ROAST

Release 5.0

Xilinx

Nov 09, 2022

GETTING STARTED

ROAST is an open-source Python framework that simplifies the development of complex validation test suites. To accomplish this, ROAST provides a collection of interfaces that allows test developers to build test suites in a highly structured manner.

Key features:

- Compose systems from Xilinx or custom components
- Define systems composed from various configuration sources
- Heirarchical configuration system
- Randomized data provider for randomized testing
- Generic APIs for simplified usage and access
- Plugin system for extensibility

To find out more, visit *ROAST features*.

CHAPTER

ONE

FIRST STEPS

New to test development? Learn how to establish a Python environment and a test repository for your testing requirements.

- **Installation**: *Installation*
- **Getting started**: *Environment Setup* | *Repository Structure*

1.1 Installation

1.1.1 Versions

ROAST supports Linux and requires Python 3.6+.

1.1.2 Basic Installation

The easiest way to install it is using pip:

```
$ pip install roast
```

To install the Xilinx plugin for roast:

```
$ pip install roast-xilinx
```

Optionally, roast specific pytest fixtures can be installed through a pytest plugin:

```
$ pip install pytest-roast
```

1.2 Getting Started

The objective of this tutorial is to build a test repository starting with a a basic test leveraging ROAST to generate its configuration.

- *Environment Setup*
- *Repository Structure*

1.2.1 Environment Setup

While ROAST is test runner agnostic, this tutorial will use pytest and *pytest fixtures* developed specifically for ROAST. This means both `roast` and `pytest-roast` packages must be installed. While optional, it is highly recommended that these are installed into a virtual environment.

Upon installation of `pytest-roast`, `pytest` will automatically be installed as a dependency.

Note: Please review *Installation* instructions on how to install Python packages using pip.

1.2.2 Repository Structure

Tests can be structured in a number of ways. In this tutorial, we'll start with a basic structure.

Create a repository with the following structure and files:

```
basic/
└── tests/
    ├── test_basic.py
    └── conf.py
```

conf.py

```
var = "hello world"
```

test_basic.py

```
import pytest

def test_basic(create_configuration):
    conf = create_configuration()
    assert conf["var"] == "hello world"
```

In **test_basic.py**, when `pytest` is imported, all pytest fixtures will be available. Since `pytest-roast` is installed as a pytest plugin, the `create_configuration` fixture is also available and can be added as an argument to the test. When called, the configuration is read from **conf.py** and assigned to the `conf` variable. The value of `var` is then accessed by accessing `conf["var"]`.

Note: For other methods of accessing configuration values, visit *Layered Configuration System*.

First, let's see what pytest collects as tests:

```
$ pytest --collect-only
======= test session starts ========
..
collected 1 item
<Module tests/test_basic.py>
  <Function test_basic>

====== no tests ran in 0.04s =======
```

We can now execute the test:

```
$ pytest
====== test session starts =======
..
collected 1 item

tests/test_basic.py .       [100%]

======= 1 passed in 0.18s ========
```

Note: Visit *Complex Repository Structures* for advanced parameterized and categorized testing scenarios.

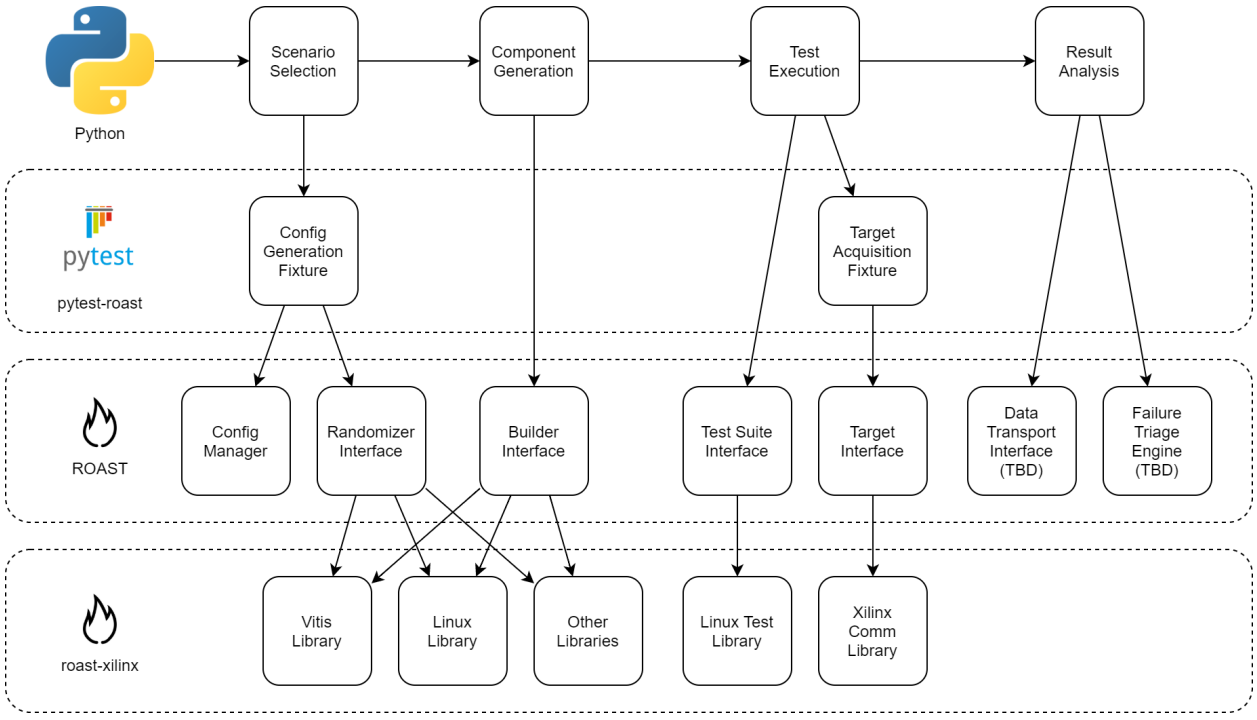
1.3 ROAST features

ROAST provides a rich feature set to assist developer in accelerating test development in areas of the test flow.

ROAST consists of three packages:

- **roast**: Core functionality that includes high-level interfaces, plugin framework, configuration management, and more
- **roast-xilinx (recommended)**: plugin for roast that includes Xilinx specific functionality such as board acquisition, PetaLinux building, Vitis wrapper, and more.
- **pytest-roast (optional)**: plugin for pytest that includes pytest fixtures for accelerated test development when using pytest as a test runner. Features includes board automation, scenario generation, and configuration generation.

Example test flow with package breakdown:



1.3.1 Define systems composed from various configuration sources

System properties can be defined in multiple file formats through the ROAST configuration management system. This allows for maximum flexibility where some configuration parameters may have been generated from another tool or users are more familiar with a particular format.

Learn more about the *Layered Configuration System*.

1.3.2 Compose systems from Xilinx or custom components

A full range of Xilinx component libraries are available in the `roast-xilinx` package. These include:

- PetaLinux
- Linux
- Xilinx Software Command-Line Tool (XSCT)
- Xilinx System Debugger (XSDB)
- JTAG, Linux boot
- ATF
- U-Boot
- SD
- AIE
- Cross-Compile
- CMake

Custom components can be created and registered within the `roast` namespace.

