ReFrame Documentation

Release 3.0 (rev: ec92346e)

CSCS

Jun 04, 2020

TABLE OF CONTENTS

ReFrame is a high-level framework for writing regression tests for HPC systems. The goal of the framework is to abstract away the complexity of the interactions with the system, separating the logic of a regression test from the low-level details, which pertain to the system configuration and setup. This allows users to write easily portable regression tests, focusing only on the functionality.

Regression tests in ReFrame are simple Python classes that specify the basic parameters of the test. The framework will load the test and will send it down a well-defined pipeline that will take care of its execution. The stages of this pipeline take care of all the system interaction details, such as programming environment switching, compilation, job submission, job status query, sanity checking and performance assessment.

ReFrame also offers a high-level and flexible abstraction for writing sanity and performance checks for your regression tests, without having to care about the details of parsing output files, searching for patterns and testing against reference values for different systems.

Writing system regression tests in a high-level modern programming language, like Python, poses a great advantage in organizing and maintaining the tests. Users can create their own test hierarchies or test factories for generating multiple tests at the same time and they can also customize them in a simple and expressive way.

Finally, ReFrame offers a powerful and efficient runtime for running and managing the execution of tests, as well as integration with common logging facilities, where ReFrame can send live data from currently running performance tests.

CHAPTER

ONE

USE CASES

A pre-release of ReFrame has been in production at the Swiss National Supercomputing Centre since early December 2016. The first public release was in May 2017 and it is being actively developed since then. Several HPC centers around the globe have adopted ReFrame for testing and benchmarking their systems in an easy, consistent and reproducible way. You can read a couple of use cases here.

CHAPTER

TWO

PUBLICATIONS

- Slides [pdf] @ 5th EasyBuild User Meeting 2020.
- Slides [pdf] @ HPC System Testing BoF, SC'19.
- Slides [pdf] @ HUST 2019, SC'19.
- Slides [pdf] @ HPC Knowledge Meeting '19.
- Slides [pdf] & Talk @ FOSDEM'19.
- Slides [pdf] @ 4th EasyBuild User Meeting.
- Slides [pdf] @ HUST 2018, SC'18.
- Slides [pdf] @ CSCS User Lab Day 2018.
- Slides [pdf] @ HPC Advisory Council 2018.
- Slides [pdf] @ SC17.
- Slides [pdf] @ CUG 2017.

2.1 Getting Started

2.1.1 Requirements

- Python 3.6 or higher. Python 2 is not supported.
- Required Python packages can be found in the `requirements.txt` file, which you can install as follows:

```
pip3 install -r requirements.txt
```

Optional Requirements

If you want to run the framework's unit tests, you will also need a C compiler that is able to compile a "Hello, World!" program and recognize the `-O3` option.

Note: Changed in version 2.8: A functional TCL modules system is no more required. ReFrame can now operate without a modules system at all.

Note: Changed in version 3.0: Support for Python 3.5 has been dropped.

2.1.2 Getting the Framework

ReFrame's latest stable version is available through different channels:

- As a PyPI package:

```
pip install reframe-hpc
```

Note: The above method performs a bare installation of ReFrame not including unittests and tutorial examples.

- As a Spack package:

```
spack install reframe
```

- As an EasyBuild package:

```
eb easybuild/easyconfigs/r/ReFrame/ReFrame-VERSION.eb -r
```

Getting the Latest and Greatest

If you want the latest development version or any pre-release, you can clone ReFrame from Github:

```
git clone https://github.com/eth-cscs/reframe.git
```

Pre-release versions are denoted with the `devX` suffix and are tagged in the repository.

2.1.3 Running the Unit Tests

You can optionally run the framework's unit tests to make sure that everything is set up correctly:

```
./test_reframe.py -v
```

The output should look like the following:

```
========================================= test session starts␣
↪==========================================
platform darwin -- Python 3.7.3, pytest-4.3.0, py-1.8.0, pluggy-0.9.0 -- /usr/local/
↪opt/python/bin/python3.7
cachedir: .pytest_cache
rootdir: /Users/karakasv/Repositories/reframe, inifile:
collected 697 items

unittests/test_argparser.py::test_arguments PASSED                              ␣
↪          [  0%]
unittests/test_argparser.py::test_parsing PASSED                                ␣
↪          [  0%]
unittests/test_argparser.py::test_option_precedence PASSED                      ␣
↪          [  0%]
unittests/test_argparser.py::test_option_with_config PASSED                     ␣
↪          [  0%]
unittests/test_argparser.py::test_option_envvar_conversion_error PASSED         ␣
↪          [  0%]
unittests/test_buildsystems.py::TestMake::test_emit_from_buildsystem PASSED     ␣
↪          [  0%]
```

(continues on next page)

(continued from previous page)

```
unittests/test_buildsystems.py::TestMake::test_emit_from_env PASSED
↪          [  1%]
unittests/test_buildsystems.py::TestMake::test_emit_no_env_defaults PASSED
↪          [  1%]
unittests/test_buildsystems.py::TestCMake::test_emit_from_buildsystem PASSED
↪          [  1%]
unittests/test_buildsystems.py::TestCMake::test_emit_from_env PASSED
↪          [  1%]
unittests/test_buildsystems.py::TestCMake::test_emit_no_env_defaults PASSED
↪          [  1%]
unittests/test_buildsystems.py::TestAutotools::test_emit_from_buildsystem PASSED
↪          [  1%]
unittests/test_buildsystems.py::TestAutotools::test_emit_from_env PASSED
↪          [  1%]
unittests/test_buildsystems.py::TestAutotools::test_emit_no_env_defaults PASSED
↪          [  2%]
unittests/test_buildsystems.py::TestSingleSource::test_emit_from_env PASSED
↪          [  2%]
unittests/test_buildsystems.py::TestSingleSource::test_emit_no_env PASSED
↪          [  2%]
unittests/test_check_filters.py::TestCheckFilters::test_have_cpu_only PASSED
↪          [  2%]
unittests/test_check_filters.py::TestCheckFilters::test_have_gpu_only PASSED
↪          [  2%]
unittests/test_check_filters.py::TestCheckFilters::test_have_name PASSED
↪          [  2%]
unittests/test_check_filters.py::TestCheckFilters::test_have_not_name PASSED
↪          [  2%]
unittests/test_check_filters.py::TestCheckFilters::test_have_prgenv PASSED
↪          [  3%]
unittests/test_check_filters.py::TestCheckFilters::test_have_tags PASSED
↪          [  3%]
unittests/test_check_filters.py::TestCheckFilters::test_invalid_regex PASSED
↪          [  3%]
unittests/test_check_filters.py::TestCheckFilters::test_partition PASSED
↪          [  3%]
unittests/test_cli.py::test_check_success PASSED
↪          [  3%]
unittests/test_cli.py::test_check_submit_success SKIPPED
↪          [  3%]
unittests/test_cli.py::test_check_failure PASSED
↪          [  3%]
<... output omitted ...>
unittests/test_utility.py::TestPpretty::test_simple_types PASSED
↪          [ 95%]
unittests/test_utility.py::TestPpretty::test_mixed_types PASSED
↪          [ 95%]
unittests/test_utility.py::TestPpretty::test_obj_print PASSED
↪          [ 95%]
unittests/test_utility.py::TestChangeDirCtxManager::test_change_dir_working PASSED
↪          [ 95%]
unittests/test_utility.py::TestChangeDirCtxManager::test_exception_propagation PASSED
↪          [ 95%]
unittests/test_utility.py::TestMiscUtilities::test_allx PASSED
↪          [ 95%]
unittests/test_utility.py::TestMiscUtilities::test_decamelize PASSED
↪          [ 96%]
```

(continues on next page)

(continued from previous page)

```
unittests/test_utility.py::TestMiscUtilities::test_sanitize PASSED
↪          [ 96%]
unittests/test_utility.py::TestScopedDict::test_construction PASSED
↪          [ 96%]
unittests/test_utility.py::TestScopedDict::test_contains PASSED
↪          [ 96%]
unittests/test_utility.py::TestScopedDict::test_delitem PASSED
↪          [ 96%]
unittests/test_utility.py::TestScopedDict::test_iter_items PASSED
↪          [ 96%]
unittests/test_utility.py::TestScopedDict::test_iter_keys PASSED
↪          [ 96%]
unittests/test_utility.py::TestScopedDict::test_iter_values PASSED
↪          [ 97%]
unittests/test_utility.py::TestScopedDict::test_key_resolution PASSED
↪          [ 97%]
unittests/test_utility.py::TestScopedDict::test_scope_key_name_pseudoconflict PASSED
↪          [ 97%]
unittests/test_utility.py::TestScopedDict::test_setitem PASSED
↪          [ 97%]
unittests/test_utility.py::TestScopedDict::test_update PASSED
↪          [ 97%]
unittests/test_utility.py::TestReadOnlyViews::test_mapping PASSED
↪          [ 97%]
unittests/test_utility.py::TestReadOnlyViews::test_sequence PASSED
↪          [ 97%]
unittests/test_utility.py::TestOrderedSet::test_concat_files PASSED
↪          [ 98%]
unittests/test_utility.py::TestOrderedSet::test_construction PASSED
↪          [ 98%]
unittests/test_utility.py::TestOrderedSet::test_construction_empty PASSED
↪          [ 98%]
unittests/test_utility.py::TestOrderedSet::test_construction_error PASSED
↪          [ 98%]
unittests/test_utility.py::TestOrderedSet::test_difference PASSED
↪          [ 98%]
unittests/test_utility.py::TestOrderedSet::test_intersection PASSED
↪          [ 98%]
unittests/test_utility.py::TestOrderedSet::test_operators PASSED
↪          [ 98%]
unittests/test_utility.py::TestOrderedSet::test_reversed PASSED
↪          [ 99%]
unittests/test_utility.py::TestOrderedSet::test_str PASSED
↪          [ 99%]
unittests/test_utility.py::TestOrderedSet::test_union PASSED
↪          [ 99%]
unittests/test_utility.py::TestOrderedSet::test_unique_abs_paths PASSED
↪          [ 99%]
unittests/test_versioning.py::TestVersioning::test_comparing_versions PASSED
↪          [ 99%]
unittests/test_versioning.py::TestVersioning::test_version_format PASSED
↪          [ 99%]
unittests/test_versioning.py::TestVersioning::test_version_validation PASSED
↪          [100%]

============================== 620 passed, 77 skipped in 64.58 seconds
↪==============================
```

You will notice that several tests will be skipped. ReFrame uses a generic configuration by default, so that it can run on any system. As a result, all tests for scheduler backends, environment modules, container platforms etc. will be skipped. As soon as you configure ReFrame specifically for your system, you may rerun the test suite using your system configuration file by passing the `--rfm-user-config=CONFIG_FILE`.

2.1.4 Where to Go from Here

The *Configuring ReFrame for Your Site* page guides you through the basic configuration aspects of ReFrame. The *ReFrame Tutorials* will allow you to get a first idea on how to write and run ReFrame tests. *Advanced Topics* explain different aspects of the framework whereas the *ReFrame Manuals* provide complete reference guides for the command line interface, the configuration parameters and the programming APIs for writing tests.

2.2 Configuring ReFrame for Your Site

ReFrame comes pre-configured with a minimal generic configuration that will allow you to run ReFrame on any system. This will allow you to run simple local tests using the default compiler of the system. Of course, ReFrame is much more powerful than that. This section will guide you through configuring ReFrame for your HPC cluster. We will use as a starting point a simplified configuration for the Piz Daint supercomputer at CSCS and we will elaborate along the way.

If you started using ReFrame from version 3.0, you can keep on reading this section, otherwise you are advised to have a look first at the *Migrating to ReFrame 3* page.

ReFrame's configuration file can be either a JSON file or Python file storing the site configuration in a JSON-formatted string. The latter format is useful in cases that you want to generate configuration parameters on-the-fly, since ReFrame will import that Python file and the load the resulting configuration. In the following we will use a Python-based configuration file also for historical reasons, since it was the only way to configure ReFrame in versions prior to 3.0.

2.2.1 Locating the Configuration File

ReFrame looks for a configuration file in the following locations in that order:

1. `${HOME}/.reframe/settings.{py,json}`
2. `${RFM_INSTALL_PREFIX}/settings.{py,json}`
3. `/etc/reframe.d/settings.{py,json}`

If both `settings.py` and `settings.json` are found, the Python file is preferred. The `RFM_INSTALL_PREFIX` variable refers to the installation directory of ReFrame or the top-level source directory if you are running ReFrame from source. Users have no control over this variable. It is always set by the framework upon startup.

If no configuration file is found in any of the predefined locations, ReFrame will fall back to a generic configuration that allows it to run on any system. You can find this generic configuration file here. Users may *not* modify this file.

There are two ways to provide a custom configuration file to ReFrame:

1. Pass it through the `-C` or `--config-file` option.
2. Specify it using the `RFM_CONFIG_FILE` environment variable.

Command line options take always precedence over their respective environment variables.

2.2.2 Anatomy of the Configuration File

The whole configuration of ReFrame is a single JSON object whose properties are responsible for configuring the basic aspects of the framework. We'll refer to these top-level properties as *sections*. These sections contain other objects which further define in detail the framework's behavior. If you are using a Python file to configure ReFrame, this big JSON configuration object is stored in a special variable called `site_configuration`.

We will explore the basic configuration of ReFrame through the following configuration file that permits ReFrame to run on Piz Daint. For the complete listing and description of all configuration options, you should refer to the *Configuration Reference*.

```
site_configuration = {
    'systems': [
        {
            'name': 'daint',
            'descr': 'Piz Daint',
            'hostnames': ['daint'],
            'modules_system': 'tmod',
            'partitions': [
                {
                    'name': 'login',
                    'descr': 'Login nodes',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
                    'max_jobs': 4,
                },
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes (Haswell/P100)',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'modules': ['daint-gpu'],
                    'access': ['--constraint=gpu'],
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
                    'container_platforms': [
                        {
                            'type': 'Singularity',
                            'modules': ['Singularity']
                        }
                    ],
                    'max_jobs': 100,
                },
                {
                    'name': 'mc',
                    'descr': 'Multicore nodes (Broadwell)',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
```

(continues on next page)

(continued from previous page)

```
                    'modules': ['daint-mc'],
                    'access': ['--constraint=mc'],
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
                    'container_platforms': [
                        {
                            'type': 'Singularity',
                            'modules': ['Singularity']
                        }
                    ],
                    'max_jobs': 100,
                }
            ]
        }
    ],
    'environments': [
        {
            'name': 'PrgEnv-cray',
            'modules': ['PrgEnv-cray']
        },
        {
            'name': 'PrgEnv-gnu',
            'modules': ['PrgEnv-gnu']
        },
        {
            'name': 'PrgEnv-intel',
            'modules': ['PrgEnv-intel']
        },
        {
            'name': 'PrgEnv-pgi',
            'modules': ['PrgEnv-pgi']
        }
    ],
    'logging': [
        {
            'level': 'debug',
            'handlers': [
                {
                    'type': 'file',
                    'name': 'reframe.log',
                    'level': 'debug',
                    'format': '[%(asctime)s] %(levelname)s: %(check_name)s:
↪%(message)s',   # noqa: E501
                    'append': False
                },
                {
                    'type': 'stream',
                    'name': 'stdout',
                    'level': 'info',
                    'format': '%(message)s'
                },
                {
                    'type': 'file',
```

(continues on next page)

(continued from previous page)

```
                    'name': 'reframe.out',
                    'level': 'info',
                    'format': '%(message)s',
                    'append': False
                }
            ],
            'handlers_perflog': [
                {
                    'type': 'filelog',
                    'prefix': '%(check_system)s/%(check_partition)s',
                    'level': 'info',
                    'format': '%(check_job_completion_time)s|reframe %(version)s|
↪%(check_info)s|jobid=%(check_jobid)s|%(check_perf_var)s=%(check_perf_value)s|ref=
↪%(check_perf_ref)s (l=%(check_perf_lower_thres)s, u=%(check_perf_upper_thres)s)',
↪# noqa: E501
                    'datefmt': '%FT%T%:z',
                    'append': True
                }
            ]
        }
    ],
    'general': [
        {
            'check_search_path': ['tutorial/'],
        }
    ]
}
```

There are three required sections that each configuration file must provide: `systems`, `environments` and `logging`. We will first cover these and then move on to the optional ones.

Systems Configuration

ReFrame allows you to configure multiple systems in the same configuration file. Each system is a different object inside the `systems` section. In our example we define only one system, namely Piz Daint:

```
    'systems': [
        {
            'name': 'daint',
            'descr': 'Piz Daint',
            'hostnames': ['daint'],
            'modules_system': 'tmod',
            'partitions': [
                {
                    'name': 'login',
                    'descr': 'Login nodes',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
                    'max_jobs': 4,
```

(continues on next page)

(continued from previous page)

```
                },
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes (Haswell/P100)',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'modules': ['daint-gpu'],
                    'access': ['--constraint=gpu'],
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
                    'container_platforms': [
                        {
                            'type': 'Singularity',
                            'modules': ['Singularity']
                        }
                    ],
                    'max_jobs': 100,
                },
                {
                    'name': 'mc',
                    'descr': 'Multicore nodes (Broadwell)',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'modules': ['daint-mc'],
                    'access': ['--constraint=mc'],
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
                    'container_platforms': [
                        {
                            'type': 'Singularity',
                            'modules': ['Singularity']
                        }
                    ],
                    'max_jobs': 100,
                }
            ]
        }
    ],
```

Each system is associated with a set of properties, which in this case are the following:

- `name`: The name of the system. This should be an alphanumeric string (dashes - are allowed) and it will be used to refer to this system in other contexts.
- `descr`: A detailed description of the system.
- `hostnames`: This is a list of hostname patterns following the Python Regular Expression Syntax, which will be used by ReFrame when it tries to automatically select a configuration entry for the current system.
- `modules_system`: The environment modules system that should be used for loading environment modules

on this system. In this case, the classic Tcl implementation of the environment modules.

- `partitions`: The list of partitions that are defined for this system. Each partition is defined as a separate object. We devote the rest of this section in system partitions, since they are an essential part of ReFrame's configuration.

A system partition in ReFrame is not bound to a real scheduler partition. It is a virtual partition or separation of the system. In the example shown here, we define three partitions that none of them corresponds to a scheduler partition. The `login` partition refers to the login nodes of the system, whereas the `gpu` and `mc` partitions refer to two different set of nodes in the same cluster that are effectively separated using Slurm constraints. Let's pick the `gpu` partition and look into it in more detail:

```
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes (Haswell/P100)',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'modules': ['daint-gpu'],
                    'access': ['--constraint=gpu'],
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
                    'container_platforms': [
                        {
                            'type': 'Singularity',
                            'modules': ['Singularity']
                        }
                    ],
                    'max_jobs': 100,
                },
```

The basic properties of a partition are the following:

- `name`: The name of the partition. This should be an alphanumeric string (dashes - are allowed) and it will be used to refer to this partition in other contexts.
- `descr`: A detailed description of the system partition.
- `scheduler`: The workload manager (job scheduler) used in this partition for launching parallel jobs. In this particular example, the Slurm scheduler is used. For a complete list of the supported job schedulers, see here.
- `launcher`: The parallel job launcher used in this partition. In this case, the `srun` command will be used. For a complete list of the supported parallel job launchers, see here.
- `access`: A list of scheduler options that will be passed to the generated job script for gaining access to that logical partition. Notice how in this case, the nodes are selected through a constraint and not an actual scheduler partition.
- `environs`: The list of environments that ReFrame will use to run regression tests on this partition. These are just symbolic names that refer to environments defined in the `environments` section described below.
- `container_platforms`: A set of supported container platforms in this partition. Each container platform is an object with a name and list of environment modules to load, in order to enable this platform. For a complete list of the supported container platforms, see here.
- `max_jobs`: The maximum number of concurrent regression tests that may be active (i.e., not completed) on this partition. This option is relevant only when ReFrame executes with the asynchronous execution policy.

Environments Configuration

We have seen already environments to be referred to by the `environs` property of a partition. An environment in ReFrame is simply a collection of environment modules, environment variables and compiler and compiler flags definitions. None of these attributes is required. An environment can simply by empty, in which case it refers to the actual environment that ReFrame runs in. In fact, this is what the generic fallback configuration of ReFrame does.

Environments in ReFrame are configured under the `environments` section of the documentation. In our configuration example for Piz Daint, we define each ReFrame environment to correspond to each of the Cray-provided programming environments. In other systems, you could define a ReFrame environment to wrap a toolchain (MPI + compiler combination):

```
    'environments': [
        {
            'name': 'PrgEnv-cray',
            'modules': ['PrgEnv-cray']
        },
        {
            'name': 'PrgEnv-gnu',
            'modules': ['PrgEnv-gnu']
        },
        {
            'name': 'PrgEnv-intel',
            'modules': ['PrgEnv-intel']
        },
        {
            'name': 'PrgEnv-pgi',
            'modules': ['PrgEnv-pgi']
        }
    ],
```

Each environment is associated with a name. This name will be used to reference this environment in different contexts, as for example in the `environs` property of the system partitions. This environment definition is minimal, since the default values for the rest of the properties serve our purpose.

An important feature in ReFrame's configuration, is that you can define section objects differently for different systems or system partitions. In the following, for demonstration purposes, we define `PrgEnv-gnu` differently for the `mc` partition of the `daint` system (notice the condensed form of writing this as `daint:mc`):

```
{
    'name': 'PrgEnv-gnu',
    'modules': ['PrgEnv-gnu', 'openmpi'],
    'cc':  'mpicc',
    'cxx': 'mpicxx',
    'ftn': 'mpif90',
    'target_systems': ['daint:mc']
}
```

This environment loads different modules and sets the compilers differently, but the most important part is the `target_systems` property. This property is a list of systems or system/partition combinations (as in this case) where this definition of the environment is in effect. This means that `PrgEnv-gnu` will defined this way only for regression tests running on `daint:mc`. For all the other systems, it will be defined as shown before.

