WT-B.VCn',
                          'PCC_monitor.Va', 'PCC_monitor.Vb', 'PCC_monitor.Vc',
                          'PCC_monitor.VA', 'PCC_monitor.VB', 'PCC_monitor.VC',
                          'Grid UI1.Vrms_meas_kV', 'Grid UI1.Qmeas_kVAr', 'Grid UI1.Pmeas_kW',
                          'Wind Power Plant (Generic) UI1.Pmeas_kW', 'PV Power Plant (Generic) UI1.Pmeas_kW',
                          'Battery ESS (Generic) UI1.Pmeas_kW', 'Diesel Genset (Generic) UI1.Pmeas_kW',
                          'PCC_monitor.Synch_check.PLLs.VABC', 'PCC_monitor.Synch_check.PLLs.Vabc'
                          #'Wind Power Plant (Generic) UI1.wind_speed_m_per_s',
                          #'Wind Power Plant (Generic) UI1.MCB_status', 'PV Power Plant (Generic) UI1.MCB_status',
                          #'Diesel Genset (Generic) UI1.MCB_status', 'Battery ESS (Generic) UI1.MCB_status',
                      ],)
```

- The *cap_duration* variable defines how long the capture lasts for in seconds. In this case, it is 1 second.
- The *time_before_fault* variable defines when the selected fault is enabled. In this case, it is halfway after the fault begins and so the fault occurs at 0.5 seconds.
- The *rate* defines the frequency at which the selected signals are sampled. The variable fs was set at the beginning of the script to be 100e3 and so the signals are sampled every 1/100e3 or 0.00001 seconds. Increase or decrease this number to change the resolution of the capture.
- The *signals* list defines all signals that are to be captured. The signals in the list must have the correct schematic names otherwise the capture returns an error.

Note:

- An easy way to find the correct name for signals is by using the API Wizard's "Choose" button on a value after selecting any API function. The API Wizard is also helpful for inserting other API functions like *set_contactor* and *start_capture*. A tutorial on how to use the API Wizard and other TyphoonTest features is available on HIL Academy.





All signals are listed in this drop down menu, which is helpful to find various signal names for the capture.

The selected fault is then enabled halfway through the capture and the capture results are taken after it is finished.

```
    cap.wait(secs=time_before_fault)

    hil.set_contactor(name=f'{fault}.enable',
                          swControl=True,
                          swState=True,
                          )

    df = cap.get_capture_results(wait_capture=True)
```

- The get_capture_results function is responsible for returning the capture results as a Pandas DataFrame. This is useful incase we want to do anything else with the signals, such as outputting them as a .csv file.

At this point, all the signals that were included in the capture function are available to look at over the 1 second period. Pytest uses the Allure framework to create html reports of tests after they are down. It is possible to look under get_capture_results of the html report to see all the captured signals and their respective graphs.

However, there is an additional function solely for the purpose of comparing multiple signals on a single plot. This is useful for viewing 3-phase signals as well as power signals.

There is also an additional function *to.csv()* dedicated to writing csv files using any selected signal. The df.index is a different format than usual so the command *df.index.total_seconds()* is meant to format the time into only seconds.

4.3 Setter Fixtures and Indirect Parameterization

You may have noticed that there are no setup and teardown lines for the other 3 fixtures that have not been discussed yet:

- **set_fault_type**
- **set_fault**
- **set_fault_resistance**

These fixtures also have the same name as the name of the parameters in the **test_faults** test case. The reason these fixtures exist are to solve a major problem within the test script.

By the time the **test_faults** test case begins, the **load_model** fixture setup has already finished. As explained earlier, the **load_model** fixture is responsible for setting the fault type and resistance. However, the type and resistance parameter values are set when **test_faults** begins, so how can the **load_model** fixture know which fault type and fault resistance to set if it runs beforehand? Furthermore, how does the fixture know which fault to change these values for? The answer lies within the use of Indirect Parameterization.

- Indirect Parameterization allows for a fixture to receive a parameter before then passing them to the test.
- The only caveat is that the fixture and parameter require the same name.

As explained earlier, fixtures can be called by other fixtures and also return values, so the first 3 fixtures return their values in the *yield* section of the fixture, to the **load_model** fixture, which is then called by **test_faults**.