Learn more about *Component Plugin Models* and *Component Plugins*.

1.3.3 Configuration can be specified or command line overridden

Configuration values can have interpolation expressions that allows parameters to be dynamically changed. The replacement values can be established in another configuration file or through the command line. This provides test repositories a way to signifcantly scale with unlimited test variants.

Learn more about *String Interpolation*, *Complex Repository Structures* and *Pytest Fixtures and Options*.

1.3.4 Generate data for randomized testing

ROAST includes a data provider (randomization engine) to expand coverage in tests beyond static values. A database of parameters and possible values can be established where randomized values can be generated. Constraints (excluded values and range limits) can be defined to shape returned values.

The provider includes various methods to generate randomized data and to add custom randomization algorithms.

Learn more about the *Data Provider*.

CHAPTER

TWO

ROAST FEATURE OVERVIEW

- **Scaling test suites through configuration**: *Layered Configuration System* | *Complex Repository Structures*
- **Test execution with pytest**: *Pytest Fixtures and Options*
- **Randomization**: *Data Provider*

2.1 Layered Configuration System

ROAST uses a hierarchical or layered configuration system that is based on the open source library python-configuration.

This allows for configuration to be defined through a single file with key/value pairs or through a hierarchical template-based approach. In the template-based approach, a master configuration is established with interpolation expressions (placeholders) to be replaced with the corresponding value established in another configuration file through a process called string interpolation.

Configuration files must be named **conf** and can be written in a number of format types including:

- Python (conf.py)
- YAML (conf.yaml)
- TOML (conf.toml)
- JSON (conf.json)
- INI (conf.ini)

See also:

Getting Started tutorial for examples of project structures.

2.1.1 Single File

In the single file approach, configuration can be defined through a file (**conf.py**).

```
var = "my_string"
```

2.1.2 Template-Based (Heirarchical)

In the template-based approach, a variable can be defined with its value to be substituted with another variable's value. The top-level configuation file (**conf.py**):

```
var = {string_var}
```

At the test specific level, the variable and its value can be defined for a specific test or category of tests. The test specific configuration file (**specific/conf.py**):

```
string_var = "my_string"
```

When both files are loaded in the library, the test specific configuration file is "layered" on top of the template. The Python functionality behind this is str.format.

> **Warning:** For Python configuration files, string interpolation is only supported in iterables.

2.1.3 Dot-Based Variables

Dot-based variables can be created from configuration files.

Python

A separator of `__` is used to represent a `.`. For example:

```
aa__bb__cc = 1
```

would result in the configuration

```
{
    'aa.bb.c': 1,
}
```

An alternative is to use `python-box`. For example:

```
from box import Box
aa = Box(default_box=True)
aa.bb.cc = 1

del Box
```

> **Note:** If this method is used, the last statement of `del Box` should be included. Otherwise, a key/value pair of "Box" and a `Box` class object will appear in the configuration.

> **Warning:** String interpolation is only supported in iterables and not dictionaries.

Other Formats

With other formats, if heirarchy is part of the specification, dot-based variables will be created. For example, a TOML configuration:

```
[section]
var = "mystring"
```

would result in the configuration

```
{
    'section.var': 'mystring'
}
```

2.1.4 Accessing Configuration Values

When creating a configuration, the `ConfigurationSet` object that is returned can be accessed like a dictionary. If the configuration is assigned to `conf`, the value of `var` can be obtained using:

```
conf.get("var")
conf["var"]
conf.var
```

Similarly, dot-based variable values can be obtained using:

```
conf.get("section.var")
conf["section.var"]
conf.section.var
```

2.2 Complex Repository Structures

In this tutorial, we're going to greatly expand the repository to accomodate for test variations and expand to complex parameterized and categoried tests.

2.2.1 Parameterized

Now we're going to parameterize the test. Create with the following structure and files:

```
parameterized/
└── tests/
    ├── parameter1/
    |   └── conf.py
    ├── parameter2/
    |   └── conf.py
    ├── test_parameterized.py
    └── conf.py
```

conf.py

```
var = "{hello_world}"
```

Here, we are defining a variable `var` with an interpolation expression expecting another variable named `hello_world` to replace its value.

parameter1/conf.py

```
hello_world = "hello parameter 1"
```

When this configuration file is layered onto the first, the value of both `hello_world` and `var` is `"hello parameter 1"`.

parameter2/conf.py

```
hello_world = "hello parameter 2"
```

The value of `hello_world` and `var` is `"hello parameter 2"`.

test_parameterized.py

```
import pytest
from collections import namedtuple

Properties = namedtuple("Properties", ["parameter", "expected"])

def get_test_properties():
    p1 = Properties("parameter1", "hello parameter 1")
    p2 = Properties("parameter2", "hello parameter 2")
    return [p1, p2]

@pytest.mark.parametrize("properties", get_test_properties())
def test_parameterized(properties, create_configuration):
    conf = create_configuration(test_name="", params=[properties.parameter])
    assert conf.var == properties.expected
```

In **test_parameterized.py**, we have defined a test named `test_parameterized()`. The `@pytest.mark.parametrize` decorator defines that the value of the `properties` variable will have its value determined by the output of the `get_test_properties()` function for each iteration of the test. When this is passed into the `params` parameter of the `create_configuration` pytest fixture as a list, the additional configuration files from the `params` directory are retrieved for that iteration.

Note: The `params` parameter is a list to allow additional depths of directories. For this tutorial, we have a depth of 1.

The `test_name` parameter is set to empty string `""`. The *Categorized* section will describe this in further detail.

For the first iteration, `properties.parameter` will have a value of `"parameter1"`. The `params` parameter will have a value of `["parameter1"]`. This will cause the `create_configuration` fixture to search for configuration files in **tests** and **tests/parameter1** directories. The `properties.expected` value that is compared with `config.var` is `"hello parameter1"`.

For the second iteration, `properties.parameter` has a value of `"parameter2"`. The `params` parameter has a value of `["parameter2"]` and the configuration files from directories **tests** and **test/parameter2** will be used. The `properties.expected` value that is compared with `config.var` is `"hello parameter2"`.

Let's see what pytest collects as tests:

```
$ pytest --collect-only
=========== test session starts ===========
```

(continues on next page)

(continued from previous page)

```
..
collected 2 items
<Module tests/test_parameterized.py>
  <Function test_parameterized[properties0]>
  <Function test_parameterized[properties1]>

========= no tests ran in 0.02s ===========
```

In the output, the number of iterations and the parameters of each are shown.

