Scenario Builder

The scenario builder is a Python tool allowing to build scenarios programmatically in Python and generate JSON files that can be imported into a project of an existing OpenBACH installation.

Please refer to the Introduction to OpenBACH-extra and OpenBACH API page to see how to install and set up your platform to use the API described in this page.

In summary, the scenario builder allows to create a `Scenario` capable of holding, initializing and configuring the OpenBACH functions:

- Start Job Instance
- Start Scenario Instance
- Stop Job Instance
- Stop Scenario Instance
- If
- While
- Push File

Scenario

Contruction

An OpenBACH scenario can be built using the `Scenario` class.

You need to provide a name for your scenario, and an optional description:

```
DESCRIPTION = """Simple scenario

This simple scenario example aims at showing the capabilities
of the scenario_builder API.
"""
scenario = Scenario('tutorial', DESCRIPTION)
```

Managing content

You can add or remove arguments, constants or OpenBACH functions to your scenario:

- Arguments

```
scenario.add_argument(argument_name, argument_description)
scenario.add_arguments(**arguments)
scenario.remove_argument(argument_name)
scenario.remove_arguments(*arguments_names)
```

Pretty straightforward, the `add_arguments` version uses keyword arguments as `argument_name=argument_description` to call the `add_argument` version.

Arguments are placeholders that can be used as values for OpenBACH functions whose real value

won't be known until the scenario is started.

- Constants

```
scenario.add_constant(constant_name, constant_value)
scenario.add_constants(**constants)
scenario.remove_constant(constant_name)
scenario.remove_constants(*constants_names)
```

Similar to arguments, the `add_constants` version uses keyword arguments as `constant_name=constant_value` to call the `add_constant` version.

Constants are placeholders that can be used as values for OpenBACH functions whose value is already defined. Useful to factor in values common to several OpenBACH functions and change it in a single place if need be.

- OpenBACH functions

```
scenario.add_function(…)
scenario.remove_function(openbach_function_instance)
```

Defining and configuring OpenBACH functions is explained in details later in this guide. Removing OpenBACH functions must be done using the object returned by the `add_function` call.

Exporting to OpenBACH

Once your scenario is ready, you can call the `build` method to get a dictionary representation of this scenario. This dictionary is suitable to be written in a file as JSON data ready to be imported into OpenBACH.

```
with open('tutorial.json', 'w') as scenario_file:
    json.dump(scenario.build(), scenario_file)
```

As a shortcut, the `write` method does exactly that; so this is equivalent:

```
scenario.write('tutorial.json')
```

It is up to you to then send your JSON file to OpenBACH using either the auditorium scripts or the web interface.

Utilities

The `extract_function_id` method will let you retrieve OpenBACH function's paths from their names.

```
scenario.extract_function_id(*job_names, include_subscenarios=False, **filtered_jobs)
```

It is a generator that will yield paths corresponding to all the given *job_names*; these paths are suitable to use as *jobs* arguments to post-processing jobs, for instance. The keyword arguments should assign a filtering function to a job name. Once a matching *start_job_instance* OpenBACH

function is found by its name, it is passed as the sole argument to the filtering function which should return a boolean value indicating if this job should be considered or not. This allows to discriminate job by their configuration parameters (think *iperf* client vs. *iperf* server), for instance.

The `find_openbach_function` will return an `OpenBachFunction` instance for one of these paths.

OpenBACH Functions

Initialization

In order to initialize the OpenBACH function:

```
name_of_function = scenario.add_function(type_of_function, wait_delay=wait_delay,
wait_launched=[...], wait_finished=[...])
```

The *type_of_function* is the only mandatory parameter, and the possible values are:

- *start_job_instance*
- *stop_scenario_instance*
- *start_job_instance*
- *stop_scenario_instance*
- *if*
- *while*
- *push_file*

Wait_launched, wait_finished and wait_delay are used for the scheduling conditions (as in the web interface case):

- *wait_launched*: a list of other OpenBACH functions that must be completed before this one starts (defaults to None);
- *wait_finished*: a list of other *start_job_instance* or *start_scenario_instance* OpenBACH functions; each job/scenario started by these functions must reach completion before this function starts (defaults to None);
- *wait_delay*: an amount of time, in seconds, to wait with respect to the *wait_launched/wait_finished* functions, before executing its action (defaults to 0.0).