Logging configuration

ReFrame has a powerful logging mechanism that gives fine grained control over what information is being logged, where it is being logged and how this information is formatted. Additionally, it allows for logging performance data from performance tests into different channels. Let's see how logging is defined in our example configuration, which also represents a typical one for logging:

```
    'logging': [
        {
            'level': 'debug',
            'handlers': [
                {
                    'type': 'file',
                    'name': 'reframe.log',
                    'level': 'debug',
                    'format': '[%(asctime)s] %(levelname)s: %(check_name)s:
↪%(message)s',    # noqa: E501
                    'append': False
                },
                {
                    'type': 'stream',
                    'name': 'stdout',
                    'level': 'info',
                    'format': '%(message)s'
                },
                {
                    'type': 'file',
                    'name': 'reframe.out',
                    'level': 'info',
                    'format': '%(message)s',
                    'append': False
                }
            ],
            'handlers_perflog': [
                {
                    'type': 'filelog',
                    'prefix': '%(check_system)s/%(check_partition)s',
                    'level': 'info',
                    'format': '%(check_job_completion_time)s|reframe %(version)s|
↪%(check_info)s|jobid=%(check_jobid)s|%(check_perf_var)s=%(check_perf_value)s|ref=
↪%(check_perf_ref)s (l=%(check_perf_lower_thres)s, u=%(check_perf_upper_thres)s)',
↪# noqa: E501
                    'datefmt': '%FT%T%:z',
                    'append': True
                }
            ]
        }
    ],
```

Logging is configured under the `logging` section of the configuration, which is a list of logger objects. Unless you want to configure logging differently for different systems, a single logger object is enough. Each logger object is associated with a logging level stored in the `level` property and has a set of logging handlers that are actually responsible for handling the actual logging records. ReFrame's output is performed through the logging mechanism, meaning that if you don't specify any logging handler, you will not get any output from ReFrame! The `handlers` property of the logger object holds the actual handlers. Notice that you can use multiple handlers at the same time, which enables you to feed ReFrame's output to different sinks and at different verbosity levels. All handler objects share a set of common properties. These are the following:

- `type`: This is the type of the handler, which determines its functionality. Depending on the handler type, handler-specific properties may be allowed or required.
- `level`: The cut-off level for messages reaching this handler. Any message with a lower level number will be filtered out.
- `format`: A format string for formatting the emitted log record. ReFrame uses the format specifiers from Python Logging, but also defines its owns specifiers.
- `datefmt`: A time format string for formatting timestamps. There are two log record fields that are considered timestamps: (a) `asctime` and (b) `check_job_completion_time`. ReFrame follows the time formatting syntax of Python's time.strftime() with a small tweak allowing full RFC3339 compliance when formatting time zone differences.

We will not go into the details of the individual handlers here. In this particular example we use three handlers of two distinct types:

1. A file handler to print debug messages in the `reframe.log` file using a more extensive message format that contains a timestamp, the level name etc.
2. A stream handler to print any informational messages (and warnings and errors) from ReFrame to the standard output. This handles essentially the actual output of ReFrame.
3. A file handler to print the framework's output in the `reframe.out` file.

It might initially seem confusing the fact that there are two `level` properties: one at the logger level and one at the handler level. Logging in ReFrame works hierarchically. When a message is logged, an log record is created, which contains metadata about the message being logged (log level, timestamp, ReFrame runtime information etc.). This log record first goes into ReFrame's internal logger, where the record's level is checked against the logger's level (here `debug`). If the log record's level exceeds the log level threshold from the logger, it is forwarded to the logger's handlers. Then each handler filters the log record differently and takes care of formatting the log record's message appropriately. You can view logger's log level as a general cut off. For example, if we have set it to `warning`, no debug or informational messages would ever be printed.

Finally, there is a special set of handlers for handling performance log messages. These are stored in the `handlers_perflog` property. The performance handler in this example will create a file per test and per system/partition combination and will append the performance data to it every time the test is run. Notice in the `format` property how the message to be logged is structured such that it can be easily parsed from post processing tools. Apart from file logging, ReFrame offers more advanced performance logging capabilities through Syslog and Graylog.

General configuration options

General configuration options of the framework go under the `general` section of the configuration file. This section is optional. In this case, we define the search path for ReFrame test files to be the `tutorial/` subdirectory and we also instruct ReFrame to recursively search for tests there. There are several more options that can go into this section, but the reader is referred to the *Configuration Reference* for the complete list.

Other configuration options

There are finally two more optional configuration sections that are not discussed here:

1. The `schedulers` section holds configuration variables specific to the different scheduler backends and
2. the `modes` section defines different execution modes for the framework. Execution modes are discussed in the *How ReFrame Executes Tests* page.

2.2.3 Picking a System Configuration

As discussed previously, ReFrame's configuration file can store the configurations for multiple systems. When launched, ReFrame will pick the first matching configuration and load it. This process is performed as follows: ReFrame first tries to obtain the hostname from `/etc/xthostname`, which provides the unqualified *machine name* in Cray systems. If this cannot be found, the hostname will be obtained from the standard `hostname` command. Having retrieved the hostname, ReFrame goes through all the systems in its configuration and tries to match the hostname against any of the patterns defined in each system's `hostnames` property. The detection process stops at the first match found, and that system's configuration is selected.

As soon as a system configuration is selected, all configuration objects that have a `target_systems` property are resolved against the selected system, and any configuration object that is not applicable is dropped. So, internally, ReFrame keeps an *instantiation* of the site configuration for the selected system only. To better understand this, let's assume that we have the following `environments` defined:

```
'environments': [
    {
        'name': 'PrgEnv-cray',
        'modules': ['PrgEnv-cray']
    },
    {
        'name': 'PrgEnv-gnu',
        'modules': ['PrgEnv-gnu']
    },
    {
        'name': 'PrgEnv-gnu',
        'modules': ['PrgEnv-gnu', 'openmpi'],
        'cc':  'mpicc',
        'cxx': 'mpicxx',
        'ftn': 'mpif90',
        'target_systems': ['foo']
    }
],
```

If the selected system is `foo`, then ReFrame will use the second definition of `PrgEnv-gnu` which is specific to the `foo` system.

You can override completely the system auto-selection process by specifying a system or system/partition combination with the `--system` option, e.g., `--system=daint` or `--system=daint:gpu`.

2.2.4 Querying Configuration Options

ReFrame offers the powerful `--show-config` command-line option that allows you to query any configuration parameter of the framework and see how it is set for the selected system. Using no arguments or passing `all` to this option, the whole configuration for the currently selected system will be printed in JSON format, which you can then pipe to a JSON command line editor, such as jq, and either get a colored output or even generate a completely new ReFrame configuration!

Passing specific configuration keys in this option, you can query specific parts of the configuration. Let's see some concrete examples:

- Query the current system's partitions:

```
./bin/reframe -C tutorial/config/settings.py --system=daint --show-config=systems/
↪0/partitions
```

```
[
  {
    "name": "login",
    "descr": "Login nodes",
    "scheduler": "local",
    "launcher": "local",
    "environs": [
      "PrgEnv-cray",
      "PrgEnv-gnu",
      "PrgEnv-intel",
      "PrgEnv-pgi"
    ],
    "max_jobs": 4
  },
  {
    "name": "gpu",
    "descr": "Hybrid nodes (Haswell/P100)",
    "scheduler": "slurm",
    "launcher": "srun",
    "modules": [
      "daint-gpu"
    ],
    "access": [
      "--constraint=gpu"
    ],
    "environs": [
      "PrgEnv-cray",
      "PrgEnv-gnu",
      "PrgEnv-intel",
      "PrgEnv-pgi"
    ],
    "container_platforms": [
      {
        "name": "Singularity",
        "modules": [
          "Singularity"
        ]
      }
    ],
    "max_jobs": 100
  },
  {
    "name": "mc",
    "descr": "Multicore nodes (Broadwell)",
    "scheduler": "slurm",
    "launcher": "srun",
    "modules": [
      "daint-mc"
    ],
    "access": [
      "--constraint=mc"
    ],
    "environs": [
      "PrgEnv-cray",
      "PrgEnv-gnu",
      "PrgEnv-intel",
      "PrgEnv-pgi"
```

(continues on next page)

(continued from previous page)

```
    ],
    "container_platforms": [
      {
        "name": "Singularity",
        "modules": [
          "Singularity"
        ]
      }
    ],
    "max_jobs": 100
  }
]
```

Check how the output changes if we explicitly set system to `daint:login`:

```
./bin/reframe -C tutorial/config/settings.py --system=daint:login --show-
↪config=systems/0/partitions
```

```
[
  {
    "name": "login",
    "descr": "Login nodes",
    "scheduler": "local",
    "launcher": "local",
    "environs": [
      "PrgEnv-cray",
      "PrgEnv-gnu",
      "PrgEnv-intel",
      "PrgEnv-pgi"
    ],
    "max_jobs": 4
  }
]
```

ReFrame will internally represent system `daint` as having a single partition only. Notice also how you can use indexes to objects elements inside a list.

- Query an environment configuration:

```
./bin/reframe -C tutorial/config/settings.py --system=daint --show-
↪config=environments/@PrgEnv-gnu
```

```
{
  "name": "PrgEnv-gnu",
  "modules": [
    "PrgEnv-gnu"
  ]
}
```

If an object has a `name` property you can address it by name using the `@name` syntax, instead of its index.

- Query an environment's compiler:

```
./bin/reframe -C tutorial/config/settings.py --system=daint --show-
↪config=environments/@PrgEnv-gnu/cxx
```

```
"CC"
```

Notice that although the C++ compiler is not defined in the environment's definitions, ReFrame will print the default value, if you explicitly query its value.

2.3 ReFrame Tutorials

2.3.1 Tutorial 1: The Basics

This tutorial will guide you through writing your first regression tests with ReFrame. We will start with the most common and simple case of a regression test that compiles a code, runs it and checks its output. We will then expand this example gradually by adding functionality and more advanced sanity and performance checks. By the end of the tutorial, you should be able to start writing your first regression tests with ReFrame.

All the tutorial examples can be found under the `tutorial/` directory in ReFrame's source code. Regardless how you have installed ReFrame, you can get the tutorial examples by cloning the project on Github.

This tutorial is tailored to the Piz Daint supercomputer, but you can use the tutorial tests on your system with slight adaptations. The configuration file for the tutorial can be found in `tutorial/config/settings.py` and we list it also here for completeness:

```
site_configuration = {
    'systems': [
        {
            'name': 'daint',
            'descr': 'Piz Daint',
            'hostnames': ['daint'],
            'modules_system': 'tmod',
            'partitions': [
                {
                    'name': 'login',
                    'descr': 'Login nodes',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
                    'max_jobs': 4,
                },
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes (Haswell/P100)',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'modules': ['daint-gpu'],
                    'access': ['--constraint=gpu'],
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
```

(continues on next page)

(continued from previous page)

```
                    'container_platforms': [
                        {
                            'type': 'Singularity',
                            'modules': ['Singularity']
                        }
                    ],
                    'max_jobs': 100,
                },
                {
                    'name': 'mc',
                    'descr': 'Multicore nodes (Broadwell)',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'modules': ['daint-mc'],
                    'access': ['--constraint=mc'],
                    'environs': [
                        'PrgEnv-cray',
                        'PrgEnv-gnu',
                        'PrgEnv-intel',
                        'PrgEnv-pgi'
                    ],
                    'container_platforms': [
                        {
                            'type': 'Singularity',
                            'modules': ['Singularity']
                        }
                    ],
                    'max_jobs': 100,
                }
            ]
        }
    ],
    'environments': [
        {
            'name': 'PrgEnv-cray',
            'modules': ['PrgEnv-cray']
        },
        {
            'name': 'PrgEnv-gnu',
            'modules': ['PrgEnv-gnu']
        },
        {
            'name': 'PrgEnv-intel',
            'modules': ['PrgEnv-intel']
        },
        {
            'name': 'PrgEnv-pgi',
            'modules': ['PrgEnv-pgi']
        }
    ],
    'logging': [
        {
            'level': 'debug',
            'handlers': [
                {
                    'type': 'file',
                    'name': 'reframe.log',
```

(continues on next page)

(continued from previous page)

```
                    'level': 'debug',
                    'format': '[%(asctime)s] %(levelname)s: %(check_name)s:
↪%(message)s',    # noqa: E501
                    'append': False
                },
                {
                    'type': 'stream',
                    'name': 'stdout',
                    'level': 'info',
                    'format': '%(message)s'
                },
                {
                    'type': 'file',
                    'name': 'reframe.out',
                    'level': 'info',
                    'format': '%(message)s',
                    'append': False
                }
            ],
            'handlers_perflog': [
                {
                    'type': 'filelog',
                    'prefix': '%(check_system)s/%(check_partition)s',
                    'level': 'info',
                    'format': '%(check_job_completion_time)s|reframe %(version)s|
↪%(check_info)s|jobid=%(check_jobid)s|%(check_perf_var)s=%(check_perf_value)s|ref=
↪%(check_perf_ref)s (l=%(check_perf_lower_thres)s, u=%(check_perf_upper_thres)s)',
↪# noqa: E501
                    'datefmt': '%FT%T%:z',
                    'append': True
                }
            ]
        }
    ],
    'general': [
        {
            'check_search_path': ['tutorial/'],
        }
    ]
}
```

The First Regression Test

The following is a simple regression test that compiles and runs a serial C program, which computes a matrix-vector product (`tutorial/src/example_matrix_vector_multiplication.c`), and verifies its sane execution. As a sanity check, it simply looks for a specific output in the output of the program. Here is the full code for this test:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn
```

(continues on next page)

(continued from previous page)

```
@rfm.simple_test
class Example1Test(rfm.RegressionTest):
    def __init__(self):
        self.descr = 'Simple matrix-vector multiplication example'
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.sourcepath = 'example_matrix_vector_multiplication.c'
        self.executable_opts = ['1024', '100']
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}
```

A regression test written in ReFrame is essentially a Python class that must eventually derive from *RegressionTest*. To make a test visible to the framework, you must decorate your final test class with one of the following decorators:

- *@simple_test*: for registering a single parameterless instantiation of your test.
- *@parameterized_test*: for registering multiple instantiations of your test.

Let's see in more detail how the `Example1Test` is defined:

```
@rfm.simple_test
class Example1Test(rfm.RegressionTest):
    def __init__(self):
```

The `__init__()` method is the constructor of your test. It is usually the only method you need to implement for your tests, especially if you don't want to customize any of the regression test pipeline stages. When your test is instantiated, the framework assigns a default name to it. This name is essentially a concatenation of the fully qualified name of the class and string representations of the constructor arguments, with any non-alphanumeric characters converted to underscores. In this example, the auto-generated test name is simply `Example1Test`. You may change the name of the test later in the constructor by setting the *name* attribute.

Note: Changed in version 2.19: Calling `super().__init__()` inside the constructor of a test is no more needed.

Warning: New in version 2.12: ReFrame requires that the names of all the tests it loads are unique. In case of name clashes, it will refuse to load the conflicting test.

The next line sets a more detailed description of the test:

```
self.descr = 'Simple matrix-vector multiplication example'
```

This is optional and it defaults to the auto-generated name of the test, if not specified.

Note: If you explicitly set only the name of the test, the description will not be automatically updated and will still keep its default value.

The next two lines specify the systems and the programming environments that this test is valid for:

```
self.valid_systems = ['*']
self.valid_prog_environs = ['*']
```

Both of these variables accept a list of system names or environment names, respectively. The `*` symbol is a wildcard meaning any system or any programming environment. The system and environment names listed in these variables must correspond to names of systems and environments defined in the ReFrame's *configuration file*.

Note: If a name specified in these lists does not appear in the settings file, it will be simply ignored.

When specifying system names you can always specify a partition name as well by appending `:<partname>` to the system's name. For example, given the configuration for our tutorial, `daint:gpu` would refer specifically to the `gpu` virtual partition of the system `daint`. If only a system name (without a partition) is specified in the *`self.valid_systems`* variable, e.g., `daint`, it means that this test is valid for any partition of this system.

The next line specifies the source file that needs to be compiled:

```
self.sourcepath = 'example_matrix_vector_multiplication.c'
```

ReFrame expects any source files, or generally resources, of the test to be inside an `src/` directory, which is at the same level as the regression test file. If you inspect the directory structure of the `tutorial/` folder, you will notice that:

```
tutorial/
    example1.py
    src/
        example_matrix_vector_multiplication.c
```

Notice also that you need not specify the programming language of the file you are asking ReFrame to compile or the compiler to use. ReFrame will automatically pick the correct compiler based on the extension of the source file. The exact compiler that is going to be used depends on the programming environment that the test is running with. For example, given our configuration, if it is run with `PrgEnv-cray`, the Cray C compiler will be used, if it is run with `PrgEnv-gnu`, the GCC compiler will be used etc. A user can associate compilers with programming environments in the ReFrame's *configuration file*.

The next line in our first regression test specifies a list of options to be used for running the generated executable (the matrix dimension and the number of iterations in this particular example):

```
self.executable_opts = ['1024', '100']
```

Notice that you do not need to specify the executable name. Since ReFrame compiled it and generated it, it knows the name. We will see in "*Tutorial 2: Customizing Further a Regression Test*" how you can specify the name of the executable, in cases that ReFrame cannot guess its name.

The next lines specify what should be checked for assessing the sanity of the result of the test:

```
self.sanity_patterns = sn.assert_found(
    r'time for single matrix vector multiplication', self.stdout)
```

This expression simply asks ReFrame to look for `time for single matrix vector multiplication` in the standard output of the test. The *`sanity_patterns`* attribute can only be assigned the result of a special type of functions, called *sanity functions*. Sanity functions are special in the sense that they are evaluated lazily. You can generally treat them as normal Python functions inside a *`sanity_patterns`* expression. ReFrame provides already a wide range of useful sanity functions ranging from wrappers to the standard built-in functions of Python to functions related to parsing the output of a regression test. For a complete listing of the available functions, you may have a look at the *Sanity Functions Reference*.

In our example, the *`assert_found`* function accepts a regular expression pattern to be searched in a file and either returns `True` on success or raises a `SanityError` in case of failure with a descriptive message. This function accepts any valid Python Regular Expression. As a file argument, *`assert_found`* accepts any filename, which will

be resolved against the stage directory of the test. You can also use the *stdout* and *stderr* attributes to reference the standard output and standard error, respectively.

Tip: You don't need to care about handling exceptions, and error handling in general, inside your test. The framework will automatically abort the execution of the test, report the error and continue with the next test case.

The last two lines of the regression test are optional, but serve a good role in a production environment:

```
self.maintainers = ['you-can-type-your-email-here']
self.tags = {'tutorial'}
```

In the *maintainers* attribute you may store a list of persons responsible for the maintenance of this test. In case of failure, this list will be printed in the failure summary.

The *tags* attribute is a set of tags that you can assign to this test. This is useful for categorizing the tests and helps in quickly selecting the tests of interest.

Note: The values assigned to the attributes of a *RegressionTest* are validated and if they don't have the correct type, an error will be issued by ReFrame. For a list of all the attributes and their types, please refer to the *ReFrame Programming APIs* guide.

Running the Tutorial Examples

ReFrame offers a rich command-line interface that allows you to control several aspects of its executions. Here we will only show you how to run a specific tutorial test:

```
./bin/reframe -C tutorial/config/settings.py -c tutorial/example1.py -r
```

If everything is configured correctly for your system, you should get an output similar to the following:

```
[ReFrame Setup]
  version:           3.0-dev7 (rev: 85ca676f)
  command:           './bin/reframe -C tutorial/config/settings.py -c tutorial/
↪example1.py -r'
  launched by:       user@daint104
  working directory: '/path/to/reframe'
  settings file:     'tutorial/config/settings.py'
  check search path: (R) '/path/to/reframe/tutorial/example1.py'
  stage directory:   '/path/to/reframe/stage'
  output directory:  '/path/to/reframe/output'

[==========] Running 1 check(s)
[==========] Started on Wed Jun  3 08:50:46 2020

[----------] started processing Example1Test (Simple matrix-vector multiplication␣
↪example)
[ RUN      ] Example1Test on daint:login using PrgEnv-cray
[ RUN      ] Example1Test on daint:login using PrgEnv-gnu
[ RUN      ] Example1Test on daint:login using PrgEnv-intel
[ RUN      ] Example1Test on daint:login using PrgEnv-pgi
[ RUN      ] Example1Test on daint:gpu using PrgEnv-cray
[ RUN      ] Example1Test on daint:gpu using PrgEnv-gnu
[ RUN      ] Example1Test on daint:gpu using PrgEnv-intel
```

(continues on next page)

(continued from previous page)

```
[ RUN      ] Example1Test on daint:gpu using PrgEnv-pgi
[ RUN      ] Example1Test on daint:mc using PrgEnv-cray
[ RUN      ] Example1Test on daint:mc using PrgEnv-gnu
[ RUN      ] Example1Test on daint:mc using PrgEnv-intel
[ RUN      ] Example1Test on daint:mc using PrgEnv-pgi
[----------] finished processing Example1Test (Simple matrix-vector multiplication␣
↪example)

[----------] waiting for spawned checks to finish
[       OK ] ( 1/12) Example1Test on daint:login using PrgEnv-intel [compile: 1.940s␣
↪run: 20.747s total: 23.778s]
[       OK ] ( 2/12) Example1Test on daint:login using PrgEnv-cray [compile: 0.347s␣
↪run: 25.096s total: 26.591s]
[       OK ] ( 3/12) Example1Test on daint:mc using PrgEnv-intel [compile: 2.019s␣
↪run: 6.286s total: 8.357s]
[       OK ] ( 4/12) Example1Test on daint:mc using PrgEnv-cray [compile: 0.506s run:␣
↪11.056s total: 11.744s]
[       OK ] ( 5/12) Example1Test on daint:gpu using PrgEnv-cray [compile: 0.435s␣
↪run: 19.499s total: 20.483s]
[       OK ] ( 6/12) Example1Test on daint:login using PrgEnv-gnu [compile: 1.648s␣
↪run: 25.631s total: 27.964s]
[       OK ] ( 7/12) Example1Test on daint:mc using PrgEnv-gnu [compile: 1.825s run:␣
↪10.434s total: 12.301s]
[       OK ] ( 8/12) Example1Test on daint:gpu using PrgEnv-intel [compile: 2.018s␣
↪run: 16.316s total: 18.529s]
[       OK ] ( 9/12) Example1Test on daint:login using PrgEnv-pgi [compile: 1.643s␣
↪run: 22.118s total: 24.090s]
[       OK ] (10/12) Example1Test on daint:gpu using PrgEnv-gnu [compile: 1.729s run:␣
↪20.028s total: 21.901s]
[       OK ] (11/12) Example1Test on daint:gpu using PrgEnv-pgi [compile: 1.753s run:␣
↪15.128s total: 16.923s]
[       OK ] (12/12) Example1Test on daint:mc using PrgEnv-pgi [compile: 1.732s run:␣
↪35.556s total: 37.330s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 12 test case(s) from 1 check(s) (0 failure(s))
[==========] Finished on Wed Jun  3 08:51:46 2020
```

Notice how our regression test is run on every partition of the configured system and for every programming environment.

Now that you have got a first understanding of how a regression test is written in ReFrame, let's try to expand our example.

Inspecting the ReFrame Generated Files

ReFrame generates several files during the execution of a test. When developing or debugging a regression test it is important to be able to locate them and inspect them.

As soon as the *setup* stage of the test is executed, a stage directory specific to this test is generated. All the required resources for the test are copied to this directory, and this will be the working directory for the compilation, running, sanity and performance checking phases. If the test is successful, this stage directory is removed, unless the `--keep-stage-files` option is passed in the command line. Before removing this directory, ReFrame copies the following files to a dedicated output directory for this test:

- The generated build script and its standard output and standard error. This allows you to inspect exactly how

your test was compiled.

- The generated run script and its standard output and standard error. This allows you to inspect exactly how your test was run and verify that the sanity checking was correct.
- Any other user-specified files.

If a regression test fails, its stage directory will not be removed. This allows you to reproduce exactly what ReFrame was trying to perform and will help you debug the problem with your test.

Let's rerun our first example and instruct ReFrame to keep the stage directory of the test, so that we can inspect it.