We can now execute the tests:

```
$ pytest
======== test session starts =========
..
collected 2 items

tests/test_parameterized.py ..  [100%]

========= 2 passed in 0.06s ==========
```

If we only wanted to execution one particular iteration:

```
$ pytest -k test_parameterized[properties0]
=============== test session starts ================
..
collected 2 items / 1 deselected / 1 selected

tests/test_parameterized.py .                  [100%]

========= 1 passed, 1 deselected in 0.010 ==========
```

2.2.2 Categorized

In this next section, we're going to increase the complexity with additional tests in another module. Create the following structure and files:

```
categorized/
└── tests/
    ├── category
    |   ├── something
    |   |   ├── parameter1
    |   |   |   └── conf.py
    |   |   └── parameter2
    |   |       └── conf.py
    |   ├── something_else
    |   |   ├── parameter1
    |   |   |   └── conf.py
    |   |   └── parameter2
    |   |       └── conf.py
    |   ├── test_something_else.py
```

(continues on next page)

(continued from previous page)

```
    |   └── test_something.py
    └── conf.py
```

conf.py

```
var = "{hello_world}"
```

something/parameter1/conf.py

```
hello_world = "hello parameter 1"
```

something/parameter2/conf.py

```
hello_world = "hello parameter 2"
```

something_else/parameter1/conf.py

```
hello_world = "hello parameter 3"
```

something_else/parameter2/conf.py

```
hello_world = "hello parameter 4"
```

test_something.py

```
import pytest
from collections import namedtuple

Properties = namedtuple("Properties", ["parameter", "expected"])

def get_test_properties():
    p1 = Properties("parameter1", "hello world 1")
    p2 = Properties("parameter2", "hello world 2")
    return [p1, p2]

@pytest.mark.parametrize("properties", get_test_properties())
def test_something(properties, create_configuration):
    conf = create_configuration(params=[properties.parameter])
    assert conf.var == properties.expected
```

test_something_else.py

```
import pytest
from collections import namedtuple

Properties = namedtuple("Properties", ["parameter", "expected"])

def get_test_properties():
    p1 = Properties("parameter1", "hello world 3")
    p2 = Properties("parameter2", "hello world 4")
    return [p1, p2]

@pytest.mark.parametrize("properties", get_test_properties())
```

(continues on next page)

(continued from previous page)

```
def test_something_else(properties, create_configuration):
    conf = create_configuration(params=[properties.parameter])
    assert conf.var == properties.expected
```

Notice that in the `create_configuration` call of both modules, the `test_name` parameter is not specified. When not specified, the value internally is taken from the node name. The `"test_"` prefix is removed along with the characters after `[`.

For example, if we execute:

```
$ pytest -k test_something[properties0]
```

The variable `test_name` will be `"something"`. If we execute:

```
$ pytest -k test_something_else[properties0]
```

The variable `test_name` will be `"something_else"`.

In both cases, the `test_name` directory will be an additional directory that is searched for configuration files.

The order of search directories is **top level**, **category**, **test name**, and **parameter**.

For the test case of **test_something[properties0]**, the order of directories searched is: **tests** (top level), **category** (directory of test modules), **something** (based on `test_name`), and **parameter1** (based on `params`).

Let's now execute the tests:

```
$ pytest
========== test session starts ==========
..
collected 4 items

category/test_something.py ..      [ 50%]
category/test_something_else.py .. [100%]

============ 4 passed in 0.21s ============
```

2.3 Pytest Fixtures and Options

While the ROAST test framework is test-tool agnostic, pytest fixtures have been provided to simplify test creation and execution. The fixtures provided are grouped into three categories.

2.3.1 Configuration Generation

The features in this section provide functionality related to generating a test configuration.

create_configuration fixture

This fixture will generate a configuration based on the location of the test file being executed. The optional parameters to this fixture are:

- `test_name` - This specifies a test configuration to be retrieved in a subdirectory relative to the test file. If not provided, this will resolve to `request.node.name`.
- `base_params` - The list provided is written to the base_params attribute of the configuration.
- `params` - This allows a list of test configurations to be retrieved in subdirectories relative to the test file. This is typically used to retrieved definitions when parameterizing tests.
- `overrides` - This allows variable overrides to be specified. This can be another Python file, key/value pair, or both. If not provided, it will attempt to use `pytest.override` from the *--override option*.
- `machine` - This specifies an override file based on the machine type where the test will be executed. If not provided, it will attempt to use `pytest.machine` from the *--machine option*.

See also:

`roast.confParser.generate_conf()`

--override option

This option allows the user to specify an override file, key/value pairs, or both to override variable values in the configuration generated from configuration files.

Examples:

To override a variable named "my_version":

```
$ pytest --override my_version=2020.1
```

To override a list variable named "my_list":

```
$ pytest --override my_list=c,d
```

To override multiple variables defined in a Python file:

```
$ pytest --override /path/to/file.py
```

To override using both file and key/value pair:

```
$ pytest --override /path/to/file.py my_version=2020.1
```

--machine option

This option allows the user to specific an override file from a specific location. The functionality is the same as using an override by file except that the file location does not need to be specified.

To override using a machine file:

```
$ pytest --machine zynq
```

--randomize option

This option allows the user to set a global *randomize* configuration parameter to *True* or *False*. If not specified, this is set to *False* by default.

To enable randomization:

```
$ pytest --randomize
```

2.3.2 Board Acquisition

The fixtures in this section provide wrappers for easy board acquisition.

board_session fixtures

These fixtures will return an instantiated `Board` object based on the `board_type` keyword argument. For example, the `host_board_session` fixture will instantiate a `Board` object with `board_type="host_target"` and return a `TargetBoard` object.

This is accomplished through loading of the `TargetBoard` object as an entry point upon installation of the `roast-xilinx` package. Custom `Board` classes can be written and registered as a plugin where additional fixtures can be created to call the custom classes.

board fixture

This fixture wraps *board_session fixtures* and attempts to retrieve the `board_interface` key from the configuration to be used as the `board_type`. A valid board type must be specified otherwise an exception will be generated.

Upon return, the `Board` object will have attributes such as:

- config
- target_console - console session with a board

Additionally, the `start()` method will be called to initialized the board.

2.3.3 Scenario Generation

The fixture is an all-in-one wrapper for configuration generation and automated loading of test suites and system components of a test system.

create_scenario fixture

This fixture wraps the *create_configuration fixture* and calls `roast.component.scenario()` to return a `Scenario` object which contains handles to all loaded plugins (instantiated classes).

2.4 Data Provider

The ROAST data provider (randomization engine) is a data generator based on the Mimesis open source library.

The generated data can be used to define a range of values or a set of discrete values for a parameter of a component. Examples include:

- Clock divider where the value can be from 3 to 7
- Bus data width where the value can be 32, 64, or 128 bits

The functions supplied by this engine are bound to the Python random module. While the examples shown are numerical based, the base functions can use objects for elements within sequences. For example, the order of objects within a sequence can be randomized when calling the shuffle method.

2.4.1 Randomizer class

The `Randomizer` class is the base provider. The functionality includes:

- *Custom data providers*

Initialize random generator with seed

An optional seed can be provided for reproducable randomization. If seed is omitted or `None`, the current system time is used.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer(seed=12345)
```

Disable randomization

In some scenarios, randomization may need to be disabled. Randomization is enabled by default and can be disabled at the global or parameter level. To disable, set `randomize` to `False`. When disabled, the `default` value will be returned.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer(randomize=False)
```

To disable at the parameter level, set the `randomize` property to `False` when loading parameter values into the data provider. Both `default` and `randomize` properties are discussed in the section, *Load provider with JSON data file*.