```
@pytest.fixture(scope='function')
def set_fault_type(request):
#Fixture Setup

    yield request.param
#Fixture Teardown

    pass
```

- The request variable here essentially tells the fixture that the parameter
- Unlike setting fault resistances and types, the contactor on the selected fault is enabled during the simulation and thus during the test case. Because Indirect Parameterization allows for the parameter to get sent back to the test after it is sent to the fixture, the name of the fault (and all the other fixtures) to be enabled is still received during the test.

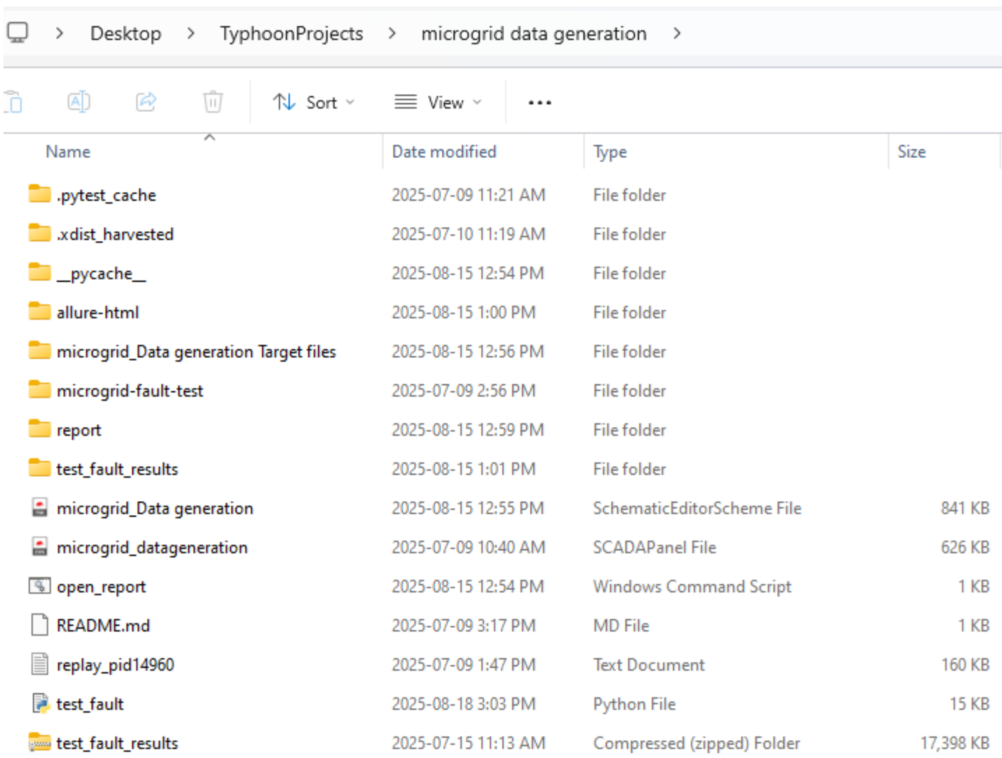
You can read more about indirect parameterization here:

- https://docs.pytest.org/en/stable/example/parametrize.html#indirect-parametrization

5 Allure Reports and Working Directory

Once the test is finished, a .cmd script called *open_report* runs automatically and generates a report .html which opens on your browser. The report includes all test cases and their distinct steps, such as starting the simulation or gathering capture results. As explained earlier, the script can be modified so that the report includes extra graphs. An example of this is the extra plots which overlay different signals together.

It is also possible to manually open *open_report* which can be found in the working directory, assuming a test case has run once before. The working directory should include the **test_fault.py** file as well as the schematic file:



> Desktop > TyphoonProjects > microgrid data generation >

Sort View ...

Name	Date modified	Type	Size
.pytest_cache	2025-07-09 11:21 AM	File folder	
.xdist_harvested	2025-07-10 11:19 AM	File folder	
__pycache__	2025-08-15 12:54 PM	File folder	
allure-html	2025-08-15 1:00 PM	File folder	
microgrid_Data generation Target files	2025-08-15 12:56 PM	File folder	
microgrid-fault-test	2025-07-09 2:56 PM	File folder	
report	2025-08-15 12:59 PM	File folder	
test_fault_results	2025-08-15 1:01 PM	File folder	
microgrid_Data generation	2025-08-15 12:55 PM	SchematicEditorScheme File	841 KB
microgrid_datageneration	2025-07-09 10:40 AM	SCADAPanel File	626 KB
open_report	2025-08-15 12:54 PM	Windows Command Script	1 KB
README.md	2025-07-09 3:17 PM	MD File	1 KB
replay_pid14960	2025-07-09 1:47 PM	Text Document	160 KB
test_fault	2025-08-18 3:03 PM	Python File	15 KB
test_fault_results	2025-07-15 11:13 AM	Compressed (zipped) Folder	17,398 KB

Running *open_report* before the script has stopped running generates a report that only includes the test cases up to the last completed one. *open_report* can also open the most recently generated report, as long as it has not been override by running the script again.