```
./bin/reframe -C tutorial/config/settings.py -c tutorial/example1.py -r --keep-stage-
↪files
```

ReFrame creates a stage directory for each test case using the following pattern:

```
$STAGEDIR_PREFIX/<system>/<partition>/<prog-environ>/<test-name>
```

Let's pick the test case for the `gpu` partition and the `PrgEnv-gnu` programming environment from our first test to inspect. The default `STAGEDIR_PREFIX` is `./stage`:

```
cd stage/daint/gpu/PrgEnv-gnu/Example1Test/
```

If you do a listing in this directory, you will see all the files contained in the `tutorial/src` directory, as well as the following files:

```
rfm_Example1Test_build.err  rfm_Example1Test_job.err
rfm_Example1Test_build.out  rfm_Example1Test_job.out
rfm_Example1Test_build.sh   rfm_Example1Test_job.sh
```

The `rfm_Example1Test_build.sh` is the generated build script and the `.out` and `.err` are the compilation's standard output and standard error. Here is the generated build script for our first test:

```
#!/bin/bash

_onerror()
{
    exitcode=$?
    echo "-reframe: command \`$BASH_COMMAND' failed (exit code: $exitcode)"
    exit $exitcode
}

trap _onerror ERR

module load daint-gpu
module unload PrgEnv-cray
module load PrgEnv-gnu
cc example_matrix_vector_multiplication.c -o ./Example1Test
```

Similarly, the `rfm_Example1Test_job.sh` is the generated job script and the `.out` and `.err` files are the corresponding standard output and standard error. The generated job script for the test case we are currently inspecting is the following:

```
#!/bin/bash -l
#SBATCH --job-name="rfm_Example1Test_job"
#SBATCH --time=0:10:0
#SBATCH --ntasks=1
#SBATCH --output=rfm_Example1Test_job.out
```

(continues on next page)

(continued from previous page)

```
#SBATCH --error=rfm_Example1Test_job.err
#SBATCH --constraint=gpu
module load daint-gpu
module unload PrgEnv-cray
module load PrgEnv-gnu
srun ./Example1Test 1024 100
```

It is interesting to check here the generated job script for the `login` partition of the example system, which does not use a workload manager:

```
cat stage/daint/login/PrgEnv-gnu/Example1Test/rfm_Example1Test_job.sh
```

```
#!/bin/bash -l
module unload PrgEnv-cray
module load PrgEnv-gnu
 ./Example1Test 1024 100
```

This is one of the advantages in using ReFrame: You do not have to care about the system-level details of the target system that your test is running. Based on its configuration, ReFrame will generate the appropriate commands to run your test.

Customizing the Compilation Phase

In this example, we write a regression test to compile and run the OpenMP version of the matrix-vector product program, that we have shown before. The full code of this test follows:

```
import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class Example2aTest(rfm.RegressionTest):
    def __init__(self):
        self.descr = 'Matrix-vector multiplication example with OpenMP'
        self.valid_systems = ['*']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_openmp.c'
        self.build_system = 'SingleSource'
        self.executable_opts = ['1024', '100']
        self.variables = {
            'OMP_NUM_THREADS': '4'
        }
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    @rfm.run_before('compile')
    def setflags(self):
        env = self.current_environ.name
        if env == 'PrgEnv-cray':
            self.build_system.cflags = ['-homp']
        elif env == 'PrgEnv-gnu':
            self.build_system.cflags = ['-fopenmp']
```

(continues on next page)

(continued from previous page)

```
        elif env == 'PrgEnv-intel':
            self.build_system.cflags = ['-openmp']
        elif env == 'PrgEnv-pgi':
            self.build_system.cflags = ['-mp']
```

This example introduces two new concepts:

1. We need to set the `OMP_NUM_THREADS` environment variable, in order to specify the number of threads to use with our program.
2. We need to specify different flags for the different compilers provided by the programming environments we are testing. Notice also that we now restrict the validity of our test only to the programming environments that we know how to handle (see the *`valid_prog_environs`*).

To define environment variables to be set during the execution of a test, you should use the *`variables`* attribute of the regression test. This is a dictionary, whose keys are the names of the environment variables and whose values are the values of the environment variables. Notice that both the keys and the values must be strings.

From version 2.14, ReFrame manages compilation of tests through the concept of build systems. Any customization of the build process should go through a build system. For straightforward cases, as in our first example, where no customization is needed, ReFrame automatically picks the correct build system to build the code. In this example, however, we want to set the flags for compiling the OpenMP code. Assuming our test supported only GCC, we could simply add the following lines in the `__init__()` method of our test:

```
self.build_system = 'SingleSource'
self.build_system.cflags = ['-fopenmp']
```

The *`SingleSource`* build system that we use here supports the compilation of a single file only. Each build system type defines a set of variables that the user can set. Based on the selected build system, ReFrame will generate a build script that will be used for building the code. The generated build script can be found in the stage or the output directory of the test, along with the output of the compilation. This way, you may reproduce exactly what ReFrame does in case of any errors. More on the build systems feature can be found here.

Getting back to our test, simply setting the `cflags` to `-fopenmp` globally in the test will make it fail for programming environments other than `PrgEnv-gnu`, since the OpenMP flags vary for the different compilers. Ideally, we need to set the `cflags` differently for each programming environment. To achieve this we need to define a method that will set the compilation flags based on the current programming environment (i.e., the environment that test currently runs with) and schedule it to run before the `compile` stage of the test pipeline as follows:

```
@rfm.run_before('compile')
def setflags(self):
    env = self.current_environ.name
    if env == 'PrgEnv-cray':
        self.build_system.cflags = ['-homp']
    elif env == 'PrgEnv-gnu':
        self.build_system.cflags = ['-fopenmp']
    elif env == 'PrgEnv-intel':
        self.build_system.cflags = ['-openmp']
    elif env == 'PrgEnv-pgi':
        self.build_system.cflags = ['-mp']
```

In this function we retrieve the current environment from the `current_environ` attribute, so we can then differentiate the build system's flags based on its name. Note that we cannot retrieve the current programming environment inside the test's constructor, since as described in in the "*How ReFrame Executes Tests*" page, it is during the setup phase that a regression test is prepared for a new system partition and a new programming environment. The second important thing in this function is the `@run_before('compile')` decorator. This decorator will attach this function to the `compile` stage of the pipeline and will execute it before entering this stage. Similarly to the

@run_before decorator, there is also the *@run_after*, which will run the decorated function after the specified pipeline stage. The decorated function may have any name, but it should be a method of the test taking no arguments (i.e., its sole argument should the `self`).

You may attach multiple functions to the same stage as in the following example:

```
@rfm.run_before('compile')
def setflags(self):
    env = self.current_environ.name
    if env == 'PrgEnv-cray':
        self.build_system.cflags = ['-homp']
    elif env == 'PrgEnv-gnu':
        self.build_system.cflags = ['-fopenmp']
    elif env == 'PrgEnv-intel':
        self.build_system.cflags = ['-openmp']
    elif env == 'PrgEnv-pgi':
        self.build_system.cflags = ['-mp']

@rfm.run_before('compile')
def set_more_flags(self):
    self.build_system.flags += ['-g']
```

In this case, the decorated functions will be executed before the compilation stage in the order that they are defined in the regression test.

There is also the possibility to attach a single function to multiple stages by stacking the *@run_before* or *@run_after* decorators. In the following example `var` will be set to `2` after the setup phase is executed:

```
def __init__(self):
    ...
    self.var = 0


@rfm.run_before('setup')
@rfm.run_after('setup')
def inc(self):
    self.var += 1
```

Another important feature of the hooks syntax, is that hooks are inherited by derived tests, unless you override the function and re-hook it explicitly. In the following example, the `setflags()` will be executed before the compilation phase of the `DerivedTest`:

```
class BaseTest(rfm.RegressionTest):
    def __init__(self):
        ...
        self.build_system = 'Make'

    @rfm.run_before('compile')
    def setflags(self):
        if self.current_environ.name == 'X':
            self.build_system.cppflags = ['-Ifoo']


 @rfm.simple_test
 class DerivedTest(BaseTest):
    def __init__(self):
        super().__init__()
        ...
```

If you override a hooked function in a derived class, the base class' hook will not be executed, unless you explicitly call it with `super()`. In the following example, we completely disable the `setflags()` hook of the base class:

```
@rfm.simple_test
class DerivedTest(BaseTest):
    @rfm.run_before('compile')
    def setflags(self):
        pass
```

Notice that in order to redefine a hook, you need not only redefine the method in the derived class, but you should hook it at the same pipeline phase. Otherwise, the base class hook will be executed.

Warning: Changed in version 3.0: Configuring your test per programming environment and per system partition by overriding the *`setup`* method is deprecated. Please refer to the "*Migrating to ReFrame 3*" guide for more details.

Warning: Changed in version 2.17: Support for setting the compiler flags in the programming environment has been dropped completely.

An alternative implementation using dictionaries

Here we present an alternative implementation of the same test using a dictionary to hold the compilation flags for the different programming environments. The advantage of this implementation is that you move the different compilation flags in the initialization phase, where also the rest of the test's specification is, thus making it more concise.

The `setflags()` method is now very simple: it gets the correct compilation flags from the `prgenv_flags` dictionary and applies them to the build system.

```
import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class Example2bTest(rfm.RegressionTest):
    def __init__(self):
        self.descr = 'Matrix-vector multiplication example with OpenMP'
        self.valid_systems = ['*']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_openmp.c'
        self.build_system = 'SingleSource'
        self.executable_opts = ['1024', '100']
        self.prgenv_flags = {
            'PrgEnv-cray':  ['-homp'],
            'PrgEnv-gnu':   ['-fopenmp'],
            'PrgEnv-intel': ['-openmp'],
            'PrgEnv-pgi':   ['-mp']
        }
        self.variables = {
            'OMP_NUM_THREADS': '4'
        }
        self.sanity_patterns = sn.assert_found(
```

(continues on next page)

(continued from previous page)

```
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    @rfm.run_before('compile')
    def setflags(self):
        self.build_system.cflags = self.prgenv_flags[self.current_environ.name]
```

Tip: A regression test is like any other Python class, so you can freely define your own attributes. If you accidentally try to write on a reserved *RegressionTest* attribute that is not writeable, ReFrame will prevent this and it will throw an error.

Running on Multiple Nodes

So far, all our tests run on a single node. Depending on the actual system that ReFrame is running, the test may run locally or be submitted to the system's job scheduler. In this example, we write a regression test for the MPI+OpenMP version of the matrix-vector product. The source code of this program is in `tutorial/src/example_matrix_vector_multiplication_mpi_openmp.c`. The regression test file follows:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class Example3Test(rfm.RegressionTest):
    def __init__(self):
        self.descr = 'Matrix-vector multiplication example with MPI'
        self.valid_systems = ['daint:gpu', 'daint:mc']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_mpi_openmp.c'
        self.executable_opts = ['1024', '10']
        self.build_system = 'SingleSource'
        self.prgenv_flags = {
            'PrgEnv-cray':  ['-homp'],
            'PrgEnv-gnu':   ['-fopenmp'],
            'PrgEnv-intel': ['-openmp'],
            'PrgEnv-pgi':   ['-mp']
        }
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.num_tasks = 8
        self.num_tasks_per_node = 2
        self.num_cpus_per_task = 4
        self.variables = {
            'OMP_NUM_THREADS': str(self.num_cpus_per_task)
        }
        self.maintainers = ['you-can-type-your-email-here']
```

(continues on next page)

(continued from previous page)

```
        self.tags = {'tutorial'}

    @rfm.run_before('compile')
    def setflags(self):
        self.build_system.cflags = self.prgenv_flags[self.current_environ.name]
```

This test is pretty much similar to the *test example* for the OpenMP code we have shown before, except that it adds some information about the configuration of the distributed tasks. It also restricts the valid systems only to those that support distributed execution. Let's take the changes step-by-step:

First we need to specify for which partitions this test is meaningful by setting the `valid_systems` attribute:

```
self.valid_systems = ['daint:gpu', 'daint:mc']
```

We only specify the partitions that are configured with a job scheduler. If we try to run the generated executable on the login nodes, it will fail. So we remove this partition from the list of the supported systems.

The most important addition to this check are the variables controlling the distributed execution:

```
self.num_tasks = 8
self.num_tasks_per_node = 2
self.num_cpus_per_task = 4
```

By setting these variables, we specify that this test should run with 8 MPI tasks in total, using two tasks per node. Each task may use four logical CPUs. Based on these variables ReFrame will generate the appropriate scheduler flags to meet that requirement. For example, for Slurm these variables will result in the following flags: `--ntasks=8`, `--ntasks-per-node=2` and `--cpus-per-task=4`. ReFrame provides several more variables for configuring the job submission. As shown in the following Table, they follow closely the corresponding Slurm options. For schedulers that do not provide the same functionality, some of the variables may be ignored.

`RegressionTest` attribute	Corresponding SLURM option
`time_limit = '10m30s`	`--time=00:10:30`
`use_multithreading = True`	`--hint=multithread`
`use_multithreading = False`	`--hint=nomultithread`
`exclusive_access = True`	`--exclusive`
`num_tasks=72`	`--ntasks=72`
`num_tasks_per_node=36`	`--ntasks-per-node=36`
`num_cpus_per_task=4`	`--cpus-per-task=4`
`num_tasks_per_core=2`	`--ntasks-per-core=2`
`num_tasks_per_socket=36`	`--ntasks-per-socket=36`

Testing a GPU Code

In this example, we will create two regression tests for two different GPU versions of our matrix-vector code: OpenACC and CUDA. Let's start with the OpenACC regression test:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn
```

(continues on next page)

(continued from previous page)

```
@rfm.simple_test
class Example4Test(rfm.RegressionTest):
    def __init__(self):
        self.descr = 'Matrix-vector multiplication example with OpenACC'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_openacc.c'
        self.build_system = 'SingleSource'
        self.executable_opts = ['1024', '100']
        self.modules = ['craype-accel-nvidia60']
        self.num_gpus_per_node = 1
        self.prgenv_flags = {
            'PrgEnv-cray': ['-hacc', '-hnoomp'],
            'PrgEnv-pgi':  ['-acc', '-ta=tesla:cc60']
        }
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    @rfm.run_before('compile')
    def setflags(self):
        self.build_system.cflags = self.prgenv_flags[self.current_environ.name]
```

The things to notice in this test are the restricted list of system partitions and programming environments that this test supports and the use of the *modules* variable:

```
self.modules = ['craype-accel-nvidia60']
```

The *modules* variable takes a list of environment modules that should be loaded during the setup phase of the test. In this particular test, we need to load the `craype-accel-nvidia60` module, which enables the generation of a GPU binary from an OpenACC code.

It's advisable for GPU-enabled tests to define also the *num_gpus_per_node* attribute, since this information may be needed by some system partitions in order to get the right nodes:

```
self.num_gpus_per_node = 1
```

The regression test for the CUDA code is slightly simpler:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class Example5Test(rfm.RegressionTest):
    def __init__(self):
        self.descr = 'Matrix-vector multiplication example with CUDA'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu', 'PrgEnv-pgi']
```

(continues on next page)

(continued from previous page)

```
        self.sourcepath = 'example_matrix_vector_multiplication_cuda.cu'
        self.executable_opts = ['1024', '100']
        self.modules = ['cudatoolkit']
        self.num_gpus_per_node = 1
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}
```

ReFrame will recognize the `.cu` extension of the source file and it will try to invoke `nvcc` for compiling the code. In this case, there is no need to differentiate across the programming environments, since the compiler will be eventually the same. `nvcc` in our example is provided by the `cudatoolkit` module, which we list it in the *`modules`* variable.

More Advanced Sanity Checking

So far we have done a very simple sanity checking. We are only looking if a specific line is present in the output of the test program. In this example, we expand the regression test of the serial code, so as to check also if the printed norm of the result vector is correct.

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class Example6Test(rfm.RegressionTest):
    def __init__(self):
        self.descr = 'Matrix-vector multiplication with L2 norm check'
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.sourcepath = 'example_matrix_vector_multiplication.c'

        matrix_dim = 1024
        iterations = 100
        self.executable_opts = [str(matrix_dim), str(iterations)]

        expected_norm = matrix_dim
        found_norm = sn.extractsingle(
            r'The L2 norm of the resulting vector is:\s+(?P<norm>\S+)',
            self.stdout, 'norm', float)
        self.sanity_patterns = sn.all([
            sn.assert_found(
                r'time for single matrix vector multiplication', self.stdout),
            sn.assert_lt(sn.abs(expected_norm - found_norm), 1.0e-6)
        ])
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}
```

The only difference with our first example is actually the more complex expression to assess the sanity of the test. Let's go over it line-by-line. The first thing we do is to extract the norm printed in the standard output.

```
found_norm = sn.extractsingle(
    r'The L2 norm of the resulting vector is:\s+(?P<norm>\S+)',
    self.stdout, 'norm', float)
```

The *extractsingle* sanity function extracts some information from a single occurrence (by default the first) of a pattern in a filename. In our case, this function will extract the `norm` capturing group from the match of the regular expression `r'The L2 norm of the resulting vector is:\s+(?P<norm>\S+)'` in standard output, it will convert it to float and it will return it. Unnamed capturing groups in regular expressions are also supported, which you can reference by their group number. For example, we could have written the same statement as follows:

```
found_norm = sn.extractsingle(
    r'The L2 norm of the resulting vector is:\s+(\S+)',
    self.stdout, 1, float)
```

Notice that we replaced the `'norm'` argument with `1`, which is the capturing group number.

Note: In regular expressions, capturing group `0` corresponds always to the whole match. In sanity functions dealing with regular expressions, this will yield the whole line that matched.

A useful counterpart of *extractsingle* is the *extractall* function, which instead of a single occurrence, returns a list of all the occurrences found. For a more detailed description of this and other sanity functions, please refer to the sanity function reference.

The next four lines is the actual sanity check:

```
self.sanity_patterns = sn.all([
    sn.assert_found(
        r'time for single matrix vector multiplication', self.stdout),
    sn.assert_lt(sn.abs(expected_norm - found_norm), 1.0e-6)
])
```

This expression combines two conditions that need to be true, in order for the sanity check to succeed:

1. Find in standard output the same line we were looking for already in the first example.
2. Verify that the printed norm does not deviate significantly from the expected value.

The *all()* function is responsible for combining the results of the individual subexpressions. It is essentially the Python built-in `all()` function, exposed as a sanity function, and requires that all the elements of the iterable it takes as an argument evaluate to `True`. As mentioned before, all the `assert_*()` functions either return `True` on success or raise `SanityError`. So, if everything goes smoothly, *sn.all()* will evaluate to `True` and sanity checking will succeed.

The expression for the second condition is more interesting. Here, we want to assert that the absolute value of the difference between the expected and the found norm are below a certain value. The important thing to mention here is that you can combine the results of sanity functions in arbitrary expressions, use them as arguments to other functions, return them from functions, assign them to variables etc. Remember that sanity functions are not evaluated at the time you call them. They will be evaluated later by the framework during the sanity checking phase. If you include the result of a sanity function in an expression, the evaluation of the resulting expression will also be deferred. For a detailed description of the mechanism behind the sanity functions, please have a look at the "*Understanding the Mechanism of Sanity Functions*" page.

Writing a Performance Test

An important aspect of regression testing is checking for performance regressions. ReFrame offers a flexible way of extracting and manipulating performance data from the program output, as well as a comprehensive way of setting performance thresholds per system and system partitions.

In this example, we extend the CUDA test presented *previously*, so as to check also the performance of the matrix-vector multiplication.

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class Example7Test(rfm.RegressionTest):
    def __init__(self):
        self.descr = 'Matrix-vector multiplication (CUDA performance test)'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-gnu', 'PrgEnv-cray', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_cuda.cu'
        self.build_system = 'SingleSource'
        self.build_system.cxxflags = ['-O3']
        self.executable_opts = ['4096', '1000']
        self.modules = ['cudatoolkit']
        self.num_gpus_per_node = 1
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.perf_patterns = {
            'perf': sn.extractsingle(r'Performance:\s+(?P<Gflops>\S+) Gflop/s',
                                     self.stdout, 'Gflops', float)
        }
        self.reference = {
            'daint:gpu': {
                'perf': (50.0, -0.1, 0.1, 'Gflop/s'),
            }
        }
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}
```

The are two new variables set in this test that basically enable the performance testing:

- *perf_patterns* This variable defines which are the performance patterns we are looking for and how to extract the performance values.
- *reference* This variable is a collection of reference values for different systems.

Let's have a closer look at each of them:

```
self.perf_patterns = {
    'perf': sn.extractsingle(r'Performance:\s+(?P<Gflops>\S+) Gflop/s',
                             self.stdout, 'Gflops', float)
}
```

The *perf_patterns* attribute is a dictionary, whose keys are *performance variables* (i.e., arbitrary names assigned to the performance values we are looking for), and its values are *sanity expressions* that specify how to obtain these

performance values from the output. A sanity expression is a Python expression that uses the result of one or more *sanity functions*. In our example, we name the performance value we are looking for simply as `perf` and we extract its value by converting to float the regex capturing group named `Gflops` from the line that was matched in the standard output.