Random module

The random module can be accessed directly through the class for methods such as the distribution functions.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.random.random()
0.27405095889396036
>>> randomizer.random.uniform(10, 100)
23.291595105628538
```

Load provider with JSON data file

A JSON data file can be loaded to establish a database of parameters and possible values.

Listing 1: parameters.json

```
{
    "ip": {
        "attribute1": {
            "default": 64,
            "elements": [32, 64, 128, 256, 512],
            "excluded": [64, 128]
        },
        "attribute2": {
            "default": 127,
            "range": [0, 255],
```

(continues on next page)

(continued from previous page)

```
            "excluded": [35, 36, 37]
        },
        "attribute3": {
            "default": 1.4,
            "range": [0, 2, 0.2]
        },
        "attribute4": {
            "default": 18,
            "range": [10, 20],
            "randomize": false
        },
        "attribute5": {
            "range": [20, 30],
            "replace": false
        },
        "attribute6": {
            "default": "0X08",
            "elements": ["0x02", "0o10", "0b10000"],
            "format": "#010b"
        },
        "attribute7": {
            "default": "0x0C",
            "range": ["0X00", "0O20", "0B10"],
            "format": "#04x"
        },
        "attribute8": {
            "default": "1.000000e+03",
            "range": ["8.000000e+02", "1.200000e+03", 50],
            "format": "e"
        },
        "attribute9": {
            "range": [1, 100]
        },
        "attribute10": {
            "range": [1, 100],
            "preset": "LOW_HEAVY"
        },
        "attribute11": {
            "range": [1, 100],
            "shape": [1, 10]
        }
    }
}
```

For each attribute, there are nine properties that can be defined:

1. **elements** - Sequence of discrete values.
2. **range** - Sequence of range parameters: start, stop, and step.
    - If one value is provided, it is the stop value. It assumes that start is 0 and step is 1.
    - If two values are provided, it is the start and stop values. It assumes step is 1.
3. **excluded** - Sequence of values to never return.

4. **default** - Value returned if the global `randomize` or parameter `randomize` is set to `False`.
5. **format** - Defines how string values are presented. See Format Specification Mini-Language for details. To specify hex, octal, or binary formatted strings, use the alternative format starting with `#`.
6. **randomize** - Boolean to disable randomization for the specific parameter.
7. **replace** - Boolean to determine whether sampling is with or without replacement. This setting will override the global `replace` setting.
8. **preset** - Weighted randomization preset to use to shape the generated data. `LOW_HEAVY`, `HIGH_HEAVY`, `NORMAL`, `INVERSE_NORMAL`, and `EXTERME_LIMITS`.
9. **shape** - Sequence of shape parameter values for the randomization distribution function. This is for the Alpha and Beta values of the Beta distribution.

Note:

- See *Exclude values used in previous runs* for details on usage of `replace`.
- See *Single or sequence of random values (weighted distribution)* for details on usage of `preset` and `shape`.

Warning:

- Either `elements` or `range` must exist or an exception will be raised. Both should not be used in the same attribute. If both exist, `elements` will have priority.
- Both `preset` and `shape` should not be used for the same attribute. If both exist, `preset` will have priority.

When a JSON file is specified, it is read and stored into `parameters`.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> randomizer.parameters
<Box: {'ip': {'attribute1': {'default': 64, 'elements': [32, 64, 128, 256, 512],
↪'excluded': [64, 128]}, 'attribute2': {'default': 127, 'range': [0, 255], 'excluded':␣
↪[35, 36, 37]}, 'attribute3': {'default': 9, 'range': [0, 127]}, 'attribute4': {'default
↪': 127}, 'attribute5': {'default': 2, 'range': [0, 10, 2], 'excluded': [4, 6]},
↪'attribute6': {'default': 14, 'range': [20]}, 'attribute7': {'default': 1.4, 'range':␣
↪[0, 2, 0.2]}, 'attribute8': {'default': -1.4, 'range': [0, -2, -0.2]}, 'attribute9': {
↪'default': -1.6, 'range': [-2, -1, 0.2]}, 'attribute10': {'default': 18, 'range': [10,␣
↪20], 'randomize': False}, 'attribute11': {'range': [20, 30], 'replacement': False},
↪'attribute12': {'default': 8, 'elements': [2, 8, 16], 'format': '#010b'}, 'attribute13
↪': {'default': 12, 'range': [0, 16, 2], 'format': '#04x'}, 'attribute14': {'default':␣
↪1000.0, 'range': [800.0, 1200.0, 50], 'format': 'e'}, 'delay_500': {'default': 52,
↪'range': [20, 100]}, 'delay_501': {'default': 52, 'range': [20, 100]}, 'delay_502': {
↪'default': 52, 'range': [20, 100]}, 'delay_503': {'default': 52, 'range': [20, 100]},
↪'ramp_500': {'default': 2, 'range': [1, 30]}, 'ramp_501': {'default': 2, 'range': [1,␣
↪30]}, 'ramp_502': {'default': 2, 'range': [1, 30]}, 'ramp_503': {'default': 2, 'range
↪': [1, 30]}}}>
```

To retrieve the default value of `ip.attribute1`:

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer(randomize=False)
>>> randomizer.datafile = "parameters.json"
>>> randomizer.get_value("ip.attribute1")
64
>>> randomizer.get_value("ip.attribute6")
'0b00001000'
>>> randomizer.get_value("ip.attribute7")
'0x0c'
>>> randomizer.get_value("ip.attribute8")
'1.000000e+03'
```

Note:

- Internally, string values are converted to `float`. Hex, octal, and binary strings are converted to `int`.
- Changed in version 4.0: `values` replaced with `elements` due to library conflict.
- Added in versions 4.0: `format`, `randomize`, `replace`, `preset`, and `shape` properties.

Load provider through configuration

The randomization parameters can be loaded through configuration. The same properties defined in the previous section, *Load provider with JSON data file*, are used. To load through configuration, the parameters will need to be defined within a Box so that dot-based keys can be used to traverse the dictionary. After the configuration is generated, store it into `parameters`. There are two methods to define through configuration.

Method 1 (nested dictionary):

Listing 2: conf.py

```
from box import Box

parameters = Box(
    {
        "ip": {
            "attribute1": {
                "default": 64,
                "elements": [32, 64, 128, 256, 512],
                "excluded": [64, 128]
            },
            "attribute2": {
                "default": 127,
                "range": [0, 255],
                "excluded": [35, 36, 37]
            },
            "attribute3": {
                "default": 1.4,
                "range": [0, 2, 0.2]
            },
            "attribute4": {
                "default": 18,
```

(continues on next page)

(continued from previous page)

```
                "range": [10, 20],
                "randomize": false
            },
            "attribute5": {
                "range": [20, 30],
                "replace": false
            }
        }
    },
    box_dots=True,
)

del Box
```

Method 2 (dot-based):

Listing 3: conf.py

```
from box import Box

parameters = Box(default_box=True, box_intact_types=[list, tuple])
parameters.ip.attribute1.default = 64
parameters.ip.attribute1.elements = [32, 64, 128, 256, 512]
parameters.ip.attribute1.excluded = [64, 128]
parameters.ip.attribute2.default = 127
parameters.ip.attribute2.range = [0, 255]
parameters.ip.attribute2.excluded = [35, 36, 37]
parameters.ip.attribute3.default = 1.4
parameters.ip.attribute3.range = [0, 2, 0.2]
parameters.ip.attribute4.default = 18
parameters.ip.attribute4.range = [10, 20]
parameters.ip.attribute4.randomize = False
parameters.ip.attribute5.range = [20, 30]
parameters.ip.attribute5.replace = False

del Box
```

With either method, the configuration can be generated through the *Layered Configuration System*. If using `pytest`, use the fixture for *Configuration Generation*.