Configuration

In order to configure an OpenBACH function, you need to call its `configure` method. Each type of function has its own `configure` parameters:

- If *start_job_instance*:

```
name_of_function.configure(job_name, entity_name, offset=0, **parameters_of_job)
```

This function will start the OpenBACH job *job_name* on the agent associated to the entity *entity_name* *offset* seconds after the OpenBACH function is scheduled. The job will be configured with the values of the parameters provided in the *parameters_of_job* keyword arguments.

Note that when configuring a job whose accept sub-commands, you need to provide a dictionary to be able to configure the parameters of the sub-command. Example: `iperf3_srv.configure('iperf3', srv_entity, offset=0, interval=1.0, server={'exit':True, 'bind':srv_ip})`. If you want to select a sub-command without parameters, provide an empty dictionary to the sub-command choice.

- If *start_scenario_instance*:

```
name_of_function.configure(scenario_name, **arguments_of_scenario)
```

This function will start the OpenBACH scenario *scenario_name* when it is scheduled. Extra arguments provided as keyword arguments will be passed to the launched scenario.

scenario_name can be a string to start an existing scenario in your OpenBACH project; or a `Scenario` instance to configure a sub-scenario alongside the one holding this OpenBACH function.

Note that using a `Scenario` instance will allow you to navigate through them all using the `Scenario.subscenarios` iterator on your main scenario. This could help `build`ing them all at once, for instance.

- If *stop_job_instance*:

```
name_of_function.configure(*functions_to_stop)
```

This function will stop the provided OpenBACH functions as soon as it is scheduled; granted they are of type *start_job_instance*.

- If *stop_scenario_instance*:

```
name_of_function.configure(scenario_to_stop)
```

This function will stop the provided OpenBACH function as soon as it is scheduled; granted it is of type *start_scenario_instance*.

- If *if*:

```
name_of_function.configure(condition)
name_of_function.configure_if_true(*openbach_functions)
name_of_function.configure_if_false(*openbach_functions)
```

This function will evaluate the *condition* when it is scheduled and launch one or the others configured OpenBACH functions depending on the result of this evaluation.

Note that only the `configure` call is mandatory.

- If *while*:

```
name_of_function.configure(condition)
name_of_function.configure_while_body(*openbach_functions)
name_of_function.configure_while_end(*openbach_functions)
```

As for the *if* OpenBACH function, only the `configure` call is mandatory. As expected, the OpenBACH functions of the body will be executed as long as the condition holds true; the *end* functions will always be evaluated once the condition turns false.

- If *push_file*:

```
name_of_function.configure(entity_name, path_on_the_controller, path_on_the_agent, users=(),
groups=())
```

When this OpenBACH function is scheduled, the file referenced by *path_on_the_controller* will be copied to *path_on_the_agent* on the agent associated to the entity *entity_name*. You can provide a single string for each parameters or a list of strings to copy several files at once. The length of both lists must match.

If provided, the *users* and *groups* parameters will control ownership of the files on the agent; their length must also match with the other parameters.

Example

This example creates and returns a scenario containing one argument, *ip_dst*, and three functions:

- *hping_function*, that starts the job *hping* to ping *$ip_dst*
- *fping_function*, that starts the job *fping* to ping *$ip_dst*
- *stop_function*, that stops *hping_function* and *fping_function* 20s after they started

example.py

```
#Import the Scenario class from scenario_builder and the ScenarioObserver from the
auditorium-scripts.
from scenario_builder import Scenario

def build_delay_scenario(client, server, scenario_name):
    scenario = Scenario(scenario_name, 'Comparison of 2 types of RTT measurements')
    scenario.add_argument('ip_dst', 'Target of the pings and server ip adress')

    hping_function = scenario.add_function('start_job_instance')
    hping_function.configure('hping', client, offset=0, destination_ip='$ip_dst')
    fping_function = scenario.add_function('start_job_instance')
    fping_function.configure('fping', client, offset=0, destination_ip='$ip_dst')
    stop_function = scenario.add_function('stop_job_instance',
wait_launched=[hping_function, fping_function], wait_delay=20)
    stop_function.configure(hping_function, fping_function)
    return scenario
```