Each of the performance variables defined in *`perf_patterns`* must be resolved in the *`reference`* dictionary of reference values. When the framework obtains a performance value from the output of the test it searches for a reference value in the *`reference`* dictionary, and then it checks whether the user supplied tolerance is respected. Let's go over the *`reference`* dictionary of our example and explain its syntax in more detail:

```
self.reference = {
    'daint:gpu': {
        'perf': (50.0, -0.1, 0.1, 'Gflop/s'),
    }
}
```

This is a special type of dictionary that we call *scoped dictionary*, because it defines scopes for its keys. In order to resolve a reference value for a performance variable, ReFrame creates the following key `<current_sys>:<current_part>:<perf_variable>` and looks it up inside the *`reference`* dictionary. If our example, since this test is only allowed to run on the `daint:gpu` partition of our system, ReFrame will look for the `daint:gpu:perf` reference key. The `perf` subkey will then be searched in the following scopes in this order: `daint:gpu`, `daint`, `*`. The first occurrence will be used as the reference value of the `perf` performance variable. In our example, the `perf` key will be resolved in the `daint:gpu` scope giving us the reference value.

Reference values in ReFrame are specified as a four-tuple comprising the reference value, the lower and upper thresholds and the measurement unit. If no unit is relevant, then you have to insert `None` explicitly. Thresholds are specified as decimal fractions of the reference value. For nonnegative reference values, the lower threshold must lie in the [-1,0], whereas the upper threshold may be any positive real number or zero. In our example, the reference value for this test on `daint:gpu` is 50 Gflop/s $\pm$10%. Setting a threshold value to `None` disables the threshold. If you specify a measurement unit as well, you will be able to log it in the performance logs of the test; this is handy when you are inspecting or plotting the performance values.

ReFrame will always add a default `*` entry in the `reference` dictionary, if it does not exist, with the reference value of `(0, None, None, <unit>)`, where `unit` is derived from the unit of each respective performance variable. This is useful when using ReFrame for benchmarking purposes and you would like to run a test on an unknown system.

Note: New in version 2.16: Reference tuples may now optionally contain units.

New in version 2.19: A default `*` entry is now always added to the reference dictionary.

Changed in version 3.0: Reference tuples now require the measurement unit.

Combining It All Together

As we have mentioned before and as you have already experienced with the examples in this tutorial, regression tests in ReFrame are written in pure Python. As a result, you can leverage the language features and capabilities to organize better your tests and decrease the maintenance cost. In this example, we are going to reimplement all the tests of the tutorial with much less code and in a single file. Here is the final example code that combines all the tests discussed before:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

```

(continues on next page)

(continued from previous page)

```
import reframe as rfm
import reframe.utility.sanity as sn


class BaseMatrixVectorTest(rfm.RegressionTest):
    def __init__(self, test_version):
        self.descr = '%s matrix-vector multiplication' % test_version
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.build_system = 'SingleSource'
        self.prgenv_flags = None

        matrix_dim = 1024
        iterations = 100
        self.executable_opts = [str(matrix_dim), str(iterations)]

        expected_norm = matrix_dim
        found_norm = sn.extractsingle(
            r'The L2 norm of the resulting vector is:\s+(?P<norm>\S+)',
            self.stdout, 'norm', float)
        self.sanity_patterns = sn.all([
            sn.assert_found(
                r'time for single matrix vector multiplication', self.stdout),
            sn.assert_lt(sn.abs(expected_norm - found_norm), 1.0e-6)
        ])
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    @rfm.run_before('compile')
    def setflags(self):
        if self.prgenv_flags is not None:
            env = self.current_environ.name
            self.build_system.cflags = self.prgenv_flags[env]


@rfm.simple_test
class SerialTest(BaseMatrixVectorTest):
    def __init__(self):
        super().__init__('Serial')
        self.sourcepath = 'example_matrix_vector_multiplication.c'


@rfm.simple_test
class OpenMPTest(BaseMatrixVectorTest):
    def __init__(self):
        super().__init__('OpenMP')
        self.sourcepath = 'example_matrix_vector_multiplication_openmp.c'
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.prgenv_flags = {
            'PrgEnv-cray':  ['-homp'],
            'PrgEnv-gnu':   ['-fopenmp'],
            'PrgEnv-intel': ['-openmp'],
            'PrgEnv-pgi':   ['-mp']
        }
        self.variables = {
            'OMP_NUM_THREADS': '4'
```

(continues on next page)

(continued from previous page)

```
        }


@rfm.simple_test
class MPITest(BaseMatrixVectorTest):
    def __init__(self):
        super().__init__('MPI')
        self.valid_systems = ['daint:gpu', 'daint:mc']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_mpi_openmp.c'
        self.prgenv_flags = {
            'PrgEnv-cray':  ['-homp'],
            'PrgEnv-gnu':   ['-fopenmp'],
            'PrgEnv-intel': ['-openmp'],
            'PrgEnv-pgi':   ['-mp']
        }
        self.num_tasks = 8
        self.num_tasks_per_node = 2
        self.num_cpus_per_task = 4
        self.variables = {
            'OMP_NUM_THREADS': str(self.num_cpus_per_task)
        }


@rfm.simple_test
class OpenACCTest(BaseMatrixVectorTest):
    def __init__(self):
        super().__init__('OpenACC')
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_openacc.c'
        self.modules = ['craype-accel-nvidia60']
        self.num_gpus_per_node = 1
        self.prgenv_flags = {
            'PrgEnv-cray': ['-hacc', '-hnoomp'],
            'PrgEnv-pgi':  ['-acc', '-ta=tesla:cc60']
        }


@rfm.simple_test
class CudaTest(BaseMatrixVectorTest):
    def __init__(self):
        super().__init__('CUDA')
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-gnu', 'PrgEnv-cray', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_cuda.cu'
        self.modules = ['cudatoolkit']
        self.num_gpus_per_node = 1
```

This test abstracts away the common functionality found in almost all of our tutorial tests (executable options, sanity checking, etc.) to a base class, from which all the concrete regression tests derive. Each test then redefines only the parts that are specific to it. Notice also that only the actual tests, i.e., the derived classes, are made visible to the framework through the *@simple_test* decorator. Decorating the base class has no meaning, because it does not correspond to an actual test.

The total line count of this refactored example is less than half of that of the individual tutorial tests. Another interesting

thing to note here is the base class accepting additional additional parameters to its constructor, so that the concrete subclasses can initialize it based on their needs.

Summary

This concludes the first ReFrame tutorial. We have covered all basic aspects of writing regression tests in ReFrame and you should now be able to start experimenting by writing your first useful tests. The next tutorial covers further topics in customizing a regression test to your needs.

2.3.2 Tutorial 2: Customizing Further a Regression Test

In this section, we are going to show some more elaborate use cases of ReFrame. The corresponding scripts as well as the source code of the examples discussed here can be found in the directory `tutorial/advanced/`.

Working with Makefiles

We have already shown how you can compile a single source file associated with your regression test. In this example, we show how ReFrame can leverage Makefiles to build executables.

Compiling a regression test through a Makefile is straightforward with ReFrame. If the *`sourcepath`* attribute refers to a directory, then ReFrame will try to figure out the type of project and select the correct build system. If it is not a CMake or Autotools based project, it will try to use `make` for building it. More specifically, ReFrame first copies the *`sourcesdir`* to the stage directory at the beginning of the compilation phase, then switches to the `{stagedir}/{sourcepath}/` and, finally, invokes `make`.

Note: The *`sourcepath`* attribute must be a relative path refering to a subdirectory of *`sourcesdir`*, i.e., relative paths starting with `..` will be rejected.

By default, *`sourcepath`* is the empty string and *`sourcesdir`* is set to `'src/'`, if such a directory exists. As a result, by not specifying a *`sourcepath`* at all, ReFrame will eventually compile the files found in the `src/` directory. This is exactly what our first example here does.

For completeness, here are the contents of `Makefile` provided:

```
EXECUTABLE := advanced_example1

.SUFFIXES: .o .c

OBJS := advanced_example1.o

$(EXECUTABLE): $(OBJS)
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^

$(OBJS): advanced_example1.c
        $(CC) $(CPPFLAGS) $(CFLAGS) -c $(LDFLAGS) -o $@ $^
```

The corresponding `tutorial/advanced/src/advanced_example1.c` source file consists of a simple printing of a message, whose content depends on the preprocessor variable `MESSAGE`:

```
#include <stdio.h>

int main(){
```

(continues on next page)

(continued from previous page)

```
#ifdef MESSAGE
    char *message = "SUCCESS";
#else
    char *message = "FAILURE";
#endif
    printf("Setting of preprocessor variable: %s\n", message);
    return 0;
}
```

The purpose of the regression test in this case is to set the preprocessor variable `MESSAGE` via `CPPFLAGS` and then check the standard output for the message `SUCCESS`, which indicates that the preprocessor flag has been passed and processed correctly by the Makefile.

The contents of this regression test are the following:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class MakefileTest(rfm.RegressionTest):
    def __init__(self):
        self.descr = ('ReFrame tutorial demonstrating the use of Makefiles '
                      'and compile options')
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.executable = './advanced_example1'
        self.build_system = 'Make'
        self.build_system.cppflags = ['-DMESSAGE']
        self.sanity_patterns = sn.assert_found('SUCCESS', self.stdout)
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}
```

The important bit here is how we set up the build system for this test:

```
    self.build_system = 'Make'
    self.build_system.cppflags = ['-DMESSAGE']
```

First, we set the build system to *Make* and then set the preprocessor flags for the compilation. ReFrame will invoke `make` as follows:

```
make -j 1 CC='cc' CXX='CC' FC='ftn' NVCC='nvcc' CPPFLAGS='-DMESSAGE'
```

The compiler variables (`CC`, `CXX` etc.) are set based on the corresponding values specified in the coniguration of the current environment. You may instruct the build system to ignore the default values from the environment by setting the following:

```
self.build_system.flags_from_environ = False
```

In this case, `make` will be invoked as follows:

```
make -j 1 CPPFLAGS='-DMESSAGE'
```

Notice that the `-j 1` option is always generated. You may change the maximum build concurrency as follows:

```
self.build_system.max_concurrency = 4
```

By setting *max_concurrency* to `None`, no limit for concurrent parallel jobs will be placed. This means that `make -j` will be used for building.

Finally, you may also customize the name of the `Makefile`. You can achieve that by setting the corresponding variable of the *Make* build system:

```
self.build_system.makefile = 'Makefile_custom'
```

More details on ReFrame's build systems, you may find here.

Retrieving the source code from a Git repository

It might be the case that a regression test needs to clone its source code from a remote repository. This can be achieved in two ways with ReFrame. One way is to set the `sourcesdir` attribute to `None` and explicitly clone or checkout a repository using the *prebuild_cmds*:

```
self.sourcesdir = None
self.prebuild_cmds = ['git clone https://github.com/me/myrepo .']
```

By setting `sourcesdir` to `None`, you are telling ReFrame that you are going to provide the source files in the stage directory. The working directory of the `prebuild_cmds` and `postbuild_cmds` commands will be the stage directory of the test.

An alternative way to retrieve specifically a Git repository is to assign its URL directly to the `sourcesdir` attribute:

```
self.sourcesdir = 'https://github.com/me/myrepo'
```

ReFrame will attempt to clone this repository inside the stage directory by executing `git clone <repo> .` and will then procede with the compilation as described above.

Note: ReFrame recognizes only URLs in the `sourcesdir` attribute and requires passwordless access to the repository. This means that the SCP-style repository specification will not be accepted. You will have to specify it as URL using the `ssh://` protocol (see Git documentation page).

Add a configuration step before compiling the code

It is often the case that a configuration step is needed before compiling a code with `make`. To address this kind of projects, ReFrame aims to offer specific abstractions for "configure-make" style of build systems. It supports CMake-based projects through the *CMake* build system, as well as Autotools-based projects through the *Autotools* build system.

For other build systems, you can achieve the same effect using the *Make* build system and the *prebuild_cmds* for performing the configuration step. The following code snippet will configure a code with `./custom_configure` before invoking `make`:

```
self.prebuild_cmds = ['./custom_configure -with-mylib']
self.build_system = 'Make'
self.build_system.cppflags = ['-DHAVE_FOO']
self.build_system.flags_from_environ = False
```

The generated build script then will have the following lines:

```
./custom_configure -with-mylib
make -j 1 CPPFLAGS='-DHAVE_FOO'
```

Implementing a Run-Only Regression Test

There are cases when it is desirable to perform regression testing for an already built executable. The following test uses the `echo` Bash shell command to print a random integer between specific lower and upper bounds. Here is the full regression test:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class ExampleRunOnlyTest(rfm.RunOnlyRegressionTest):
    def __init__(self):
        self.descr = ('ReFrame tutorial demonstrating the class'
                      'RunOnlyRegressionTest')
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.sourcesdir = None

        lower = 90
        upper = 100
        self.executable = 'echo "Random: $((RANDOM%({1}+1-{0})+{0}))"'.format(
            lower, upper)
        self.sanity_patterns = sn.assert_bounded(sn.extractsingle(
            r'Random: (?P<number>\S+)', self.stdout, 'number', float),
            lower, upper)
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}
```

There is nothing special for this test compared to those presented earlier except that it derives from the *RunOnlyRegressionTest* and that it does not contain any resources (`self.sourcesdir = None`). Note that run-only regression tests may also have resources, as for instance a precompiled executable or some input data. The copying of these resources to the stage directory is performed at the beginning of the run phase. For standard regression tests, this happens at the beginning of the compilation phase, instead. Furthermore, in this particular test the *executable* consists only of standard Bash shell commands. For this reason, we can set *sourcesdir* to `None` informing ReFrame that the test does not have any resources.

Note: Changed in version 3.0: It is no more necessary to explicitly set *sourcesdir* to `None` for run-only tests without resources.

Implementing a Compile-Only Regression Test

ReFrame provides the option to write compile-only tests which consist only of a compilation phase without a specified executable. This kind of tests must derive from the *CompileOnlyRegressionTest* class provided by the framework. The following example reuses the code of our first example in this section and checks that no warnings are issued by the compiler:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class ExampleCompileOnlyTest(rfm.CompileOnlyRegressionTest):
    def __init__(self):
        self.descr = ('ReFrame tutorial demonstrating the class'
                      'CompileOnlyRegressionTest')
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.sanity_patterns = sn.assert_not_found('warning', self.stderr)
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}
```

The important thing to note here is that the standard output and standard error of the tests, accessible through the *stdout* and *stderr* attributes, are now the corresponding those of the compilation command. So sanity checking can be done in exactly the same way as with a normal test.

Leveraging Environment Variables

We have already demonstrated in "*Tutorial 1: The Basics*" that ReFrame allows you to load the required modules for regression tests and also set any needed environment variables. When setting environment variables for your test through the *variables* attribute, you can assign them values of other, already defined, environment variables using the standard notation `$OTHER_VARIABLE` or `${OTHER_VARIABLE}`. The following regression test sets the `CUDA_HOME` environment variable to the value of the `CUDATOOLKIT_HOME` and then compiles and runs a simple program:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class EnvironmentVariableTest(rfm.RegressionTest):
    def __init__(self):
        self.descr = ('ReFrame tutorial demonstrating the use'
                      'of environment variables provided by loaded modules')
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['*']
```

(continues on next page)

(continued from previous page)

```
        self.modules = ['cudatoolkit']
        self.variables = {'CUDA_HOME': '$CUDATOOLKIT_HOME'}
        self.executable = './advanced_example4'
        self.build_system = 'Make'
        self.build_system.makefile = 'Makefile_example4'
        self.sanity_patterns = sn.assert_found(r'SUCCESS', self.stdout)
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}
```

Before discussing this test in more detail, let's first have a look in the source code and the Makefile of this example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef CUDA_HOME
#   define CUDA_HOME ""
#endif

int main() {
    char *cuda_home_compile = CUDA_HOME;
    char *cuda_home_runtime = getenv("CUDA_HOME");
    if (cuda_home_runtime &&
        strnlen(cuda_home_runtime, 256) &&
        strnlen(cuda_home_compile, 256) &&
        !strncmp(cuda_home_compile, cuda_home_runtime, 256)) {
        printf("SUCCESS\n");
    } else {
        printf("FAILURE\n");
        printf("Compiled with CUDA_HOME=%s, ran with CUDA_HOME=%s\n",
               cuda_home_compile,
               cuda_home_runtime ? cuda_home_runtime : "<null>");
    }

    return 0;
}
```

This program is pretty basic, but enough to demonstrate the use of environment variables from ReFrame. It simply compares the value of the `CUDA_HOME` macro with the value of the environment variable `CUDA_HOME` at runtime, printing `SUCCESS` if they are not empty and match. The Makefile for this example compiles this source by simply setting `CUDA_HOME` to the value of the `CUDA_HOME` environment variable:

```
EXECUTABLE := advanced_example4

CPPFLAGS = -DCUDA_HOME=\"$(CUDA_HOME)\"

.SUFFIXES: .o .c

OBJS := advanced_example4.o

$(EXECUTABLE): $(OBJS)
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^

$(OBJS): advanced_example4.c
        $(CC) $(CPPFLAGS) $(CFLAGS) -c $(LDFLAGS) -o $@ $^
```

(continues on next page)

(continued from previous page)

```
clean:
        /bin/rm -f $(OBJS) $(EXECUTABLE)
```

Coming back now to the ReFrame regression test, the `CUDATOOLKIT_HOME` environment variable is defined by the `cudatoolkit` module. If you try to run the test, you will see that it will succeed, meaning that the `CUDA_HOME` variable was set correctly both during the compilation and the runtime. ReFrame will generate the following instructions in the shell scripts (build and run) associated with this test:

```
module load cudatoolkit
export CUDA_HOME=$CUDATOOLKIT_HOME
```

Finally, as already mentioned *previously*, since the name of the makefile is not one of the standard ones, it must be set explicitly in the build system:

```
self.build_system.makefile = 'Makefile_example4'
```

Setting a Time Limit for Regression Tests

ReFrame gives you the option to limit the execution time of regression tests. The following example demonstrates how you can achieve this by limiting the execution time of a test that tries to sleep 100 seconds:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class TimeLimitTest(rfm.RunOnlyRegressionTest):
    def __init__(self):
        self.descr = ('ReFrame tutorial demonstrating the use '
                      'of a user-defined time limit')
        self.valid_systems = ['daint:gpu', 'daint:mc']
        self.valid_prog_environs = ['*']
        self.time_limit = '1m'
        self.executable = 'sleep'
        self.executable_opts = ['100']
        self.sanity_patterns = sn.assert_found(
            r'CANCELLED.*DUE TO TIME LIMIT', self.stderr)
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}
```

The important bit here is the following line that sets the time limit for the test to one minute:

```
self.time_limit = '1m'
```

The *time_limit* attribute is a string in the form `<DAYS>d<HOURS>h<MINUTES>m<SECONDS>s` and will impose a *runtime* limit to the associated job. It will not kill the test if the job is pending for more time. Time limits are implemented for all the scheduler backends.

The sanity condition for this test verifies that associated job has been canceled due to the time limit (note that this message is SLURM-specific).

```
self.sanity_patterns = sn.assert_found(
    r'CANCELLED.*DUE TO TIME LIMIT', self.stderr)
```

Note: New in version 3.0: The *max_pending_time* attribute was added to force termination of a test if it is pending for too long.

Applying a sanity function iteratively

It is often the case that a common sanity pattern has to be applied many times. In this example we will demonstrate how the above situation can be easily tackled using the sanity functions offered by ReFrame. Specifically, we would like to execute the following shell script and check that its output is correct:

```
#!/usr/bin/env bash

if [ -z $LOWER ]; then
    export LOWER=90
fi

if [ -z $UPPER ]; then
    export UPPER=100
fi

for i in {1..100}; do
    echo Random: $((RANDOM%($UPPER+1-$LOWER)+$LOWER))
done
```

The above script simply prints 100 random integers between the limits given by the variables `LOWER` and `UPPER`. In the corresponding regression test we want to check that all the random numbers printed lie between 90 and 100 ensuring that the script executed correctly. Hence, a common sanity check has to be applied to all the printed random numbers. In ReFrame this can achieved by the use of *map()* sanity function accepting a function and an iterable as arguments. Through *map()* the given function will be applied to all the members of the iterable object. Note that since *map()* is a sanity function, its execution will be deferred. The ReFrame test is the following:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class DeferredIterationTest(rfm.RunOnlyRegressionTest):
    def __init__(self):
        self.descr = ('ReFrame tutorial demonstrating the use of deferred '
                      'iteration via the `map` sanity function.')
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.executable = './random_numbers.sh'
        numbers = sn.extractall(
            r'Random: (?P<number>\S+)', self.stdout, 'number', float)
        self.sanity_patterns = sn.and_(
```

(continues on next page)

(continued from previous page)

```
            sn.assert_eq(sn.count(numbers), 100),
            sn.all(sn.map(lambda x: sn.assert_bounded(x, 90, 100), numbers)))
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}
```

First the random numbers are extracted through the *extractall()* function as follows:

```
numbers = sn.extractall(
    r'Random: (?P<number>\S+)', self.stdout, 'number', float)
```

The `numbers` variable is a deferred iterable, which upon evaluation will return all the extracted numbers. In order to check that the extracted numbers lie within the specified limits, we make use of the *map()* sanity function, which will apply the *assert_bounded()* to all the elements of `numbers`.

There is still a small complication that needs to be addressed. The *all()* function returns `True` for empty iterables, which is not what we want. So we must ensure that all the numbers are extracted as well. To achieve this, we make use of *count()* to get the number of elements contained in `numbers` combined with *assert_eq()* to check that the number is indeed 100. Finally, both of the above conditions have to be satisfied for the program execution to be considered successful, hence the use of the *and_()* function. Note that the `and` operator is not deferrable and will trigger the evaluation of any deferrable argument passed to it. The full syntax for the `sanity_patterns` is the following:

```
self.sanity_patterns = sn.and_(
    sn.assert_eq(sn.count(numbers), 100),
    sn.all(sn.map(lambda x: sn.assert_bounded(x, 90, 100), numbers)))
```

Note: New in version 2.13: ReFrame offers also the *allx()* sanity function which, conversely to the builtin `all()` function, will return `False` if its iterable argument is empty.

Customizing the Generated Job Script

It is often the case that you must run some commands before or after the parallel launch of your executable. This can be easily achieved by using the *prerun_cmds* and *postrun_cmds* attributes of a ReFrame test.