```
>>> from roast.providers.randomizer import Randomizer
>>> from roast.confParser import generate_conf
>>> randomizer = Randomizer()
>>> config = generate_conf()
>>> randomizer.parameters = config.parameters
>>> randomizer.get_value("ip.attribute1")
256
```

Note: New in version 4.0.

Boolean choice

This will randomly return `True` or `False`.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.boolean()
False
>>> randomizer.boolean()
True
```

All possible values

The does not have any randomization and is a helper function to return all possible values with excluded values removed.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> randomizer.get_all_values("ip.attribute1")
[32, 256, 512]
>>> randomizer.get_all_values("ip.attribute8")
['8.000000e+02', '8.500000e+02', '9.000000e+02', '9.500000e+02', '1.000000e+03', '1.
↪050000e+03', '1.100000e+03', '1.150000e+03', '1.200000e+03']
```

Single random value

This will return a random element.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> randomizer.get_value("ip.attribute1")
256
>>> randomizer.get_value("ip.attribute7")
'0x10'
```

Sequence of random values (weighted)

A sequence of choices can be randomly generated from a sequence of values. There are three options:

1. Length - To define how many elements are in returned sequence.
2. Weights - To define which elements should be selected more often. The weights can be relative.
3. Unique - To define if any selected elements can be repeated.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> attribute3 = randomizer.get_all_values("ip.attribute3")
>>> attribute3
[0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0]
```

(continues on next page)

(continued from previous page)

```
>>> randomizer.choices(attribute3, length=4)
[0.4, 1.0, 1.0, 1.2]
>>> randomizer.choices(attribute3, length=4, unique=True)
[0.6, 0, 0.8, 1.8]
>>> weights = [10, 1, 1, 1, 10, 10, 1, 1, 1, 1, 10]
>>> randomizer.choices(attribute3, weights, length=4)
[0.4, 0, 1.0, 1.0]
>>> randomizer.choices(attribute3, weights, length=4, unique=True)
[2.0, 1.0, 0.8, 0]
```

Sequence of random values (essential)

A sequence of choices randomly generated from a sequence of values defined with essential elements. This can also be considered as a constrained shuffle. This is similar to the previous type, *Sequence of random values (weighted)*, without weights and results are always unique. There are two options:

1. Essential - To define which elements must be included in returned sequence.
2. Length - To define how many elements are in returned sequence.

Behaviors:

- By default, if both `essential` and `length` are not specified, a normal shuffle will be returned.
- If only `essential` is specified, a random length between the length of `essential` sequence and the length of `items` will be returned.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.shuffle(items=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[8, 4, 2, 10, 7, 6, 9, 3, 1, 5]
>>> randomizer.shuffle(items=(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), length=7)
(1, 5, 10, 8, 2, 4, 6)
>>> randomizer.shuffle(items=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], essential=[3, 6, 9])
[2, 9, 5, 7, 3, 6, 8]
>>> randomizer.shuffle(items=(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), essential=[3, 6, 9],␣
↪length=5)
(5, 6, 7, 3, 9)
```

Note: New in version 4.0.

Single or sequence of random values (weighted distribution)

A single or sequence of choices can be randomly generated from a sequence of values based on the Beta probability distribution. Presets are available to provide predetermined probability of an element being selected from the provided sequence of elements.

- LOW_HEAVY - elements near lower end more frequently
- HIGH_HEAVY - elements near higher end more frequently
- NORMAL - elements near median more frequently
- INVERSE_NORMAL - elements near ends more frequently

- EXTREME_LIMITS - elements near ends significantly more frequently

Listing 4: parameters.json

```
{
    "ip": {
        "attribute9": {
            "range": [1, 100]
        },
        "attribute10": {
            "range": [1, 100],
            "preset": "LOW_HEAVY"
        },
        "attribute11": {
            "range": [1, 100],
            "shape": [1, 10]
        }
    }
}
```

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> values = []
>>> for _ in range(10):
...     value = randomizer.get_value("ip.attribute10") # LOW_HEAVY
...     values.append(value)
...
>>> values
[1, 13, 7, 17, 38, 21, 21, 16, 7, 7]
>>>
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> values = []
>>> for _ in range(10):
...     value = randomizer.get_value("ip.attribute11") # a=1, b=10
...     values.append(value)
...
>>> values
[1, 2, 3, 9, 3, 8, 3, 2, 13, 22]
```

The preset can be directly specified when calling `get_value()`. This will override any setting specified in the configuration.

```
>>> from roast.providers.randomizer import Randomizer, WeightPreset
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> values = []
>>> for _ in range(10):
...     value = randomizer.get_value("ip.attribute9", preset=WeightPreset.LOW_HEAVY)
...     values.append(value)
...
>>> values
```

(continues on next page)

(continued from previous page)

```
[5, 3, 9, 35, 12, 5, 5, 9, 18, 15]
>>>
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> values = []
>>> for _ in range(10):
...     value = randomizer.get_value("ip.attribute9", preset=WeightPreset.HIGH_HEAVY)
...     values.append(value)
...
>>> values
[94, 95, 90, 99, 97, 93, 90, 80, 99, 97]
```

The Alpha and Beta shape parameters can also be provided as parameters. As an example, we can provide the values of the LOW_HEAVY preset. This will also override any setting in the configuration.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> values = []
>>> for _ in range(10):
...     value = randomizer.get_value("ip.attribute9", a=1, b=10)
...     values.append(value)
...
>>> values
[6, 25, 8, 16, 6, 1, 12, 5, 8, 10]
```

Warning: When using the weighted distribution feature, the distribution parameters cannot be changed on-the-fly. This means that if values have been generated for an attribute using `LOW_HEAVY`, it cannot be changed to generate `HIGH_HEAVY` values by simply specifying the new preset. This is because the weights for each possible value are generated during the initial call to `get_value()` and stored in the `weights` property within the randomizer for performance reasons. If changing of the distribution parameters is needed, instantiate another instance of the randomizer or empty the weight array.

Choices defined by expression

This will return randomized values defined by an expression.

For example, randomized delays and ramp times with a condition to define the relationship.

The condition is `ip.delay_502 >= ip.delay_503 + ip.ramp_503`

Listing 5: parameters.json

```
{
    "ip": {
        "delay_502": {
            "default": 52,
            "range": [20, 100]
        },
        "delay_503": {
            "default": 52,
```

(continues on next page)

(continued from previous page)

```
            "range": [20, 100]
        },
        "ramp_503": {
            "default": 2,
            "range": [1, 30]
        }
    }
}
```

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> randomizer.generate_conditional("ip.delay_502 >= ip.delay_503 + ip.ramp_503")
{'ip.delay_502': 67, 'ip.delay_503': 43, 'ip.ramp_503': 12}
```

In addition to defined attributes, variables can also be used to be evaluated.

For example, a dynamically defined offset as part of the expression.

The condition is `ip.delay_502 >= ip.delay_503 + offset`.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> offset = 10
>>> randomizer.generate_conditional("ip.delay_502 >= ip.delay_503 + offset",␣
↪offset=offset)
{'ip.delay_502': 80, 'ip.delay_503': 40}
>>> offset = 20
>>> randomizer.generate_conditional("ip.delay_502 >= ip.delay_503 + offset",␣
↪offset=offset)
{'ip.delay_502': 100, 'ip.delay_503': 65}
```

The complete set of operators that can be used are listed in the Arithmetic Parser User Guide.