The following example is a slightly modified version of the previous one. The lower and upper limits for the random numbers are now set inside a helper shell script in `scripts/limits.sh` and we want also to print the word `FINISHED` after our executable has finished. In order to achieve this, we need to source the helper script just before launching the executable and `echo` the desired message just after it finishes. Here is the test file:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class PrerunDemoTest(rfm.RunOnlyRegressionTest):
    def __init__(self):
        self.descr = ('ReFrame tutorial demonstrating the use of '
                      'pre- and post-run commands')
```

(continues on next page)

(continued from previous page)

```
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.prerun_cmds  = ['source scripts/limits.sh']
        self.postrun_cmds = ['echo FINISHED']
        self.executable = './random_numbers.sh'
        numbers = sn.extractall(
            r'Random: (?P<number>\S+)', self.stdout, 'number', float)
        self.sanity_patterns = sn.all([
            sn.assert_eq(sn.count(numbers), 100),
            sn.all(sn.map(lambda x: sn.assert_bounded(x, 50, 80), numbers)),
            sn.assert_found('FINISHED', self.stdout)
        ])
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}
```

Notice the use of the `prerun_cmds` and `postrun_cmds` attributes. These are lists of shell commands that are emitted verbatim in the job script. The generated job script for this example is the following:

```
#!/bin/bash -l
#SBATCH --job-name="prerun_demo_check_daint_gpu_PrgEnv-gnu"
#SBATCH --time=0:10:0
#SBATCH --ntasks=1
#SBATCH --output=prerun_demo_check.out
#SBATCH --error=prerun_demo_check.err
#SBATCH --constraint=gpu
module load daint-gpu
module unload PrgEnv-cray
module load PrgEnv-gnu
source scripts/limits.sh
srun ./random_numbers.sh
echo FINISHED
```

ReFrame generates the job shell script using the following pattern:

```
#!/bin/bash -l
{job_scheduler_preamble}
{test_environment}
{prerun_cmds}
{parallel_launcher} {executable} {executable_opts}
{postrun_cmds}
```

The `job_scheduler_preamble` contains the directives that control the job allocation. The `test_environment` are the necessary commands for setting up the environment of the test. This is the place where the modules and environment variables specified in *`modules`* and *`variables`* attributes are emitted. Then the commands specified in *`prerun_cmds`* follow, while those specified in the *`postrun_cmds`* come after the launch of the parallel job. The parallel launch itself consists of three parts:

1. The parallel launcher program (e.g., `srun`, `mpirun` etc.) with its options,
2. the regression test executable as specified in the *`executable`* attribute and
3. the options to be passed to the executable as specified in the *`executable_opts`* attribute.

A key thing to note about the generated job script is that ReFrame submits it from the stage directory of the test, so that all relative paths are resolved against it.

Working with parameterized tests

New in version 2.13.

We have seen already in the basic tutorial how we could better organize the tests so as to avoid code duplication by using test class hierarchies. An alternative technique, which could also be used in parallel with the class hierarchies, is to use *parameterized tests*. The following is a test that takes a `variant` parameter, which controls which variant of the code will be used. Depending on that value, the test is set up differently:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.parameterized_test(['MPI'], ['OpenMP'])
class MatrixVectorTest(rfm.RegressionTest):
    def __init__(self, variant):
        self.descr = 'Matrix-vector multiplication test (%s)' % variant
        self.valid_systems = ['daint:gpu', 'daint:mc']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.build_system = 'SingleSource'
        self.prgenv_flags = {
            'PrgEnv-cray':  ['-homp'],
            'PrgEnv-gnu':   ['-fopenmp'],
            'PrgEnv-intel': ['-openmp'],
            'PrgEnv-pgi':   ['-mp']
        }

        if variant == 'MPI':
            self.num_tasks = 8
            self.num_tasks_per_node = 2
            self.num_cpus_per_task = 4
            self.sourcepath = 'example_matrix_vector_multiplication_mpi_openmp.c'
        elif variant == 'OpenMP':
            self.sourcepath = 'example_matrix_vector_multiplication_openmp.c'
            self.num_cpus_per_task = 4

        self.variables = {
            'OMP_NUM_THREADS': str(self.num_cpus_per_task)
        }
        matrix_dim = 1024
        iterations = 100
        self.executable_opts = [str(matrix_dim), str(iterations)]

        expected_norm = matrix_dim
        found_norm = sn.extractsingle(
            r'The L2 norm of the resulting vector is:\s+(?P<norm>\S+)',
            self.stdout, 'norm', float)
        self.sanity_patterns = sn.all([
            sn.assert_found(
                r'time for single matrix vector multiplication', self.stdout),
            sn.assert_lt(sn.abs(expected_norm - found_norm), 1.0e-6)
        ])
```

(continues on next page)

(continued from previous page)

```
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    @rfm.run_before('compile')
    def setflags(self):
        if self.prgenv_flags is not None:
            env = self.current_environ.name
            self.build_system.cflags = self.prgenv_flags[env]
```

If you have already gone through the *Tutorial 1: The Basics*, this test can be easily understood. The new bit here is the `@parameterized_test` decorator of the `MatrixVectorTest` class. This decorator takes an arbitrary number of arguments, which are either of a sequence type (i.e., list, tuple etc.) or of a mapping type (i.e., dictionary). Each of the decorator's arguments corresponds to the constructor arguments of the decorated test that will be used to instantiate it. In the example shown, the test will be instantiated twice, once passing `variant` as `MPI` and a second time with `variant` passed as `OpenMP`. The framework will try to generate unique names for the generated tests by stringifying the arguments passed to the test's constructor:

```
[ReFrame Setup]
  version:           3.0-dev6 (rev: 89d50861)
  command:           './bin/reframe -C tutorial/config/settings.py -c tutorial/
↪advanced/advanced_example8.py -l'
  launched by:       karakasv@daint101
  working directory: '/path/to/reframe'
  check search path: (R) '/path/to/reframe/tutorial/advanced/advanced_example8.py'
  stage directory:   '/path/to/reframe/stage'
  output directory:  '/path/to/reframe/output'

[List of matched checks]
  - MatrixVectorTest_MPI (found in /path/to/reframe/tutorial/advanced/advanced_
↪example8.py)
  - MatrixVectorTest_OpenMP (found in /path/to/reframe/tutorial/advanced/advanced_
↪example8.py)

Found 2 check(s).
```

There are a couple of different ways that we could have used the `@parameterized_test` decorator. One is to use dictionaries for specifying the instantiations of our test class. The dictionaries will be converted to keyword arguments and passed to the constructor of the test class:

```
@rfm.parameterized_test({'variant': 'MPI'}, {'variant': 'OpenMP'})
```

Another way, which is quite useful if you want to generate lots of different tests at the same time, is to use either list comprehensions or generator expressions for specifying the different test instantiations:

```
@rfm.parameterized_test(*([variant] for variant in ['MPI', 'OpenMP']))
```

Tip: Combining parameterized tests and test class hierarchies can offer you a very flexible way for generating multiple related tests at once keeping at the same time the maintenance cost low. We use this technique extensively in our tests.

Flexible Regression Tests

New in version 2.15.

ReFrame can automatically set the number of tasks of a particular test, if its *num_tasks* attribute is set to a negative value or zero. In ReFrame's terminology, such tests are called *flexible*. Negative values indicate the minimum number of tasks that are acceptable for this test (a value of `-4` indicates that at least `4` tasks are required). A zero value indicates the default minimum number of tasks which is equal to *num_tasks_per_node*.

By default, ReFrame will spawn such a test on all the idle nodes of the current system partition, but this behavior can be adjusted with the `--flex-alloc-nodes` command-line option. Flexible tests are very useful for diagnostics tests, e.g., tests for checking the health of a whole set nodes. In this example, we demonstrate this feature through a simple test that runs `hostname`. The test will verify that all the nodes print the expected host name:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class HostnameCheck(rfm.RunOnlyRegressionTest):
    def __init__(self):
        self.valid_systems = ['daint:gpu', 'daint:mc']
        self.valid_prog_environs = ['PrgEnv-cray']
        self.executable = 'hostname'
        self.sourcesdir = None
        self.num_tasks = 0
        self.num_tasks_per_node = 1
        self.sanity_patterns = sn.assert_eq(
            sn.getattr(self, 'num_tasks'),
            sn.count(sn.findall(r'nid\d+', self.stdout))
        )
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}
```

The first thing to notice in this test is that *num_tasks* is set to zero. This is a requirement for flexible tests:

```
self.num_tasks = 0
```

The sanity function of this test simply counts the host names and verifies that they are as many as expected:

```
self.sanity_patterns = sn.assert_eq(
    sn.getattr(self, 'num_tasks'),
    sn.count(sn.findall(r'nid\d+', self.stdout))
)
```

Notice, however, that the sanity check does not use `num_tasks` directly, but rather access the attribute through the *sn.getattr()* sanity function, which is a replacement for the `getattr()` builtin. The reason for that is that at the time the sanity check expression is created, `num_tasks` is `0` and it will only be set to its actual value during the run phase. Consequently, we need to defer the attribute retrieval, thus we use the *sn.getattr()* sanity function instead of accessing it directly

Testing containerized applications

New in version 2.20.

ReFrame can be used also to test applications that run inside a container. A container-based test can be written as *RunOnlyRegressionTest* that sets the *container_platform*. The following example shows a simple test that runs some basic commands inside an Ubuntu 18.04 container and checks that the test has indeed run inside the container and that the stage directory was correctly mounted:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class Example10Test(rfm.RunOnlyRegressionTest):
    def __init__(self):
        self.descr = 'Run commands inside a container'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-cray']
        self.container_platform = 'Singularity'
        self.container_platform.image = 'docker://ubuntu:18.04'
        self.container_platform.commands = [
            'pwd', 'ls', 'cat /etc/os-release'
        ]
        self.container_platform.workdir = '/workdir'
        self.sanity_patterns = sn.all([
            sn.assert_found(r'^' + self.container_platform.workdir,
                            self.stdout),
            sn.assert_found(r'^advanced_example1.c', self.stdout),
            sn.assert_found(r'18.04.\d+ LTS \(Bionic Beaver\)', self.stdout),
        ])
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}
```

A container-based test in ReFrame requires that the *container_platform* is set:

```
        self.container_platform.image = 'docker://ubuntu:18.04'
```

This attribute accepts a string that corresponds to the name of the platform and it instantiates the appropriate *ContainerPlatform* object behind the scenes. In this case, the test will be using Singularity as a container platform. If such a platform is not configured for the current system, the test will fail. For a complete list of supported container platforms, the user is referred to the configuration reference.

As soon as the container platform to be used is defined, you need to specify the container image to use and the commands to run inside the container:

```
        self.container_platform.image = 'docker://ubuntu:18.04'
        self.container_platform.commands = [
            'pwd', 'ls', 'cat /etc/os-release'
        ]
```

These two attributes are mandatory for container-based check. The *image* attribute specifies the name of an image from a registry, whereas the *commands* attribute provides the list of commands to be run inside the container. It is

important to note that the *executable* and *executable_opts* attributes of the actual test are ignored in case of container-based tests.

In the above example, ReFrame will run the container as follows:

```
singularity exec -B"/path/to/test/stagedir:/workdir" docker://ubuntu:18.04 bash -c
↪'cd rfm_workdir; pwd; ls; cat /etc/os-release'
```

By default ReFrame will mount the stage directory of the test under `/rfm_workdir` inside the container and it will always prepend a `cd` command to that directory. The user commands then are then run from that directory one after the other. Once the commands are executed, the container is stopped and ReFrame goes on with the sanity and performance checks.

Users may also change the default mount point of the stage directory by using `workdir` attribute:

```
        self.container_platform.workdir = '/workdir'
```

Besides the stage directory, additional mount points can be specified through the `mount_points` attribute:

```
self.container_platform.mount_points = [('/path/to/host/dir1', '/path/to/container/
↪mount_point1'),
                                        ('/path/to/host/dir2', '/path/to/container/
↪mount_point2')]
```

For a complete list of the available attributes of a specific container platform, the reader is referred to ReFrame Programming APIs guide.

2.3.3 Tutorial 3: Using Dependencies in ReFrame Tests

New in version 2.21.

A ReFrame test may define dependencies to other tests. An example scenario is to test different runtime configurations of a benchmark that you need to compile, or run a scaling analysis of a code. In such cases, you don't want to rebuild your test for each runtime configuration. You could have a build test, which all runtime tests would depend on. This is the approach we take with the following example, that fetches, builds and runs several OSU benchmarks. We first create a basic compile-only test, that fetches the benchmarks and builds them for the different programming environments:

```
@rfm.simple_test
class OSUBuildTest(rfm.CompileOnlyRegressionTest):
    def __init__(self):
        self.descr = 'OSU benchmarks build test'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-gnu', 'PrgEnv-pgi', 'PrgEnv-intel']
        self.sourcesdir = None
        self.prebuild_cmds = [
            'wget http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-
↪benchmarks-5.6.2.tar.gz',
            'tar xzf osu-micro-benchmarks-5.6.2.tar.gz',
            'cd osu-micro-benchmarks-5.6.2'
        ]
        self.build_system = 'Autotools'
        self.build_system.max_concurrency = 8
        self.sanity_patterns = sn.assert_not_found('error', self.stderr)
```

There is nothing particular to that test, except perhaps that you can set *sourcesdir* to `None` even for a test that needs to compile something. In such a case, you should at least provide the commands that fetch the code inside the *prebuild_cmds* attribute.

For the next test we need to use the OSU benchmark binaries that we just built, so as to run the MPI ping-pong benchmark. Here is the relevant part:

```
class OSUBenchmarkTestBase(rfm.RunOnlyRegressionTest):
    '''Base class of OSU benchmarks runtime tests'''

    def __init__(self):
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-gnu', 'PrgEnv-pgi', 'PrgEnv-intel']
        self.sourcesdir = None
        self.num_tasks = 2
        self.num_tasks_per_node = 1
        self.sanity_patterns = sn.assert_found(r'^8', self.stdout)


@rfm.simple_test
class OSULatencyTest(OSUBenchmarkTestBase):
    def __init__(self):
        super().__init__()
        self.descr = 'OSU latency test'
        self.perf_patterns = {
            'latency': sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
        }
        self.depends_on('OSUBuildTest')
        self.reference = {
            '*': {'latency': (0, None, None, 'us')}
        }

    @rfm.require_deps
    def set_executable(self, OSUBuildTest):
        self.executable = os.path.join(
            OSUBuildTest().stagedir,
            'osu-micro-benchmarks-5.6.2', 'mpi', 'pt2pt', 'osu_latency'
        )
        self.executable_opts = ['-x', '100', '-i', '1000']
```

First, since we will have multiple similar benchmarks, we move all the common functionality to the `OSUBenchmarkTestBase` base class. Again nothing new here; we are going to use two nodes for the benchmark and we set *`sourcesdir`* to `None`, since none of the benchmark tests will use any additional resources. The new part comes in with the `OSULatencyTest` test in the following line:

```
        self.depends_on('OSUBuildTest')
```

Here we tell ReFrame that this test depends on a test named `OSUBuildTest`. This test may or may not be defined in the same test file; all ReFrame needs is the test name. By default, the *`depends_on()`* function will create dependencies between the individual test cases of the `OSULatencyTest` and the `OSUBuildTest`, such that the `OSULatencyTest` using `PrgEnv-gnu` will depend on the outcome of the `OSUBuildTest` using `PrgEnv-gnu`, but not on the outcome of the `OSUBuildTest` using `PrgEnv-intel`. This behaviour can be changed, but it is covered in detail in *How Test Dependencies Work In ReFrame*. You can create arbitrary test dependency graphs, but they need to be acyclic. If ReFrame detects cyclic dependencies, it will refuse to execute the set of tests and will issue an error pointing out the cycle.

A ReFrame test with dependencies will execute, i.e., enter its *setup* stage, only after *all* of its dependencies have succeeded. If any of its dependencies fails, the current test will be marked as failure as well.

The next step for the `OSULatencyTest` is to set its executable to point to the binary produced by the `OSUBuildTest`. This is achieved with the following specially decorated function:

```
    @rfm.require_deps
    def set_executable(self, OSUBuildTest):
        self.executable = os.path.join(
            OSUBuildTest().stagedir,
            'osu-micro-benchmarks-5.6.2', 'mpi', 'pt2pt', 'osu_latency'
        )
        self.executable_opts = ['-x', '100', '-i', '1000']
```

The *@require_deps* decorator will bind the arguments passed to the decorated function to the result of the dependency that each argument names. In this case, it binds the `OSUBuildTest` function argument to the result of a dependency named `OSUBuildTest`. In order for the binding to work correctly the function arguments must be named after the target dependencies. However, referring to a dependency only by the test's name is not enough, since a test might be associated with multiple programming environments. For this reason, a dependency argument is actually bound to a function that accepts as argument the name of a target programming environment. If no arguments are passed to that function, as in this example, the current programming environment is implied, such that `OSUBuildTest()` is equivalent to `OSUBuildTest(self.current_environ.name)`. This call returns the actual test case of the dependency that has been executed. This allows you to access any attribute from the target test, as we do in this example by accessing the target test's stage directory, which we use to construct the path of the executable. This concludes the presentation of the `OSULatencyTest` test. The `OSUBandwidthTest` is completely analogous.

The `OSUAllreduceTest` shown below is similar to the other two, except that it is parameterized. It is essentially a scalability test that is running the `osu_allreduce` executable created by the `OSUBuildTest` for 2, 4, 8 and 16 nodes.

```
@rfm.parameterized_test(*([1 << i] for i in range(1, 5)))
class OSUAllreduceTest(OSUBenchmarkTestBase):
    def __init__(self, num_tasks):
        super().__init__()
        self.descr = 'OSU Allreduce test'
        self.perf_patterns = {
            'latency': sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
        }
        self.depends_on('OSUBuildTest')
        self.reference = {
            '*': {'latency': (0, None, None, 'us')}
        }
        self.num_tasks = num_tasks

    @rfm.require_deps
    def set_executable(self, OSUBuildTest):
        self.executable = os.path.join(
            OSUBuildTest().stagedir,
            'osu-micro-benchmarks-5.6.2', 'mpi', 'collective', 'osu_allreduce'
        )
        self.executable_opts = ['-m', '8', '-x', '1000', '-i', '20000']
```

The full set of OSU example tests is shown below:

```
# Copyright 2016-2020 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

import os
```

(continues on next page)

(continued from previous page)

```
import reframe as rfm
import reframe.utility.sanity as sn


class OSUBenchmarkTestBase(rfm.RunOnlyRegressionTest):
    '''Base class of OSU benchmarks runtime tests'''

    def __init__(self):
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-gnu', 'PrgEnv-pgi', 'PrgEnv-intel']
        self.sourcesdir = None
        self.num_tasks = 2
        self.num_tasks_per_node = 1
        self.sanity_patterns = sn.assert_found(r'^8', self.stdout)


@rfm.simple_test
class OSULatencyTest(OSUBenchmarkTestBase):
    def __init__(self):
        super().__init__()
        self.descr = 'OSU latency test'
        self.perf_patterns = {
            'latency': sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
        }
        self.depends_on('OSUBuildTest')
        self.reference = {
            '*': {'latency': (0, None, None, 'us')}
        }

    @rfm.require_deps
    def set_executable(self, OSUBuildTest):
        self.executable = os.path.join(
            OSUBuildTest().stagedir,
            'osu-micro-benchmarks-5.6.2', 'mpi', 'pt2pt', 'osu_latency'
        )
        self.executable_opts = ['-x', '100', '-i', '1000']


@rfm.simple_test
class OSUBandwidthTest(OSUBenchmarkTestBase):
    def __init__(self):
        super().__init__()
        self.descr = 'OSU bandwidth test'
        self.perf_patterns = {
            'bandwidth': sn.extractsingle(r'^4194304\s+(\S+)',
                                          self.stdout, 1, float)
        }
        self.depends_on('OSUBuildTest')
        self.reference = {
            '*': {'bandwidth': (0, None, None, 'MB/s')}
        }

    @rfm.require_deps
    def set_executable(self, OSUBuildTest):
        self.executable = os.path.join(
            OSUBuildTest().stagedir,
            'osu-micro-benchmarks-5.6.2', 'mpi', 'pt2pt', 'osu_bw'
```

(continues on next page)

(continued from previous page)

```
        )
        self.executable_opts = ['-x', '100', '-i', '1000']


@rfm.parameterized_test(*([1 << i] for i in range(1, 5)))
class OSUAllreduceTest(OSUBenchmarkTestBase):
    def __init__(self, num_tasks):
        super().__init__()
        self.descr = 'OSU Allreduce test'
        self.perf_patterns = {
            'latency': sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
        }
        self.depends_on('OSUBuildTest')
        self.reference = {
            '*': {'latency': (0, None, None, 'us')}
        }
        self.num_tasks = num_tasks

    @rfm.require_deps
    def set_executable(self, OSUBuildTest):
        self.executable = os.path.join(
            OSUBuildTest().stagedir,
            'osu-micro-benchmarks-5.6.2', 'mpi', 'collective', 'osu_allreduce'
        )
        self.executable_opts = ['-m', '8', '-x', '1000', '-i', '20000']


@rfm.simple_test
class OSUBuildTest(rfm.CompileOnlyRegressionTest):
    def __init__(self):
        self.descr = 'OSU benchmarks build test'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-gnu', 'PrgEnv-pgi', 'PrgEnv-intel']
        self.sourcesdir = None
        self.prebuild_cmds = [
            'wget http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-
↪benchmarks-5.6.2.tar.gz',
            'tar xzf osu-micro-benchmarks-5.6.2.tar.gz',
            'cd osu-micro-benchmarks-5.6.2'
        ]
        self.build_system = 'Autotools'
        self.build_system.max_concurrency = 8
        self.sanity_patterns = sn.assert_not_found('error', self.stderr)
```

Notice that the order in which dependencies are defined in a test file is irrelevant. In this case, we define `OSUBuildTest` at the end. ReFrame will make sure to properly sort the tests and execute them.