Sequence of choices defined by expression

This will return a sequence of randomized values based on a condition.

For example, a randomized sequence of four delay values where the each random value needs to be greater than the previous where the condition is: `delay_500 < delay_501 < delay_502 < delay_503`.

Listing 6: parameters.json

```
{
    "ip": {
        "delay_500": {
            "default": 52,
            "range": [20, 100]
        },
        "delay_501": {
            "default": 52,
            "range": [20, 100]
```

(continues on next page)

(continued from previous page)

```
        },
        "delay_502": {
            "default": 52,
            "range": [20, 100]
        },
        "delay_503": {
            "default": 52,
            "range": [20, 100]
        }
    }
}
```

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> randomizer.generate_sequence("prev < current", ["ip.delay_500", "ip.delay_501", "ip.
↪delay_502", "ip.delay_503"])
{'ip.delay_500': 41, 'ip.delay_501': 78, 'ip.delay_502': 81, 'ip.delay_503': 86}
```

While both `prev` and `current` are pre-defined and can be used to define the condition, any expression can be used.

Within the expression, variables can also be used to be evaluated.

For example, a dynamically defined offset as part of the expression where the condition is `prev + offset <= current`.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> offset = 10
>>> randomizer.generate_sequence("prev + offset <= current", ["ip.delay_500", "ip.delay_
↪501", "ip.delay_502"], offset=offset)
{'ip.delay_500': 26, 'ip.delay_501': 36, 'ip.delay_502': 72}
```

The complete set of operators that can be used are listed in the Arithmetic Parser User Guide.

Exporting generated values

Whenever a random value is generated by the provider, the value is stored into the `data` class attribute, a Box dictionary. This can very useful for debugging purposes and can be exported to a JSON file. This JSON file can then be used as a database of previously generated values discussed in the section *Exclude values used in previous runs*.

Since the dictionary may have dotted keys, a special method is provided to convert any dotted keys into a nested dictionary that can be properly exported to JSON.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> randomizer.get_value("ip.attribute1")
32
>>> randomizer.data
<Box: {'ip.attribute1': [32]}>
>>> randomizer.to_json("generated.json")
```

(continues on next page)

(continued from previous page)

```
>>> with open("generated.json", "r") as f:
...     parsed = json.load(f)
...
>>> parsed
{'ip': {'attribute1': [32]}}
```

Note: New in version 4.0.

Exclude values used in previous runs

Values generated by the provider can be excluded for use in subsequent runs. This allows tests to sample without replacement over multiple runs. To enable this, import the exported data file as described in the previous section. Upon import, the global `replace` property will be set to `False`. To override and disable for a parameter, set the `replace` property to `True` at the parameter level as described in *Load provider with JSON data file*.

By default, samples are with replacement.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer(seed=12345)
>>> randomizer.datafile = "parameters.json"
>>> randomizer.get_all_values("ip.attribute1")
[32, 256, 512]
>>> randomizer.get_value("ip.attribute1")
512
>>> randomizer.get_all_values("ip.attribute1")
[32, 256, 512]
```

This can be set to without replacement at the parameter level. Example shown is for JSON parameter file.

Listing 7: parameters.json

```
{
    "ip": {
        "attribute5": {
            "range": [20, 30],
            "replace": false
        }
    }
}
```

The value generated from the first call will be excluded from possible values on the next call.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer(seed=12345)
>>> randomizer.datafile = "parameters.json"
>>> randomizer.get_all_values("ip.attribute5")
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
>>> randomizer.get_value("ip.attribute5")
26
>>> randomizer.get_all_values("ip.attribute5")
[20, 21, 22, 23, 24, 25, 27, 28, 29, 30]
```

To exclude values from an exported data file, load as a excludes file which will set the global `replace` property to `False`. The contents of the excludes file will then be the initial value of the `data` class attribute. After randomized values are generated, optionally export all of the accumulated values back to the generated values JSON file.

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> randomizer.get_value("ip.attribute1")
32
>>> randomizer.data
<Box: {'ip.attribute1': [32]}>
>>> randomizer.to_json("generated.json")
>>>
>>> randomizer = Randomizer()
>>> randomizer.datafile = "parameters.json"
>>> randomizer.excludes_file = "generated.json"
>>> randomizer.data
<Box: {'ip': {'attribute1': [32]}}>
>>> randomizer.get_all_values("ip.attribute1")
[256, 512]
>>> randomizer.get_value("ip.attribute1")
512
>>> randomizer.data
<Box: {'ip': {'attribute1': [32, 512]}}>
>>> randomizer.get_all_values("ip.attribute1")
[256]
>>> randomizer.to_json()
>>> with open("generated.json", "r") as f:
...     parsed = json.load(f)
...
>>> parsed
{'ip': {'attribute1': [32, 512]}}
```

Warning: The parameter setting will override the global setting. For example, if the global setting is `False` and the parameter setting is `True`, the attribute will sample with replacement, meaning that it will not exclude previously generated values loaded from file.

Note: New in version 4.0.

Custom data providers

Custom providers can be created and dynamically added to generate specific data.

```
from mimesis import BaseProvider

class SomeProvider(BaseProvider):
    class Meta:
        name = "some_provider"
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def hello():
        return "Hello!"
```

This can be used as such:

```
>>> from roast.providers.randomizer import Randomizer
>>> randomizer = Randomizer()
>>> randomizer.add_provider(SomeProvider)
>>> randomizer.some_provider.hello()
'Hello!'
```

Documentation for Custom Providers at Mimesis website.

CHAPTER

THREE

ROAST EXAMPLES

- **Complete examples**: *Examples Repository*

3.1 Examples Repository

In this repository, full working examples are available to showcase various features of ROAST. These range from simple "hello word" applications to full Linux images.

Varying styles of repository structures and test parameterization are provided to illustrate how test variants can be defined within a test suite.

To download these examples, visit https://github.com/Xilinx/roast-examples.

3.1.1 Prerequisites

User needs to install Python3 version >=3.6 along with pip3 and virtual environments python package.

- Python3.6 +
- pip3
- virtualenv
- picocom (Terminal emulator for serial port access and communication)
- xvfb

The PetaLinux tools need to be installed as a non-root user.