Here is the output when running the OSU tests with the asynchronous execution policy:

```
[==========] Running 7 check(s)
[==========] Started on Wed Jun  3 09:00:40 2020

[----------] started processing OSUBuildTest (OSU benchmarks build test)
[ RUN      ] OSUBuildTest on daint:gpu using PrgEnv-gnu
[ RUN      ] OSUBuildTest on daint:gpu using PrgEnv-intel
[ RUN      ] OSUBuildTest on daint:gpu using PrgEnv-pgi
[----------] finished processing OSUBuildTest (OSU benchmarks build test)
```

(continues on next page)

(continued from previous page)

```
[----------] started processing OSULatencyTest (OSU latency test)
[ RUN      ] OSULatencyTest on daint:gpu using PrgEnv-gnu
[      DEP ] OSULatencyTest on daint:gpu using PrgEnv-gnu
[ RUN      ] OSULatencyTest on daint:gpu using PrgEnv-intel
[      DEP ] OSULatencyTest on daint:gpu using PrgEnv-intel
[ RUN      ] OSULatencyTest on daint:gpu using PrgEnv-pgi
[      DEP ] OSULatencyTest on daint:gpu using PrgEnv-pgi
[----------] finished processing OSULatencyTest (OSU latency test)

[----------] started processing OSUBandwidthTest (OSU bandwidth test)
[ RUN      ] OSUBandwidthTest on daint:gpu using PrgEnv-gnu
[      DEP ] OSUBandwidthTest on daint:gpu using PrgEnv-gnu
[ RUN      ] OSUBandwidthTest on daint:gpu using PrgEnv-intel
[      DEP ] OSUBandwidthTest on daint:gpu using PrgEnv-intel
[ RUN      ] OSUBandwidthTest on daint:gpu using PrgEnv-pgi
[      DEP ] OSUBandwidthTest on daint:gpu using PrgEnv-pgi
[----------] finished processing OSUBandwidthTest (OSU bandwidth test)

[----------] started processing OSUAllreduceTest_2 (OSU Allreduce test)
[ RUN      ] OSUAllreduceTest_2 on daint:gpu using PrgEnv-gnu
[      DEP ] OSUAllreduceTest_2 on daint:gpu using PrgEnv-gnu
[ RUN      ] OSUAllreduceTest_2 on daint:gpu using PrgEnv-intel
[      DEP ] OSUAllreduceTest_2 on daint:gpu using PrgEnv-intel
[ RUN      ] OSUAllreduceTest_2 on daint:gpu using PrgEnv-pgi
[      DEP ] OSUAllreduceTest_2 on daint:gpu using PrgEnv-pgi
[----------] finished processing OSUAllreduceTest_2 (OSU Allreduce test)

[----------] started processing OSUAllreduceTest_4 (OSU Allreduce test)
[ RUN      ] OSUAllreduceTest_4 on daint:gpu using PrgEnv-gnu
[      DEP ] OSUAllreduceTest_4 on daint:gpu using PrgEnv-gnu
[ RUN      ] OSUAllreduceTest_4 on daint:gpu using PrgEnv-intel
[      DEP ] OSUAllreduceTest_4 on daint:gpu using PrgEnv-intel
[ RUN      ] OSUAllreduceTest_4 on daint:gpu using PrgEnv-pgi
[      DEP ] OSUAllreduceTest_4 on daint:gpu using PrgEnv-pgi
[----------] finished processing OSUAllreduceTest_4 (OSU Allreduce test)

[----------] started processing OSUAllreduceTest_8 (OSU Allreduce test)
[ RUN      ] OSUAllreduceTest_8 on daint:gpu using PrgEnv-gnu
[      DEP ] OSUAllreduceTest_8 on daint:gpu using PrgEnv-gnu
[ RUN      ] OSUAllreduceTest_8 on daint:gpu using PrgEnv-intel
[      DEP ] OSUAllreduceTest_8 on daint:gpu using PrgEnv-intel
[ RUN      ] OSUAllreduceTest_8 on daint:gpu using PrgEnv-pgi
[      DEP ] OSUAllreduceTest_8 on daint:gpu using PrgEnv-pgi
[----------] finished processing OSUAllreduceTest_8 (OSU Allreduce test)

[----------] started processing OSUAllreduceTest_16 (OSU Allreduce test)
[ RUN      ] OSUAllreduceTest_16 on daint:gpu using PrgEnv-gnu
[      DEP ] OSUAllreduceTest_16 on daint:gpu using PrgEnv-gnu
[ RUN      ] OSUAllreduceTest_16 on daint:gpu using PrgEnv-intel
[      DEP ] OSUAllreduceTest_16 on daint:gpu using PrgEnv-intel
[ RUN      ] OSUAllreduceTest_16 on daint:gpu using PrgEnv-pgi
[      DEP ] OSUAllreduceTest_16 on daint:gpu using PrgEnv-pgi
[----------] finished processing OSUAllreduceTest_16 (OSU Allreduce test)

[----------] waiting for spawned checks to finish
[       OK ] ( 1/21) OSUBuildTest on daint:gpu using PrgEnv-pgi [compile: 29.581s␣
↪run: 0.086s total: 29.708s]
```

(continues on next page)

(continued from previous page)

```
[       OK ] ( 2/21) OSUBuildTest on daint:gpu using PrgEnv-gnu [compile: 26.250s␣
↪run: 69.120s total: 95.437s]
[       OK ] ( 3/21) OSUBuildTest on daint:gpu using PrgEnv-intel [compile: 39.385s␣
↪run: 89.213s total: 129.871s]
[       OK ] ( 4/21) OSULatencyTest on daint:gpu using PrgEnv-pgi [compile: 0.012s␣
↪run: 145.355s total: 154.504s]
[       OK ] ( 5/21) OSUAllreduceTest_2 on daint:gpu using PrgEnv-pgi [compile: 0.
↪014s run: 148.276s total: 154.433s]
[       OK ] ( 6/21) OSUAllreduceTest_4 on daint:gpu using PrgEnv-pgi [compile: 0.
↪011s run: 149.763s total: 154.407s]
[       OK ] ( 7/21) OSUAllreduceTest_8 on daint:gpu using PrgEnv-pgi [compile: 0.
↪013s run: 151.262s total: 154.378s]
[       OK ] ( 8/21) OSUAllreduceTest_16 on daint:gpu using PrgEnv-pgi [compile: 0.
↪010s run: 152.716s total: 154.360s]
[       OK ] ( 9/21) OSULatencyTest on daint:gpu using PrgEnv-gnu [compile: 0.014s␣
↪run: 210.952s total: 220.847s]
[       OK ] (10/21) OSUBandwidthTest on daint:gpu using PrgEnv-pgi [compile: 0.015s␣
↪run: 213.285s total: 220.758s]
[       OK ] (11/21) OSUAllreduceTest_4 on daint:gpu using PrgEnv-gnu [compile: 0.
↪011s run: 215.596s total: 220.717s]
[       OK ] (12/21) OSUAllreduceTest_16 on daint:gpu using PrgEnv-gnu [compile: 0.
↪011s run: 218.742s total: 220.651s]
[       OK ] (13/21) OSUAllreduceTest_2 on daint:gpu using PrgEnv-intel [compile: 0.
↪013s run: 203.214s total: 206.115s]
[       OK ] (14/21) OSUAllreduceTest_8 on daint:gpu using PrgEnv-intel [compile: 0.
↪016s run: 204.819s total: 206.078s]
[       OK ] (15/21) OSUBandwidthTest on daint:gpu using PrgEnv-gnu [compile: 0.012s␣
↪run: 258.772s total: 266.873s]
[       OK ] (16/21) OSUAllreduceTest_8 on daint:gpu using PrgEnv-gnu [compile: 0.
↪014s run: 263.576s total: 266.752s]
[       OK ] (17/21) OSULatencyTest on daint:gpu using PrgEnv-intel [compile: 0.011s␣
↪run: 227.234s total: 231.789s]
[       OK ] (18/21) OSUAllreduceTest_4 on daint:gpu using PrgEnv-intel [compile: 0.
↪013s run: 229.729s total: 231.724s]
[       OK ] (19/21) OSUAllreduceTest_2 on daint:gpu using PrgEnv-gnu [compile: 0.
↪013s run: 286.203s total: 292.444s]
[       OK ] (20/21) OSUAllreduceTest_16 on daint:gpu using PrgEnv-intel [compile: 0.
↪028s run: 242.030s total: 242.091s]
[       OK ] (21/21) OSUBandwidthTest on daint:gpu using PrgEnv-intel [compile: 0.
↪013s run: 243.719s total: 247.384s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 21 test case(s) from 7 check(s) (0 failure(s))
[==========] Finished on Wed Jun  3 09:07:24 2020
```

Before starting running the tests, ReFrame topologically sorts them based on their dependencies and schedules them for running using the selected execution policy. With the serial execution policy, ReFrame simply executes the tests to completion as they "arrive", since the tests are already topologically sorted. In the asynchronous execution policy, tests are spawned and not waited for. If a test's dependencies have not yet completed, it will not start its execution and a `DEP` message will be printed to denote this.

Finally, ReFrame's runtime takes care of properly cleaning up the resources of the tests respecting dependencies. Normally when an individual test finishes successfully, its stage directory is cleaned up. However, if other tests are depending on this one, this would be catastrophic, since most probably the dependent tests would need the outcome of this test. ReFrame fixes that by not cleaning up the stage directory of a test until all its dependent tests have finished successfully.

2.4 Advanced Topics

2.4.1 How ReFrame Executes Tests

A ReFrame test will be normally tried for different programming environments and different partitions within the same ReFrame run. These are defined in the test's `__init__()` method, but it is not this original test object that is scheduled for execution. The following figure explains in more detail the process:

Fig. 1: How ReFrame loads and schedules tests for execution.

When ReFrame loads a test from the disk it unconditionally constructs it executing its `__init__()` method. The practical implication of this is that your test will be instantiated even if it will not run on the current system. After all the tests are loaded, they are filtered based on the current system and any other criteria (such as programming environment, test attributes etc.) specified by the user (see Test Filtering for more details). After the tests are filtered, ReFrame creates the actual *test cases* to be run. A test case is essentially a tuple consisting of the test, the system partition and the programming environment to try. The test that goes into a test case is essentially a *clone* of the original test that was instantiated upon loading. This ensures that the test case's state is not shared and may not be reused in any case. Finally, the generated test cases are passed to a *runner* that is responsible for scheduling them for execution based on the selected execution policy.

The Regression Test Pipeline

Each ReFrame test case goes through a pipeline with clearly defined stages. ReFrame tests can customize their operation as they execute by attaching hooks to the pipeline stages. The following figure shows the different pipeline stages.

Fig. 2: The regression test pipeline.

All tests will go through every stage one after the other. However, some types of tests implement some stages as no-ops, whereas the sanity or performance check phases may be skipped on demand (see `--skip-sanity-check` and `--skip-performance-check` options). In the following we describe in more detail what happens in every stage.

The Setup Phase

During this phase the test will be set up for the currently selected system partition and programming environment. The `current_partition` and `current_environ` test attributes will be set and the paths associated to this test case (stage, output and performance log directories) will be created. A job descriptor will also be created for the test case containing information about the job to be submitted later in the pipeline.

The Build Phase

During this phase the source code associated with the test is compiled using the current programming environment. If the test is "run-only," this phase is a no-op.

Before building the test, all the resources associated with it are copied to the test case's stage directory. ReFrame then temporarily switches to that directory and builds the test.

The Run Phase

During this phase a job script associated with the test case will be created and it will be submitted for execution. If the test is "run-only," its resources will be first copied to the test case's stage directory. ReFrame will temporarily switch to that directory and spawn the test's job from there. This phase is executed asynchronously (either a batch job is spawned or a local process is started) and it is up to the selected *execution policy* to block or not until the associated job finishes.

The Sanity Phase

During this phase, the sanity of the test's output is checked. ReFrame makes no assumption as of what a successful test is; it does not even look into its exit code. This is entirely up to the test to define. ReFrame provides a flexible and expressive way for specifying complex patterns and operations to be performed on the test's output in order to determine the outcome of the test.

The Performance Phase

During this phase, the performance metrics reported by the test (if it is performance test) are collected, logged and compared to their reference values. The mechanism for extracting performance metrics from the test's output is the same used by the sanity checking phase for extracting patterns from the test's output.

The Cleanup Phase

During this final stage of the pipeline, the test's resources are cleaned up. More specifically, if the test has finished successfully, all interesting test files (build/job scripts, build/job script output and any user-specified files) are copied to ReFrame's output directory and the stage directory of the test is deleted.

Note: This phase might be deferred in case a test has dependents (see *Cleaning up stage files* for more details).

Execution Policies

All regression tests in ReFrame will execute the pipeline stages described above. However, how exactly this pipeline will be executed is responsibility of the test execution policy. There are two execution policies in ReFrame: the serial and the asynchronous one.

In the serial execution policy, a new test gets into the pipeline after the previous one has exited. As the figure below shows, this can lead to long idling times in the run phase, since the execution blocks until the associated test job finishes.

In the asynchronous execution policy, multiple tests can be simultaneously on-the-fly. When a test enters the run phase, ReFrame does not block, but continues by picking the next test case to run. This continues until no more test

Fig. 3: The serial execution policy.

cases are left for execution or until a maximum concurrency limit is reached. At the end, ReFrame enters a busy-wait loop monitoring the spawned test cases. As soon as test case finishes, it resumes its pipeline and runs it to completion. The following figure shows how the asynchronous execution policy works.

Fig. 4: The asynchronous execution policy.

ReFrame tries to keep concurrency high by maintaining as many test cases as possible simultaneously active. When the concurrency limit is reached, ReFrame will first try to free up execution slots by checking if any of the spawned jobs have finished, and it will fill that slots first before throttling execution.

ReFrame uses polling to check the status of the spawned jobs, but it does so in a dynamic way, in order to ensure both responsiveness and avoid overloading the system job scheduler with excessive polling.

Timing the Test Pipeline

New in version 3.0.

ReFrame keeps track of the time a test spends in every pipeline stage and reports that after each test finishes. However, it does so from its own perspective and not from that of the scheduler backend used. This has some practical implications: As soon as a test enters the "run" phase, ReFrame's timer for that phase starts ticking regardless if the associated job is pending. Similarly, the "run" phase ends as soon as ReFrame realizes it. This will happen after the associated job has finished. For this reason, the time spent in the pipeline's "run" phase should *not* be interpreted as the actual runtime of the test, especially if a non-local scheduler backend is used.

Finally, the execution time of the "cleanup" phase is not reported when a test finishes, since it may be deferred in case that there exist tests that depend on that one. See *How Test Dependencies Work In ReFrame* for more information on how ReFrame treats tests with dependencies.

2.4.2 How Test Dependencies Work In ReFrame

Dependencies in ReFrame are defined at the test level using the `depends_on()` function, but are projected to the test cases space. We will see the rules of that projection in a while. The dependency graph construction and the subsequent dependency analysis happen also at the level of the test cases.

Let's assume that test `T1` depends in `T0`. This can be expressed inside `T1` using the `depends_on()` method:

```
@rfm.simple_test
class T1(rfm.RegressionTest):
    def __init__(self):
        ...
        self.depends_on('T0')
```

Conceptually, this dependency can be viewed at the test level as follows:

For most of the cases, this is sufficient to reason about test dependencies. In reality, as mentioned above, dependencies are handled at the level of test cases. Test cases on different partitions are always independent. If not specified differently, test cases using programming environments are also independent. This is the default behavior of the `depends_on()` function. The following image shows the actual test case dependencies assuming that both tests support the `E0` and `E1` programming environments (for simplicity, we have omitted the partitions, since tests are always independent in that dimension):

This means that test cases of `T1` may start executing before all test cases of `T0` have finished. You can impose a stricter dependency between tests, such that `T1` does not start execution unless all test cases of `T0` have finished. You can achieve this as follows:

```
@rfm.simple_test
class T1(rfm.RegressionTest):
    def __init__(self):
        ...
        self.depends_on('T0', how=rfm.DEPEND_FULLY)
```

This will create the following test case graph:

You may also create arbitrary dependencies between the test cases of different tests, like in the following example, where the dependencies cannot be represented in any of the other two ways:

These dependencies can be achieved as follows:

```
@rfm.simple_test
class T1(rfm.RegressionTest):
    def __init__(self):
        ...
        self.depends_on('T0', how=rfm.DEPEND_EXACT,
                        subdeps={'E0': ['E0', 'E1'], 'E1': ['E1']})
```

The `subdeps` argument defines the sub-dependencies between the test cases of `T1` and `T0` using an adjacency list representation.

Cyclic dependencies

Obviously, cyclic dependencies between test cases are not allowed. Cyclic dependencies between tests are not allowed either, even if the test case dependency graph is acyclic. For example, the following dependency set up is invalid:

The test case dependencies here, clearly, do not form a cycle, but the edge from `(T0, E0)` to `(T1, E1)` introduces a dependency from `T0` to `T1` forming a cycle at the test level. The reason we impose this restriction is that we wanted to keep the original processing of tests by ReFrame, where all the test cases of a test are processed before moving to the next one. Supporting this type of dependencies would require to change substantially ReFrame's output.

Dangling dependencies

In our discussion so far, `T0` and `T1` had the same valid programming environments. What happens if they do not? Assume, for example, that `T0` and `T1` are defined as follows:

```
import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class T0(rfm.RegressionTest):
    def __init__(self):
        self.valid_systems = ['P0']
        self.valid_prog_environs = ['E0']
```

(continues on next page)

(continued from previous page)

```
        ...


@rfm.simple_test
class T1(rfm.RegressionTest):
    def __init__(self):
        self.valid_systems = ['P0']
        self.valid_prog_environs = ['E0', 'E1']
        self.depends_on('T0')
        ...
```

As discussed previously, `depends_on()` will create one-to-one dependencies between the different programming environment test cases. So in this case it will try to create an edge from `(T1, E1)` to `(T0, E1)` as shown below:

This edge cannot be resolved since the target test case does not exist. ReFrame will complain and issue an error while trying to build the test dependency graph. The remedy to this is to use either `DEPEND_FULLY` or pass the exact dependencies with `DEPEND_EXACT` to `depends_on()`.

If `T0` and `T1` had their `valid_prog_environs` swapped, such that `T0` supported `E0` and `E1` and `T1` supported only `E0`, the default `depends_on()` mode would work fine. The `(T0, E1)` test case would simply have no dependent test cases.

Resolving dependencies

As shown in the *Tutorial 3: Using Dependencies in ReFrame Tests*, test dependencies would be of limited usage if you were not able to use the results or information of the target tests. Let's reiterate over the `set_executable()` function of the `OSULatencyTest` that we presented previously:

```
        self.depends_on('OSUBuildTest')
        self.reference = {
            '*': {'latency': (0, None, None, 'us')}
        }

    @rfm.require_deps
    def set_executable(self, OSUBuildTest):
```

The `@require_deps` decorator does some magic – we will unravel this shortly – with the function arguments of the `set_executable()` function and binds them to the target test dependencies by their name. However, as discussed in this section, dependencies are defined at test case level, so the `OSUBuildTest` function argument is bound to a special function that allows you to retrieve an actual test case of the target dependency. This is why you need to "call" `OSUBuildTest` in order to retrieve the desired test case. When no arguments are passed, this will retrieve the test case corresponding to the current partition and the current programming environment. We could always retrieve the `PrgEnv-gnu` case by writing `OSUBuildTest('PrgEnv-gnu')`. If a dependency cannot be resolved, because it is invalid, a runtime error will be thrown with an appropriate message.

The low-level method for retrieving a dependency is the *`getdep()`* method of the `RegressionTest`. In fact, you can rewrite `set_executable()` function as follows:

```
@rfm.run_after('setup')
def set_executable(self):
    target = self.getdep('OSUBuildTest')
    self.executable = os.path.join(
```

(continues on next page)

(continued from previous page)

```
        target.stagedir,
        'osu-micro-benchmarks-5.6.2', 'mpi', 'pt2pt', 'osu_latency'
    )
    self.executable_opts = ['-x', '100', '-i', '1000']
```

Now it's easier to understand what the `@require_deps` decorator does behind the scenes. It binds the function arguments to a partial realization of the `getdep()` function and attaches the decorated function as an after-setup hook. In fact, any `@require_deps`-decorated function will be invoked before any other after-setup hook.

Cleaning up stage files

In principle, the output of a test might be needed by its dependent tests. As a result, the stage directory of the test will only be cleaned up after all of its *immediate* dependent tests have finished successfully. If any of its children has failed, the cleanup phase will be skipped, such that all the test's files will remain in the stage directory. This allows users to reproduce manually the error of a failed test with dependencies, since all the needed resources of the failing test are left in their original location.

2.4.3 Understanding the Mechanism of Sanity Functions

This section describes the mechanism behind the sanity functions that are used for the sanity and performance checking. Generally, writing a new sanity function is as straightforward as decorating a simple Python function with the `reframe.utility.sanity.sanity_function()` decorator. However, it is important to understand how and when a deferrable function is evaluated, especially if your function takes as arguments the results of other deferrable functions.

What Is a Deferrable Function?

A deferrable function is a function whose a evaluation is deferred to a later point in time. You can define any function as deferrable by wrapping it with the `reframe.utility.sanity.sanity_function()` decorator before its definition. The example below demonstrates a simple scenario:

```
import reframe.utility.sanity as sn

@sn.sanity_function
def foo():
    print('hello')
```

If you try to call `foo()`, its code will not execute:

```
>>> foo()
<reframe.core.deferrable._DeferredExpression object at 0x2b70fff23550>
```

Instead, a special object is returned that represents the function whose execution is deferred. Notice the more general *deferred expression* name of this object. We shall see later on why this name is used.

In order to explicitly trigger the execution of `foo()`, you have to call *`evaluate`* on it:

```
>>> from reframe.utility.sanity import evaluate
>>> evaluate(foo())
hello
```

If the argument passed to *evaluate* is not a deferred expression, it will be simply returned as is.

Deferrable functions may also be combined as we do with normal functions. Let's extend our example with `foo()` accepting an argument and printing it:

```
import reframe.utility.sanity as sn

@sn.sanity_function
def foo(arg):
    print(arg)

@sn.sanity_function
def greetings():
    return 'hello'
```

If we now do `foo(greetings())`, again nothing will be evaluated:

```
>>> foo(greetings())
<reframe.core.deferrable._DeferredExpression object at 0x2b7100e9e978>
```

If we trigger the evaluation of `foo()` as before, we will get expected result:

```
>>> evaluate(foo(greetings()))
hello
```

Notice how the evaluation mechanism goes down the function call graph and returns the expected result. An alternative way to evaluate this expression would be the following:

```
>>> x = foo(greetings())
>>> x.evaluate()
hello
```

As you may have noticed, you can assign a deferred function to a variable and evaluate it later. You may also do `evaluate(x)`, which is equivalent to `x.evaluate()`.

To demonstrate more clearly how the deferred evaluation of a function works, let's consider the following `size3()` deferrable function that simply checks whether an `iterable` passed as argument has three elements inside it:

```
@sn.sanity_function
def size3(iterable):
    return len(iterable) == 3
```

Now let's assume the following example:

```
>>> l = [1, 2]
>>> x = size3(l)
>>> evaluate(x)
False
>>> l += [3]
>>> evaluate(x)
True
```

We first call `size3()` and store its result in `x`. As expected when we evaluate `x`, `False` is returned, since at the time of the evaluation our list has two elements. We later append an element to our list and reevaluate `x` and we get `True`, since at this point the list has three elements.

Note: Deferred functions and expressions may be stored and (re)evaluated at any later point in the program.

An important thing to point out here is that deferrable functions *capture* their arguments at the point they are called. If you change the binding of a variable name (either explicitly or implicitly by applying an operator to an immutable object), this change will not be reflected when you evaluate the deferred function. The function instead will operate on its captured arguments. We will demonstrate this by replacing the list in the above example with a tuple:

```
>>> l = (1, 2)
>>> x = size3(l)
>>> l += (3,)
>>> l
(1, 2, 3)
>>> evaluate(x)
False
```

Why this is happening? This is because tuples are immutable so when we are doing `l += (3,)` to append to our tuple, Python constructs a new tuple and rebinds `l` to the newly created tuple that has three elements. However, when we called our deferrable function, `l` was pointing to a different tuple object, and that was the actual tuple argument that our deferrable function has captured.

The following augmented example demonstrates this:

```
>>> l = (1, 2)
>>> x = size3(l)
>>> l += (3,)
>>> l
(1, 2, 3)
>>> evaluate(x)
False
>>> l = (1, 2)
>>> id(l)
47764346657160
>>> x = size3(l)
>>> l += (3,)
>>> id(l)
47764330582232
>>> l
(1, 2, 3)
>>> evaluate(x)
False
```

Notice the different IDs of `l` before and after the += operation. This a key trait of deferrable functions and expressions that you should be aware of.

Deferred expressions

You might be still wondering why the internal name of a deferred function refers to the more general term deferred expression. Here is why:

```
>>> @sn.sanity_function
... def size(iterable):
...     return len(iterable)
...
>>> l = [1, 2]
>>> x = 2*(size(l) + 3)
>>> x
<reframe.core.deferrable._DeferredExpression object at 0x2b1288f4e940>
>>> evaluate(x)
10
```

As you can see, you can use the result of a deferred function inside arithmetic operations. The result will be another deferred expression that you can evaluate later. You can practically use any Python builtin operator or builtin function with a deferred expression and the result will be another deferred expression. This is quite a powerful mechanism, since with the standard syntax you can create arbitrary expressions that may be evaluated later in your program.

There are some exceptions to this rule, though. The logical `and`, `or` and `not` operators as well as the `in` operator cannot be deferred automatically. These operators try to take the truthy value of their arguments by calling `bool` on them. As we shall see later, applying the `bool` function on a deferred expression causes its immediate evaluation and returns the result. If you want to defer the execution of such operators, you should use the corresponding *`and_`*, *`or_`*, *`not_`* and *`contains`* functions in *`reframe.utility.sanity`*, which basically wrap the expression in a deferrable function.

In summary deferrable functions have the following characteristics:

- You can make any function deferrable by wrapping it with the `reframe.utility.sanity.sanity_function()` decorator.
- When you call a deferrable function, its body is not executed but its arguments are *captured* and an object representing the deferred function is returned.
- You can execute the body of a deferrable function at any later point by calling *`evaluate`* on the deferred expression object that it has been returned by the call to the deferred function.
- Deferred functions can accept other deferred expressions as arguments and may also return a deferred expression.
- When you evaluate a deferrable function, any other deferrable function down the call tree will also be evaluated.
- You can include a call to a deferrable function in any Python expression and the result will be another deferred expression.

How a Deferred Expression Is Evaluated?

As discussed before, you can create a new deferred expression by calling a function whose definition is decorated by the `@sanity_function` or `@deferrable` decorator or by including an already deferred expression in any sort of arithmetic operation. When you call *`evaluate`* on a deferred expression, you trigger the evaluation of the whole subexpression tree. Here is how the evaluation process evolves:

A deferred expression object is merely a placeholder of the target function and its arguments at the moment you call it. Deferred expressions leverage also the Python's data model so as to capture all the binary and unary operators supported by the language. When you call `evaluate()` on a deferred expression object, the stored function will be called passing it the captured arguments. If any of the arguments is a deferred expression, it will be evaluated too. If the return value of the deferred expression is also a deferred expression, it will be evaluated as well.

This last property lets you call other deferrable functions from inside a deferrable function. Here is an example where we define two deferrable variations of the builtins `sum` and `len` and another deferrable function `avg()` that computes the average value of the elements of an iterable by calling our deferred builtin alternatives.

```
@sn.sanity_function
def dsum(iterable):
    return sum(iterable)

@sn.sanity_function
def dlen(iterable):
    return len(iterable)

@sn.sanity_function
def avg(iterable):
    return dsum(iterable) / dlen(iterable)
```

If you try to evaluate `avg()` with a list, you will get the expected result:

```
>>> avg([1, 2, 3, 4])
<reframe.core.deferrable._DeferredExpression object at 0x2b1288f54b70>
>>> evaluate(avg([1, 2, 3, 4]))
2.5
```

The return value of `evaluate(avg())` would normally be a deferred expression representing the division of the results of the other two deferrable functions. However, the evaluation mechanism detects that the return value is a deferred expression and it automatically triggers its evaluation, yielding the expected result. The following figure shows how the evaluation evolves for this particular example:

Fig. 5: Sequence diagram of the evaluation of the deferrable `avg()` function.

Implicit evaluation of a deferred expression

Although you can trigger the evaluation of a deferred expression at any time by calling `evaluate`, there are some cases where the evaluation is triggered implicitly:

- When you try to get the truthy value of a deferred expression by calling `bool` on it. This happens for example when you include a deferred expression in an `if` statement or as an argument to the `and`, `or`, `not` and `in` (`__contains__`) operators. The following example demonstrates this behavior:

  ```
  >>> if avg([1, 2, 3, 4]) > 2:
  ...     print('hello')
  ...
  hello
  ```

  The expression `avg([1, 2, 3, 4]) > 2` is a deferred expression, but its evaluation is triggered from the Python interpreter by calling the `bool()` method on it, in order to evaluate the `if` statement. A similar example is the following that demonstrates the behaviour of the `in` operator:

  ```
  >>> from reframe.utility.sanity import defer
  >>> l = defer([1, 2, 3])
  >>> l
  <reframe.core.deferrable._DeferredExpression object at 0x2b1288f54cf8>
  >>> evaluate(l)
  [1, 2, 3]
  >>> 4 in l
  False
  >>> 3 in l
  True
  ```

  The *`defer`* is simply a deferrable version of the identity function (a function that simply returns its argument). As expected, `l` is a deferred expression that evaluates to the `[1, 2, 3]` list. When we apply the `in` operator, the deferred expression is immediately evaluated.

  Note: Python expands this expression into `bool(l.__contains__(3))`. Although `__contains__` is also defined as a deferrable function in `_DeferredExpression`, its evaluation is triggered by the `bool` builtin.

- When you try to iterate over a deferred expression by calling the `iter` function on it. This call happens implicitly by the Python interpreter when you try to iterate over a container. Here is an example:

```
>>> @sn.sanity_function
... def getlist(iterable):
...     ret = list(iterable)
...     ret += [1, 2, 3]
...     return ret
>>> getlist([1, 2, 3])
<reframe.core.deferrable._DeferredExpression object at 0x2b1288f54dd8>
>>> for x in getlist([1, 2, 3]):
...     print(x)
...
1
2
3
1
2
3
```

Simply calling `getlist()` will not execute anything and a deferred expression object will be returned. However, when you try to iterate over the result of this call, then the deferred expression will be evaluated immediately.

- When you try to call `str` on a deferred expression. This will be called by the Python interpreter every time you try to print this expression. Here is an example with the `getlist` deferrable function:

```
>>> print(getlist([1, 2, 3]))
[1, 2, 3, 1, 2, 3]
```

How to Write a Deferrable Function?

The answer is simple: like you would with any other normal function! We've done that already in all the examples we've shown in this documentation. A question that somehow naturally comes up here is whether you can call a deferrable function from within a deferrable function, since this doesn't make a lot of sense: after all, your function will be deferred anyway.

The answer is, yes. You can call other deferrable functions from within a deferrable function. Thanks to the implicit evaluation rules as well as the fact that the return value of a deferrable function is also evaluated if it is a deferred expression, you can write a deferrable function without caring much about whether the functions you call are themselves deferrable or not. However, you should be aware of passing mutable objects to deferrable functions. If these objects happen to change between the actual call and the implicit evaluation of the deferrable function, you might run into surprises. In any case, if you want the immediate evaluation of a deferrable function or expression, you can always do that by calling *`evaluate`* on it.

The following example demonstrates two different ways writing a deferrable function that checks the average of the elements of an iterable:

```
import reframe.utility.sanity as sn

@sn.sanity_function
def check_avg_with_deferrables(iterable):
    avg = sn.sum(iterable) / sn.len(iterable)
    return -1 if avg > 2 else 1

@sn.sanity_function
def check_avg_without_deferrables(iterable):
    avg = sum(iterable) / len(iterable)
    return -1 if avg > 2 else 1
```

```
>>> evaluate(check_avg_with_deferrables([1, 2, 3, 4]))
-1
>>> evaluate(check_avg_without_deferrables([1, 2, 3, 4]))
-1
```

The first version uses the *sum* and *len* functions from *reframe.utility.sanity*, which are deferrable versions of the corresponding builtins. The second version uses directly the builtin `sum` and `len` functions. As you can see, both of them behave in exactly the same way. In the version with the deferrables, `avg` is a deferred expression but it is evaluated by the `if` statement before returning.

Generally, inside a sanity function, it is a preferable to use the non-deferrable version of a function, if that exists, since you avoid the extra overhead and bookkeeping of the deferring mechanism.

Deferrable Sanity Functions

Normally, you will not have to implement your own sanity functions, since ReFrame provides already a variety of them. You can find the complete list of provided sanity functions here.

Similarities and Differences with Generators

Python allows you to create functions that will be evaluated lazily. These are called generator functions. Their key characteristic is that instead of using the `return` keyword to return values, they use the `yield` keyword. I'm not going to go into the details of the generators, since there is plenty of documentation out there, so I will focus on the similarities and differences with our deferrable functions.

Similarities

- Both generators and our deferrables return an object representing the deferred expression when you call them.
- Both generators and deferrables may be evaluated explicitly or implicitly when they appear in certain expressions.
- When you try to iterate over a generator or a deferrable, you trigger its evaluation.

Differences

- You can include deferrables in any arithmetic expression and the result will be another deferrable expression. This is not true with generator functions, which will raise a `TypeError` in such cases or they will always evaluate to `False` if you include them in boolean expressions Here is an example demonstrating this:

```
>>> @sn.sanity_function
... def dsize(iterable):
...     print(len(iterable))
...     return len(iterable)
...
>>> def gsize(iterable):
...     print(len(iterable))
...     yield len(iterable)
...
>>> l = [1, 2]
>>> dsize(l)
<reframe.core.deferrable._DeferredExpression object at 0x2abc630abb38>
```

(continues on next page)

(continued from previous page)

```
>>> gsize(l)
<generator object gsize at 0x2abc62a4bf10>
>>> expr = gsize(l) == 2
>>> expr
False
>>> expr = gsize(l) + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'generator' and 'int'
>>> expr = dsize(l) == 2
>>> expr
<reframe.core.deferrable._DeferredExpression object at 0x2abc630abba8>
>>> expr = dsize(l) + 2
>>> expr
<reframe.core.deferrable._DeferredExpression object at 0x2abc630abc18>
```

Notice that you cannot include generators in expressions, whereas you can generate arbitrary expressions with deferrables.

- Generators are iterator objects, while deferred expressions are not. As a result, you can trigger the evaluation of a generator expression using the `next` builtin function. For a deferred expression you should use *`evaluate`* instead.
- A generator object is iterable, whereas a deferrable object will be iterable if and only if the result of its evaluation is iterable.

Note: Technically, a deferrable object is iterable, too, since it provides the `__iter__` method. That's why you can include it in iteration expressions. However, it delegates this call to the result of its evaluation.

Here is an example demonstrating this difference:

```
>>> for i in gsize(l): print(i)
...
2
2
>>> for i in dsize(l): print(i)
...
2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/users/karakasv/Devel/reframe/reframe/core/deferrable.py", line 73, in __
↪iter__
    return iter(self.evaluate())
TypeError: 'int' object is not iterable
```

Notice how the iteration works fine with the generator object, whereas with the deferrable function, the iteration call is delegated to the result of the evaluation, which is not an iterable, therefore yielding `TypeError`. Notice also, the printout of `2` in the iteration over the deferrable expression, which shows that it has been evaluated.

2.5 Use Cases

ReFrame has been publicly released on May 2017, but it has been used in production at the Swiss National Supercomputing Centre since December 2016. Since then it has gained visibility across computing centers, some of which have already integrated in their production testing workflows and others are considering to fully adopt it. To our knowledge, private companies in the HPC sector are using it as well. Here we will briefly present the use cases of ReFrame at the Swiss National Supercomputing Centre (CSCS) in Switzerland, at the National Energy Research Scientific Computing Center (NERSC) and at the Ohio Supercomputer Center (OSC) in the United States.

2.5.1 ReFrame at CSCS

CSCS uses ReFrame for both functionality and performance tests for all of its production and test development systems, among which are the Piz Daint supercomputer (Cray XC40/XC50 hybrid system), the Kesch/Escha twin systems (Cray CS-Storm used by MeteoSwiss for weather predition). The same ReFrame tests are reused as much as possible across systems with minor adaptations. The test suite of CSCS (publicly available inside ReFrame's repository) comprises tests for full scientific applications, scientific libraries, programming environments, compilation and linking, profiling and debugger tools, basic CUDA operations, performance microbenchmarks and I/O libraries. Using tags we have split the tests in three broad overlapping categories:

1. Production tests – This category comprises a large variety of tests and is run daily overnight using Jenkins.
2. Maintenance tests – This suite is essentially a small subset of the production tests, comprising mostly application sanity and performance tests, as well as sanity tests for the programming environment and the scheduler. It is run before and after maintenance of the systems.
3. Benchmarking tests – These tests are used to measure the performance of different computing and networking components and are run manually before major upgrades or when a performance problem needs to be investigated.

We are currently working on a fourth category of tests that are intended to run frequently (e.g., every 10 minutes). The purpose of these tests is to measure the system behavior and performance as perceived by the users. Example tests are the time it takes to run basic Slurm commands and/or performance basic filesystem operations. Such glitches might affect the performance of running applications and cause users to open support tickets. Collecting periodically such performance data will help us correlate system events with user application performance. Finally, there is an ongoing effort to expand our ReFrame test suite to virtual clusters based on OpenStack. The new tests will measure the responsiveness of our OpenStack installation to deploy compute instances, volumes, and perform snapshots. We plan to make them publicly available in the near future.

Our regression test suite consists of 278 tests in total, from which 204 are marked as production tests. A test can be valid for one or more systems and system partitions and can be tried with multiple programming environments. Specifically on Piz Daint, the production suite runs 640 test cases from 193 tests.

ReFrame really focuses on abstracting away all the gory details from the regression test description, hence letting the user to concentrate solely on the logic of his test. This effect can be seen in the following Table where the total amount of lines of code (loc) of the regression tests written with the previous shell script-based solution is shown in comparison to ReFrame.

Maintenance Burden	Shell-Script Based	ReFrame (May 2017)	ReFrame (Apr. 2020)
Total tests	179	122	278
Total size of tests	14635 loc	2985 loc	8421 loc
Avg. test file size	179 loc	93 loc	102 loc
Avg. effective test size	179 loc	25 loc	30 loc

The difference in the total amount of regression test code is dramatic. From the 15K lines of code of the old shell script based regression testing suite, ReFrame tests used only 3K lines of code (first public release, May 2017) achieving a

higher coverage.

Each regression test file in ReFrame is approximately 100 loc on average. However, each regression test file may contain or generate more than one related tests, thus leading to the effective decrease of the line count per test to only 30 loc. If we also account for the test cases generated per test, this number decreases further.

Separating the logical description of a regression test from all the unnecessary implementation details contributes significantly to the ease of writing and maintaining new regression tests with ReFrame.

Note: The higher test count of the older suite refers to test cases, i.e., running the same test for different programming environments, whereas for ReFrame the counts do not account for this.

Note: CSCS maintains a separate repository for tests related to HPC debugging and performance tools, which you can find here. These tests were not accounted in this analysis.

2.5.2 ReFrame at NERSC

ReFrame at NERSC covers functionality and performance of its current HPC system Cori, a Cray XC40 with Intel "Haswell" and "Knights Landing" compute nodes; as well as its smaller Cray CS-Storm cluster featuring Intel "Skylake" CPUs and NVIDIA "Volta" GPUs. The performance tests include several general-purpose benchmarks designed to stress different components of the system, including HPGMG (both finite-element and finite-volume tests), HPCG, Graph500, IOR, and others. Additionally, the tests include several benchmark codes used during NERSC system procurements, as well as several extracted benchmarks from full applications which participate in the NERSC Exascale Science Application Program (NESAP). Including NESAP applications ensures that representative components of the NERSC workload are included in the performance tests.

The functionality tests evaluate several different components of the system; for example, there are several tests for the Cray DataWarp software which enables users to interact with the Cori burst buffer. There are also several Slurm tests which verify that partitions and QoSs are correctly configured for jobs of varying sizes. The Cray programming environments, including compiler wrappers, MPI and OpenMP capability, and Shifter, are also included in these tests, and are especially impactful following changes in defaults to the programming environments.

The test battery at NERSC can be invoked both manually and automatically, depending on the need. Specifically, the full battery is typically executed manually following a significant change to the Cori system, e.g., after a major system software change, or a Cray Linux OS upgrade, before the system is released back to users. Under most other circumstances, however, only a subset of tests are typically run, and in most causes they are executed automatically. NERSC uses ReFrame's tagging capabilities to categorize the various subsets of tests, such that groups of tests which evaluate a particular component of the system can be invoked easily. For example, some performance tests are tagged as "daily", others as "weekly", "reboot", "slurm", "aries", etc., such that it is clear from the test's Python code when and how frequently a particular test is run.

ReFrame has also been integrated into NERSC's centralized data collection service used for facility and system monitoring, called the "Data Collect." The Data Collect stores data in an Elasticsearch instance, uses Logstash to ingest log information about the Cori system, and provides a web-based GUI to display results via Kibana. Cray, in turn, provides the Cray Lightweight Log Manager on XC systems such as Cori, which provides a syslog interface. ReFrame's support for Syslog, and the Python standard logging library, enabled simple integration with NERSC's Data Collect The result of this integration with ReFrame to the Data Collect is that the results from each ReFrame test executed on Cori are visible via a Kibana query within a few seconds of the test completing. One can then configure Elasticsearch to alert a system administrator if a particular system functionality stops working, or if the performance of certain benchmarks suddenly declines.

Finally, ReFrame has been automated at NERSC via the continuous integration (CI) capabilities provided by an internal GitLab instance. More specifically, GitLab was enhanced due to efforts from the US Department of Energy

Exascale Computing Project (ECP) in order to allow CI "runners" to submit jobs to queues on HPC systems such as Cori automatically via schedulable "pipelines." Automation via GitLab runners is a significant improvement over test executed automated by cron, because the runners exist outside of the Cori system, and therefore are unaffected by system shutdowns, reboots, and other disruptions. The pipelines are configured to run tests with particular tags at particular times, e.g., tests tagged with "daily" are invoked each day at the same time, tests tagged "weekly" are invoked once per week, etc.

2.5.3 ReFrame at OSC

At OSC, we use ReFrame to build the testing system for the software environment. As a change is made to an application, e.g., upgrade, module change or new installation, ReFrame tests are performed by a user-privilege account and the OSC staff members who receive the test summary can easily check the result to decide if the change should be approved.

ReFrame is configured and installed on three production systems (Pitzer, Owens and Ruby). For each application we prepare the following classes of ReFrame tests:

1. default version – checks if a new installation overwrites the default module file
2. broken executable or library – i.e. run a binary with the `--version` flag and compare the result with the module version,
3. functionality – i.e. numerical tests,
4. performance – extensive functionality checking and benchmarking,

where we currently have functionality and performance tests for a limited subset of our deployed software.

All checks are designed to be general and version independent. The correct module file is loaded at runtime, reducing the number of Python classes to be maintained. In addition, all application-based ReFrame tests are performed as regression testing of software environment when the system has critical update or rolling reboot.

ReFrame is also used for performance monitoring. We run weekly MPI tests and monthly HPCG tests. The performance data is logged directly to an internal Splunk server via Syslog protocol. The job summary is sent to the responsible OSC staff member who can watch the performance dashboards.

2.6 Migrating to ReFrame 3

ReFrame 3 brings substantial changes in its configuration. The configuration component was completely revised and rewritten from scratch in order to allow much more flexibility in how the framework's configuration options are handled, as well as to ensure the maintainability of the framework in the future.

At the same time, ReFrame 3 deprecates some common pre-2.20 test syntax in favor of the more modern and intuitive pipeline hooks, as well as renames some regression test attributes.

This guide details the necessary steps in order to easily migrate to ReFrame 3.

2.6.1 Updating Your Site Configuration

As described in Configuring ReFrame for Your Site, ReFrame's configuration file has changed substantially. However, you don't need to manually update your configuration; ReFrame will do that automatically for you. As soon as it detects an old-style configuration file, it will convert it to the new syntax save it in a temporary file:

```
$ ./bin/reframe -C unittests/resources/settings_old_syntax.py -l
./bin/reframe: the syntax of the configuration file 'unittests/resources/settings_old_
↪syntax.py' is deprecated
./bin/reframe: configuration file has been converted to the new syntax here: '/var/
↪folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/tmph5n8u3kf.py'
```

Alternatively, you can convert any old configuration file using the command line option *`--upgrade-config-file`*:

```
$ ./bin/reframe --upgrade-config-file unittests/resources/settings_old_syntax.py:new_
↪config.py
Conversion successful! The converted file can be found at 'new_config.py'.
```

Another important change is that default locations for looking up a configuration file has changed (see Configuring ReFrame for Your Site for more details). That practically means that if you were relying on ReFrame loading your `reframe/settings.py` by default, this is no longer true. You have to move it to any of the default settings locations or set the corresponding command line option or environment variable.

Note: The conversion tool will create a JSON configuration file if the extension of the target file is `.json`.

Automatic conversion limitations

ReFrame does a pretty good job in converting correctly your old configuration files, but there are some limitations:

- Your code formatting will be lost. ReFrame will use its own, which is PEP8 compliant nonetheless.
- Any comments will be lost.
- Any code that was used to dynamically generate configuration parameters will be lost. ReFrame will generate the new configuration based on what was the actual old configuration after any dynamic generation.

Warning: The very old logging configuration syntax (prior to ReFrame 2.13) is no more recognized and the configuration conversion tool does not take it into account.