This repo has been tested with below version along with dependent Xilinx tools/packages/libraries on Ubuntu 18.04 machine as mentioned below:

- roast 2.1.0
- pytest-roast 1.2.0.post1
- roast-xilinx 2.1.0
- Petalinux 2020.2
- Vitis 2020.2

Addtional documentation:

- Xilinx PetaLinux installation user guide
- Xilinx Vitis installation user guide

- Common issues user guide

In ROAST, picocom application has been used for connecting and disconnecting local board support. https://github.com/npat-efault/picocom

Tests need to be executed with a bash prompt shown in the format:

```
bash-4.2$
```

Fetch the repository roast-examples tests repo:

```
# Cloning roast-examples regression to be tested
$ git clone https://github.com/Xilinx/roast-examples.git

# Go to roast-examples directory
$ cd roast-examples
```

3.1.2 Hello World

Executing build test cases from hello_world directory:

```
$ pytest hello_world/test_hello_world.py -k "build" -vv
```

Executing run test cases from hello_world directory:

```
$ pytest hello_world/test_hello_world.py -k "run" -vv
```

3.1.3 PetaLinux

PetaLinux prerequisites

Download page link where you find packages/BSPs for petalinux test cases: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/2020-2.html

Zynq UltraScale+ MPSoC Board Support Packages - <version> —> Download:- ZCU106 BSP (BSP - 1.74 GB)

Zynq-7000 SoC Board Support Packages - <version> —> Download:- ZC706 BSP (BSP - 110.74 MB)

Note: Once plnx build test case is run successfully, you can use rootfs.cpio file from "build/zynqmp/<zcu106_bsp/zc706_bsp>/images/" and rename osl_demo's required file. Also alternately, you can use xilinx open source base rootfs file.

Incase if you have issue running multiple test cases, use usb_relay to auto power on and off board.

Replace <version> with your requirement. Also note that size of each file may be different depends on version.

Executing plnx_demo tests

Executing build test cases from plnx_demo:

```
$ pytest plnx_demo/test_zynqmq_bsp.py -k "build" -vv
```

Executing run tests cases from plnx_demo:

```
$ pytest plnx_demo/test_zynqmq_bsp.py -k "run" -vv
```

3.1.4 OSL

OSL prerequisites

User has to copy respective base-rootfs files based on platform from tar/zip folder to osl_demo_basic/component/src path directory structure as mentioned below:

```
component
└── src
    ├── mkimage
    ├── zynq
    │   └── petalinux-image-minimal-zynq-generic.rootfs.cpio
    └── zynqmp
        └── petalinux-image-minimal-zynqmp-generic.rootfs.cpio
```

Examples:

- osl_demo_basic/component/src/zynqmp/petalinux-image-minimal-zynqmp-generic.rootfs.cpio
- osl_demo_basic/component/src/zynq/petalinux-image-minimal-zynq-generic.rootfs.cpio

Executing osl_demo tests

Executing build test cases from osl_demo_basic:

```
$ pytest osl_demo_basic/test_build_osl_basic.py -k zcu106 -vv
```

Executing run test cases from osl_demo_basic:

```
$ pytest osl_demo_basic/test_run_osl_basic.py -k zcu106 -vv
```

3.1.5 Advanced OSL

Advanced OSL prerequisites

User has to create two folder namely zynqmp and zynq under component/rootfs/src path directory structure as mentioned below: And copy respective base-rootfs files based on platform from tar/zip folder.

```
component
├── rootfs
│   ├── conf.py
```

(continues on next page)

(continued from previous page)

```
    └── src
        ├── mkimage
        ├── zynq
        │   └── petalinux-image-minimal-zynq-generic.rootfs.cpio
        └── zynqmp
            └── petalinux-image-minimal-zynqmp-generic.rootfs.cpio
```

Examples:

- osl_demo/component/rootfs/src/zynqmp/petalinux-image-minimal-zynqmp-generic.rootfs.cpio
- osl_demo/component/rootfs/src/zynq/petalinux-image-minimal-zynq-generic.rootfs.cpio

Executing advanced osl_demo tests

Executing build test cases from osl_demo:

```
$ pytest osl_demo/test_build_osl.py --machine=zcu106 -vv
```

Executing run test cases from osl_demo:

```
$ pytest osl_demo/test_run_osl.py --machine="zcu106" -vv
```

CHAPTER

FOUR

ADVANCED FEATURES OF ROAST

- **Developer Interface**: *API Reference*
- **Building plugins**: *Component Plugin Models* | *Component Plugins*

4.1 API Reference

4.1.1 AIE

4.1.2 Basebuild

4.1.3 Bif

4.1.4 Board

4.1.5 Boot

4.1.6 CMake

4.1.7 ConfParser

4.1.8 Cross Compile

4.1.9 Linux

4.1.10 Logger

4.1.11 PetaLinux

4.1.12 Plugin

4.1.13 Randomizer

4.1.14 Scenario

Scenario class

A `Scenario` class in **component/_init_.py** is provided within ROAST which will create and save references to component plugins. This class has high level methods to dispatch method calls to the loaded components. These include:

- `load_component` - calls the **__init__.py** component class constructor
- `configure_component` - calls the `configure()` component method
- `build_component` - calls the `build()` component method and returns result
- `deploy_component` (TestSuite only) - calls the `deploy()` component method and returns result
- `run_component` (TestSuite only) - calls the `run()` component method and returns result

Accessor methods are also available to access the direct component object themselves. These include:

- `ts()` - returns the object of the TestSuite component
- `sys()` - returns the object of the System component specified by component name (because there can be more than one system component)

Scenario function

A `scenario` function in **component/_init_.py** is provided to instantiate `Scenario`, automatically load the components, and return the instance.

4.1.15 SD

4.1.16 Serial

4.1.17 SystemBase

4.1.18 TestSuiteBase

4.1.19 Utilities

4.1.20 Xexpect

4.1.21 XSCT

4.1.22 Xsdb

4.1.23 Yocto

4.2 Component Plugin Models

While there are several different methods to load code dynamically in Python, the best approach for ROAST is to built on top of setuptools entry points. At a high level, the primary reasons are that ROAST is a Python package and we want to avoid creating custom mechanisms. Additionally, ROAST plugins such as the component libraries can also be packaged to extend existing ROAST namespaces.

A plugin framework named stevedore includes the functionality needed by ROAST. Currently, stevedore is maintained by the Redhat Openstack project, an extremely popular cloud computing platform.

When constructing systems, we need to establish a uniform methodology across all use cases so that end users can focus on test development rather than system construction. This is realized through the creation of plugin models that are defined to ensure an uniform API across all components.

Currently, there are two plugin models implemented - *TestSuite* and *System*. The abstract interfaces are defined as `TestSuiteBase` (roast/component/testsuite.py) and `SystemBase` (roast/component/system.py).

4.2.1 TestSuite

```
from abc import ABCMeta, abstractmethod


class TestSuiteBase(metaclass=ABCMeta):
    """Base class for Test Suite component plugin
    """

    def __init__(self, config):
        self.config = config

    @abstractmethod
    def configure(self):
        """Abstract class method to configure the TestSuite component
        """

    @abstractmethod
    def build(self):
        """Abstract class method to build the TestSuite component
        """

    @abstractmethod
    def deploy(self):
        """Abstract class method to deploy the TestSuite component
        """

    @abstractmethod
    def run(self):
        """Abstract class method to run the TestSuite component
        """
```

This plugin model is designed for a component that will need to deploy other components and execute an application to verify correctness. Typically, there is only one TestSuite component for a particular test type. For example, running a validation tool to verify correctness on a constructed system.

In this plugin model, there are four methods: `configure()`, `build()`, `deploy()`, and `run()`. Each plugin that inherits from the base class will need to implement these methods.

4.2.2 System

```
from abc import ABCMeta, abstractmethod


class SystemBase(metaclass=ABCMeta):
    """Base class for System component plugins
    """

    def __init__(self, config):
        self.config = config

    @abstractmethod
    def configure(self):
        """Abstract class method to configure the System component
        """

    @abstractmethod
    def build(self):
        """Abstract class method to build the System component
        """
```

This plugin model is designed for components that make up a system. There can be one or many system components for a particular test type. For example, an operating system and programmable logic as part of a constructed system.

In this plugin model, there are only two methods: `configure()` and `build()`. Each plugin that inherits from the base class will need to implement these methods.

See also:

Component Plugins

4.3 Component Plugins

In this tutorial, we will discuss how to create and load component plugins. If you're not familiar with the component models, please first visit *Component Plugin Models* for details.

4.3.1 Creating Plugins

Steps to creating a component plugin:

1. With the two possible plugin models, `TestSuiteBase` or `SystemBase`, choose the model that fits for your component plugin.
2. Subclass the base class and implement the required methods.
3. Define the name of the plugin and extend one of two possible roast component namespaces.

Let's start with a system component named `MySystem` that has two methods, `configure()` and `build()`. This will be created in **my_system.py**.

Create a test suite component named `MyTestSuite` that has four methods, `configure()`, `build()`, `deploy()`, and `run()`. This will be create in **my_testsuite.py**.

We will need a configuration file named **conf.py** to define which components will be used in the test scenario and a test module named **test_scenario.py**. Add these into the **tests** directory:

```
repository/
└── tests/
    ├── conf.py
    ├── my_system.py
    ├── my_testsuite.py
    └── test_scenario.py
```

conf.py

```
roast = {"system": ["my_system"], "testsuite": "my_testsuite"}
var = "hello world"
```

Here, we define a system component named "my_system" and a testsuite component named "my_testsuite".

Note: These names are identifiers used by each plugin when registering as an entry point and do not need to match the module filename.

Also note that the value for system is a list since a test scenario could have more than one.

---

my_system.py

```
from roast.component.system import SystemBase

class MySystem(SystemBase):
    def __init__(self, config):
        super().__init__(config)

    def configure(self):
        print("MySystem configure called")

    def build(self):
        msg = "MySystem build called"
        print(msg)
        return msg

    def custom_method(self, data):
        msg = f"MySystem custom method called with {data}"
        print(msg)
        return msg
```

Here, we are subclassing from the SystemBase abstract base class and implementing the required methods configure() and build(). In addition, we are going to extend the class with a method named custom_method().

The super() call in __init__() is where the configuration is stored as an attribute of the class and can accessed through self.config.

my_testsuite.py

```
from roast.component.testsuite import TestSuiteBase

class MyTestSuite(TestSuiteBase):
    def __init__(self, config):
        super().__init__(config)
```

(continues on next page)

(continued from previous page)

```
    def configure(self):
        print("MyTestSuite configure called")

    def build(self):
        msg = "MyTestSuite build called"
        print(msg)
        return msg

    def deploy(self):
        print("MyTestSuite deploy called")

    def run(self):
        msg = "MyTestSuite run called"
        print(msg)
        return msg

    def custom_method(self, data):
        msg = f"MyTestSuite custom method called with {data}"
        print(msg)
        return msg
```

Similar to `MySystem`, subclass and implement the required methods. Also extend the class with a custom method.

4.3.2 Loading Plugins

In order to dynamically load component plugins, they first need to be registered in the ROAST namespace as an object that can be called through entry points. Two namespaces are available: `roast.component.testsuite` for a TestSuite component and `roast.component.system` for System components.

test_scenario.py

```
import inspect
from roast.utils import register_plugin
import my_system, my_testsuite

def test_my_scenario(create_scenario):
    system_name = "my_system"
    system_location = inspect.getsourcefile(my_system)
    register_plugin(
        system_location, system_name, "system", "my_system:MySystem",
    )
    testsuite_name = "my_testsuite"
    testsuite_location = inspect.getsourcefile(my_testsuite)
    register_plugin(
        testsuite_location, testsuite_name, "testsuite", "my_testsuite:MyTestSuite",
    )

    scn = create_scenario()
    my_ts = scn.ts
    my_sys = scn.sys(system_name)

    scn.configure_component()
```

(continues on next page)

(continued from previous page)

```
    assert my_ts.config.var == "hello world"
    assert my_sys.config.var == "hello world"

    build_results = scn.build_component()
    assert build_results[testsuite_name] == "MyTestSuite build called"
    assert build_results[system_name] == "MySystem build called"

    scn.deploy_component()

    run_results = scn.run_component()
    assert run_results[testsuite_name] == "MyTestSuite run called"

    custom_result = my_ts.custom_method(data="hello")
    assert custom_result == "MyTestSuite custom method called with hello"
    custom_result = my_sys.custom_method(data="hello")
    assert custom_result == "MySystem custom method called with hello"
```

Here, we need to first register the `MySystem` and `MyTestSuite` classes. In order to register, we will need their file locations which can be hard coded or obtained through the use of `inspect.getfile()`.

If the component objects will be packaged into a Python package, this can be defined in **setup.py**.

```
entry_points={
    "roast.component.system": ["repository.tests.my_system = my_system:MySystem",],
    "roast.component.testsuite": ["repository.test.my_testsuite = my_
↪testsuite:MyTestSuite",],
}
```

Next, we call the `create_scenario()` fixture to load the components and also generate a configuration. The variable `scn` holds references to both `MySystem` and `MyTestSuite` instances. To access the specific instance, use `scn.ts` for test suite or `scn.sys` for systems. For systems, the specific name is required since there can be more than one system component. Here, we're going to assign the instances to `my_ts` and `my_sys`.

Calling the `configure_component()` method will in turn call the `configure()` method in every loaded instance. In both `MySystem` and `MyTestSuite`, the configuration is stored as a `config` attribute. With `my_ts.config.var`, we can access the value of `var` in the `MyTestSuite` instance and similarly with `my_sys.config.var` for `MySystem`. Both should return `"hello world"`.

Similarly, when `build_component()` is called, this will call `build()` in each instance. The difference here is that values are returned in a dictionary that can be accessed using the name as the key.

The methods `deploy_component()` and `run_component()` are essentially the same as the previous two except that these call **only** the `MyTestSuite` instance since systems do not have `deploy()` or `run()` methods.

Lastly, since we access to the instances, we can call custom methods, pass parameters, and also return values.

Let's now execute the tests:

```
$ pytest -rP
=========== test session starts ===========
..
collected 1 item

tests/test_scenario .                [100%]
```

(continues on next page)

(continued from previous page)

```
==========================================
---------- Captured stdout call -----------
MySystem configure called
MyTestSuite configure called
MySystem build called
MyTestSuite build called
MyTestSuite deploy called
MyTestSuite run called
MyTestSuite custom method called with hello
MySystem custom method called with hello
============ 1 passed in 0.12s ============
```