2.6.2 Updating Your Tests

ReFrame 3.0 deprecates particular test syntax as well as certain test attributes. Some more esoteric features have also changed which may cause tests that make use of them to break. In this section we summarize all these changes and how to make these tests compatible with ReFrame 3.0

Pipeline methods and hooks

ReFrame 2.20 introduced a new powerful mechanism for attaching arbitrary functions hooks at the different pipeline stages. This mechanism provides an easy way to configure and extend the functionality of a test, eliminating essentially the need to override pipeline stages for this purpose. ReFrame 3.0 deprecates the old practice of overriding pipeline stage methods in favor of using pipeline hooks. In the old syntax, it was quite common to override the `setup()` method, in order to configure your test based on the current programming environment or the current system partition. The following is a typical example of that:

```
def setup(self, partition, environ, **job_opts):
    if environ.name == 'gnu':
        self.build_system.cflags = ['-fopenmp']
    elif environ.name == 'intel':
        self.build_system.cflags = ['-qopenmp']

    super().setup(partition, environ, **job_opts)
```

Alternatively, this example could have been written as follows:

```
def setup(self, partition, environ, **job_opts):
    super().setup(partition, environ, **job_opts)
    if self.current_environ.name == 'gnu':
        self.build_system.cflags = ['-fopenmp']
    elif self.current_environ.name == 'intel':
        self.build_system.cflags = ['-qopenmp']
```

This syntax now issues a deprecation warning. Rewriting this using pipeline hooks is quite straightforward and leads to nicer and more intuitive code:

```
@rfm.run_before('compile')
def setflags(self):
    if self.current_environ.name == 'gnu':
        self.build_system.cflags = ['-fopenmp']
    elif self.current_environ.name == 'intel':
        self.build_system.cflags = ['-qopenmp']
```

You could equally attach this function to run after the "setup" phase with `@rfm.run_after('setup')`, as in the original example, but attaching it to the "compile" phase makes more sense. However, you can't attach this function *before* the "setup" phase, because the `current_environ` will not be available and it will be still `None`.

Force override a pipeline method

Although pipeline hooks should be able to cover almost all the cases for writing tests in ReFrame, there might be corner cases that you need to override one of the pipeline methods, e.g., because you want to implement a stage differently. In this case, all you have to do is mark your test class as "special", and ReFrame will not issue any deprecation warning if you override pipeline stage methods:

```
class MyExtendedTest(rfm.RegressionTest, special=True):
    def setup(self, partition, environ, **job_opts):
        # do your custom stuff
        super().setup(partition, environ, **job_opts)
```

If you try to override the `setup()` method in any of the subclasses of `MyExtendedTest`, you will still get a deprecation warning, which a desired behavior since the subclasses should be normal tests.

Getting schedulers and launchers by name

The way to get a scheduler or launcher instance by name has changed. Prior to ReFrame 3, this was written as follows:

```
from reframe.core.launchers.registry import getlauncher


class MyTest(rfm.RegressionTest):
    ...

    @rfm.run_before('run')
    def setlauncher(self):
        self.job.launcher = getlauncher('local')()
```

Now you have to simply replace the import statement with the following:

```
from reframe.core.backends import getlauncher
```

Similarly for schedulers, the `reframe.core.schedulers.registry` module must be replaced with `reframe.core.backends`.

Other deprecations

The `prebuild_cmd` and `postbuild_cmd` test attributes are replaced by the `prebuild_cmds` and `postbuild_cmds` respectively. Similarly, the `pre_run` and `post_run` test attributes are replaced by the `prerun_cmds` and `postrun_cmds` respectively.

Suppressing deprecation warnings

Although not recommended, you can suppress any deprecation warning issued by ReFrame by passing the `--no-deprecation-warnings` flag.

2.6.3 Other Changes

ReFrame 3.0-dev0 introduced a change in the way that a search path for checks was constructed in the command-line using the `-c` option. ReFrame 3.0 reverts the behavior of the `-c` to its original one (i.e., ReFrame 2.x behavior), in which multiple paths can be specified by passing multiple times the `-c` option. Overriding completely the check search path can be achieved in ReFrame 3.0 through the *RFM_CHECK_SEARCH_PATH* environment variable or the corresponding configuration option.

2.7 ReFrame Manuals

2.7.1 ReFrame Command Line Reference

Synopsis

reframe [OPTION]... ACTION

Description

ReFrame provides both a programming interface for writing regression tests and a command-line interface for managing and running the tests, which is detailed here. The `reframe` command is part of ReFrame's frontend. This frontend is responsible for loading and running regression tests written in ReFrame. ReFrame executes tests by sending them down to a well defined pipeline. The implementation of the different stages of this pipeline is part of ReFrame's core architecture, but the frontend is responsible for driving this pipeline and executing tests through it. There are three basic phases that the frontend goes through, which are described briefly in the following.

Test discovery and test loading

This is the very first phase of the frontend. ReFrame will search for tests in its *check search path* and will load them. When ReFrame loads a test, it actually *instantiates* it, meaning that it will call its `__init__()` method unconditionally whether this test is meant to run on the selected system or not. This is something that writers of regression tests should bear in mind.

-c, --checkpath=PATH
: A filesystem path where ReFrame should search for tests. `PATH` can be a directory or a single test file. If it is a directory, ReFrame will search for test files inside this directory load all tests found in them. This option can be specified multiple times, in which case each `PATH` will be searched in order.

  The check search path can also be set using the *RFM_CHECK_SEARCH_PATH* environment variable or the `check_search_path` general configuration parameter.

-R, --recursive
: Search for test files recursively in directories found in the check search path.

  This option can also be set using the *RFM_CHECK_SEARCH_RECURSIVE* environment variable or the `check_search_recursive` general configuration parameter.

--ignore-check-conflicts
: Ignore tests with conflicting names when loading. ReFrame requires test names to be unique. Test names are used as components of the stage and output directory prefixes of tests, as well as for referencing target test dependencies. This option should generally be avoided unless there is a specific reason.

  This option can also be set using the *RFM_IGNORE_CHECK_CONFLICTS* environment variable or the `ignore_check_conflicts` general configuration parameter.

Test filtering

After all tests in the search path have been loaded, they are first filtered by the selected system. Any test that is not valid for the current system, it will be filtered out. The current system is either auto-selected or explicitly specified with the *--system* option. Tests can be filtered by different attributes and there are specific command line options for achieving this.

-t, --tag=TAG
: Filter tests by tag. `TAG` is interpreted as a Python Regular Expression; all tests that have at least a matching tag will be selected. `TAG` being a regular expression has the implication that `-t 'foo'` will select also tests that define `'foobar'` as a tag. To restrict the selection to tests defining only `'foo'`, you should use `-t 'foo$'`.

  This option may be specified multiple times, in which case only tests defining or matching *all* tags will be selected.

-n, --name=NAME
: Filter tests by name. `NAME` is interpreted as a Python Regular Expression; any test whose name matches `NAME` will be selected.

This option may be specified multiple times, in which case tests with *any* of the specified names will be selected: `-n NAME1 -n NAME2` is therefore equivalent to `-n 'NAME1|NAME2'`.

-x, --exclude=NAME
Exclude tests by name. `NAME` is interpreted as a Python Regular Expression; any test whose name matches `NAME` will be excluded.

This option may be specified multiple times, in which case tests with *any* of the specified names will be excluded: `-x NAME1 -x NAME2` is therefore equivalent to `-x 'NAME1|NAME2'`.

-p, --prgenv=NAME
Filter tests by programming environment. `NAME` is interpreted as a Python Regular Expression; any test for which at least one valid programming environment is matching `NAME` will be selected.

This option may be specified multiple times, in which case only tests matching all of the specified programming environments will be selected.

--gpu-only
Select tests that can run on GPUs. These are all tests with `num_gpus_per_node` greater than zero. This option and *`--cpu-only`* are mutually exclusive.

--cpu-only
Select tests that do not target GPUs. These are all tests with `num_gpus_per_node` equals to zero This option and *`--gpu-only`* are mutually exclusive.

The *`--gpu-only`* and *`--cpu-only`* check only the value of the `num_gpus_per_node` attribute of tests. The value of this attribute is not required to be non-zero for GPU tests. Tests may or may not make use of it.

--skip-system-check
Do not filter tests against the selected system.

--skip-prgenv-check
Do not filter tests against programming environments. Even if the *`-p`* option is not specified, ReFrame will filter tests based on the programming environments defined for the currently selected system. This option disables that filter completely.

Test actions

ReFrame will finally act upon the selected tests. There are currently two actions that can be performed on tests: (a) list the tests and (b) execute the tests. An action must always be specified.

-l, --list
List selected tests. A single line per test is printed.

-L, --list-detailed
List selected tests providing detailed information per test.

-r, --run
Execute the selected tests.

If more than one action options are specified, *`-l`* precedes *`-L`*, which in turn precedes *`-r`*.

Options controlling ReFrame output

--prefix=DIR
General directory prefix for ReFrame-generated directories. The base stage and output directories (see below) will be specified relative to this prefix if not specified explicitly.

This option can also be set using the *RFM_PREFIX* environment variable or the `prefix` system configuration parameter.

-o, --output=DIR
Directory prefix for test output files. When a test finishes successfully, ReFrame copies important output files to a test-specific directory for future reference. This test-specific directory is of the form `{output_prefix}/{system}/{partition}/{environment}/{test_name}`, where `output_prefix` is set by this option. The test files saved in this directory are the following:

- The ReFrame-generated build script, if not a run-only test.
- The standard output and standard error of the build phase, if not a run-only test.
- The ReFrame-generated job script, if not a compile-only test.
- The standard output and standard error of the run phase, if not a compile-only test.
- Any additional files specified by the `keep_files` regression test attribute.

This option can also be set using the *RFM_OUTPUT_DIR* environment variable or the `outputdir` system configuration parameter.

-s, --stage=DIR
Directory prefix for staging test resources. ReFrame does not execute tests from their original source directory. Instead it creates a test-specific stage directory and copies all test resources there. It then changes to that directory and executes the test. This test-specific directory is of the form `{stage_prefix}/{system}/{partition}/{environment}/{test_name}`, where `stage_prefix` is set by this option. If a test finishes successfully, its stage directory will be removed.

This option can also be set using the *RFM_STAGE_DIR* environment variable or the `stagedir` system configuration parameter.

--timestamp[=TIMEFMT]
Append a timestamp to the output and stage directory prefixes. `TIMEFMT` can be any valid *strftime(3)* time format. If not specified, `TIMEFMT` is set to `%FT%T`.

This option can also be set using the *RFM_TIMESTAMP_DIRS* environment variable or the `timestamp_dirs` general configuration parameter.

--perflogdir=DIR
Directory prefix for logging performance data. This option is relevant only to the `filelog` logging handler.

This option can also be set using the *RFM_PERFLOG_DIR* environment variable or the `basedir` logging handler configuration parameter.

--keep-stage-files
Keep test stage directories even for tests that finish successfully.

This option can also be set using the *RFM_KEEP_STAGE_FILES* environment variable or the `keep_stage_files` general configuration parameter.

--save-log-files
Save ReFrame log files in the output directory before exiting. Only log files generated by `file` log handlers will be copied.

This option can also be set using the *RFM_SAVE_LOG_FILES* environment variable or the `save_log_files` general configuration parameter.

Options controlling ReFrame execution

--force-local
Force local execution of tests. Execute tests as if all partitions of the currently selected system had a `local` scheduler.

--skip-sanity-check
Skip sanity checking phase.

--skip-performance-check
Skip performance checking phase. The phase is completely skipped, meaning that performance data will *not* be logged.

--strict
Enforce strict performance checking, even if a performance test is marked as not performance critical by having set its `strict_check` attribute to `False`.

--exec-policy=POLICY
The execution policy to be used for running tests. There are two policies defined:

- `serial`: Tests will be executed sequentially.
- `async`: Tests will be executed asynchronously. This is the default policy.

  The `async` execution policy executes the run phase of tests asynchronously by submitting their associated jobs in a non-blocking way. ReFrame's runtime monitors the progress of each test and will resume the pipeline execution of an asynchronously spawned test as soon as its run phase has finished. Note that the rest of the pipeline stages are still executed sequentially in this policy.

  Concurrency can be controlled by setting the `max_jobs` system partition configuration parameter. As soon as the concurrency limit is reached, ReFrame will first poll the status of all its pending tests to check if any execution slots have been freed up. If there are tests that have finished their run phase, ReFrame will keep pushing tests for execution until the concurrency limit is reached again. If no execution slots are available, ReFrame will throttle job submission.

--mode=MODE
ReFrame execution mode to use. An execution mode is simply a predefined invocation of ReFrame that is set with the `modes` configuration parameter. If an option is specified both in an execution mode and in the command-line, then command-line takes precedence.

--max-retries=NUM
The maximum number of times a failing test can be retried. The test stage and output directories will receive a `_retry<N>` suffix every time the test is retried.

Options controlling job submission

-A, --account=NAME
Submit test-related jobs using the account `NAME`. This option is relevant only for the Slurm backend and translates to Slurm's `--account` option and it precedes any options specified in the `access` system partition configuration parameter.

Deprecated since version 3.0: This is equivalent to `-J account=NAME` for Slurm.

-P, --partition=NAME
Submit test-related jobs using scheduler partition `NAME`. This option is relevant only for the Slurm, PBS and Torque backends and it translates to the `--partition` or `-q` scheduler options, respectively and it precedes any options specified in the `access` system partition configuration parameter.

Deprecated since version 3.0: This is equivalent to `-J partition=NAME` for Slurm or `-J q=NAME` for PBS/Torque.

--reservation=NAME
Submit test-related jobs on reservation `NAME`. This option is relevant only for the Slurm backend and translates to Slurm's `--reservation` option and it precedes any options specified in the `access` system partition configuration parameter.

Deprecated since version 3.0: This is equivalent to `-J reservation=NAME` for Slurm.

--nodelist=NODES
Submit test-related jobs on the selected nodes. This option is relevant only for the Slurm backend and translates to Slurm's `--nodelist` option and it precedes any options specified in the `access` system partition configuration parameter. The same node range naming conventions as of Slurm apply.

Deprecated since version 3.0: This is equivalent to `-J nodelist=NODES` for Slurm.

--exclude-nodes=NODES
Do not submit test-related jobs on the selected nodes. This option is relevant only for the Slurm backend and translates to Slurm's `--exclude` option and it precedes any options specified in the `access` system partition configuration parameter. The same node range naming conventions as of Slurm apply.

Deprecated since version 3.0: This is equivalent to `-J exclude=NODES` for Slurm.

-J, --job-option=OPTION
Pass `OPTION` directly to the job scheduler backend. The syntax for this option is `-J key=value`. If `key` starts with `-` or `#`, the option will be passed verbatim to the job script. Otherwise, ReFrame will add `-` or `--` as well as the directive corresponding to the current scheduler. This option will be emitted after any options specified in the `access` system partition configuration parameter.

Flexible node allocation

ReFrame can automatically set the number of tasks of a test, if its *num_tasks* attribute is set to a value less than or equal to zero. This scheme is conveniently called *flexible node allocation* and is valid only for the Slurm backend. When allocating nodes automatically, ReFrame will take into account all node limiting factors, such as partition `access` options, and any job submission control options described above. Nodes from this pool are allocated according to different policies. If no node can be selected, the test will be marked as a failure with an appropriate message.

--flex-alloc-nodes[=POLICY]
Set the flexible node allocation policy. Available values are the following:

- `all`: Flexible tests will be assigned as many tasks as needed in order to span over *all* the nodes of the node pool.
- `idle`: Flexible tests will be assigned as many tasks as needed in order to span over the *idle* nodes of the node pool. Querying of the node state and submission of the test job are two separate steps not executed atomically. It is therefore possible that the number of tasks assigned does not correspond to the actual idle nodes.

  This is the default policy.
- Any positive integer: Flexible tests will be assigned as many tasks as needed in order to span over the specified number of nodes from the node pool.

Options controlling ReFrame environment

ReFrame offers the ability to dynamically change its environment as well as the environment of tests. It does so by leveraging the selected system's environment modules system.

-m, --module=NAME
Load environment module NAME before acting on any tests. This option may be specified multiple times, in which case all specified modules will be loaded in order. ReFrame will *not* perform any automatic conflict resolution.

This option can also be set using the *RFM_USER_MODULES* environment variable or the user_modules general configuration parameter.

-u, --unload-module=NAME
Unload environment module NAME before acting on any tests. This option may be specified multiple times, in which case all specified modules will be unloaded in order.

This option can also be set using the *RFM_UNLOAD_MODULES* environment variable or the unload_modules general configuration parameter.

--purge-env
Unload all environment modules before acting on any tests. This will unload also sticky Lmod modules.

This option can also be set using the *RFM_PURGE_ENVIRONMENT* environment variable or the purge_environment general configuration parameter.

--non-default-craype
Test a non-default Cray Programming Environment. Since CDT 19.11, this option can be used in conjunction with *-m*, which will load the target CDT. For example:

```
reframe -m cdt/20.03 --non-default-craype -r
```

This option causes ReFrame to properly set the LD_LIBRARY_PATH for such cases. It will emit the following code after all the environment modules of a test have been loaded:

```
export LD_LIBRARY_PATH=$CRAY_LD_LIBRARY_PATH:$LD_LIBRARY_PATH
```

This option can also be set using the *RFM_NON_DEFAULT_CRAYPE* environment variable or the non_default_craype general configuration parameter.

-M, --map-module=MAPPING
Apply a module mapping. ReFrame allows manipulating test modules on-the-fly using module mappings. A module mapping has the form `old_module: module1 [module2]...` and will cause ReFrame to replace a module with another list of modules upon load time. For example, the mapping `foo: foo/1.2` will load module `foo/1.2` whenever module `foo` needs to be loaded. A mapping may also be self-referring, e.g., `gnu: gnu gcc/10.1`, however cyclic dependencies in module mappings are not allowed and ReFrame will issue an error if it detects one. This option is especially useful for running tests using a newer version of a software or library.

This option may be specified multiple times, in which case multiple mappings will be applied.

This option can also be set using the *RFM_MODULE_MAPPINGS* environment variable or the module_mappings general configuration parameter.

--module-mappings=FILE
A file containing module mappings. Each line of the file contains a module mapping in the form described in the *-M* option. This option may be combined with the *-M* option, in which case module mappings specified will be applied additionally.

This option can also be set using the *RFM_MODULE_MAP_FILE* environment variable or the `module_map_file` general configuration parameter.

Miscellaneous options

-C `--config-file=FILE`
Use `FILE` as configuration file for ReFrame.

This option can also be set using the *RFM_CONFIG_FILE* environment variable.

--show-config[`=PARAM]`
Show the value of configuration parameter `PARAM` as this is defined for the currently selected system and exit. The parameter value is printed in JSON format. If `PARAM` is not specified or if it set to `all`, the whole configuration for the currently selected system will be shown. Configuration parameters are formatted as a path navigating from the top-level configuration object to the actual parameter. The `/` character acts as a selector of configuration object properties or an index in array objects. The `@` character acts as a selector by name for configuration objects that have a `name` property. Here are some example queries:

- Retrieve all the partitions of the current system:

```
reframe --show-config=systems/0/partitions
```

- Retrieve the job scheduler of the partition named `default`:

```
reframe --show-config=systems/0/partitions/@default/scheduler
```

- Retrieve the check search path for system `foo`:

```
reframe --system=foo --show-config=general/0/check_search_path
```

--system`=NAME`
Load the configuration for system `NAME`. The `NAME` must be a valid system name in the configuration file. It may also have the form `SYSNAME:PARTNAME`, in which case the configuration of system `SYSNAME` will be loaded, but as if it had `PARTNAME` as its sole partition. Of course, `PARTNAME` must be a valid partition of system `SYSNAME`. If this option is not specified, ReFrame will try to pick the correct configuration entry automatically. It does so by trying to match the hostname of the current machine again the hostname patterns defined in the `hostnames` system configuration parameter. The system with the first match becomes the current system. For Cray systems, ReFrame will first look for the *unqualified machine name* in `/etc/xthostname` before trying retrieving the hostname of the current machine.

This option can also be set using the *RFM_SYSTEM* environment variable.

--failure-stats
Print failure statistics at the end of the run.

--performance-report
Print a performance report for all the performance tests that have been run. The report shows the performance values retrieved for the different performance variables defined in the tests.

--nocolor
Disable output coloring.

This option can also be set using the *RFM_COLORIZE* environment variable or the `colorize` general configuration parameter.

--upgrade-config-file`=OLD[:NEW]`
Convert the old-style configuration file `OLD`, place it into the new file `NEW` and exit. If a new file is not given, a file in the system temporary directory will be created.

-v, --verbose
Increase verbosity level of output. This option can be specified multiple times. Every time this option is specified, the verbosity level will be increased by one. There are the following message levels in ReFrame listed in increasing verbosity order: `critical`, `error`, `warning`, `info`, `verbose` and `debug`. The base verbosity level of the output is defined by the `level` stream logging handler configuration parameter.

This option can also be set using the *RFM_VERBOSE* environment variable or the `verbose` general configuration parameter.

-V, --version
Print version and exit.

-h, --help
Print a short help message and exit.

Environment

Several aspects of ReFrame can be controlled through environment variables. Usually environment variables have counterparts in command line options or configuration parameters. In such cases, command-line options take precedence over environment variables, which in turn precede configuration parameters. Boolean environment variables can have any value of `true`, `yes` or `y` (case insensitive) to denote true and any value of `false`, `no` or `n` (case insensitive) to denote false.

Here is an alphabetical list of the environment variables recognized by ReFrame:

RFM_CHECK_SEARCH_PATH
A colon-separated list of filesystem paths where ReFrame should search for tests.

Associated command line option	*-c*
Associated configuration parameter	`check_search_path` general configuration parameter

RFM_CHECK_SEARCH_RECURSIVE
Search for test files recursively in directories found in the check search path.

Associated command line option	*-R*
Associated configuration parameter	`check_search_recursive` general configuration parameter

RFM_COLORIZE
Enable output coloring.

Associated command line option	*--nocolor*
Associated configuration parameter	`colorize` general configuration parameter

RFM_CONFIG_FILE
Set the configuration file for ReFrame.

Associated command line option	*-C*
Associated configuration parameter	N/A

RFM_GRAYLOG_SERVER
The address of the Graylog server to send performance logs. The address is specified in `host:port` format.

Associated command line option	N/A
Associated configuration parameter	`address` graylog log handler configuration parameter

RFM_IGNORE_CHECK_CONFLICTS
Ignore tests with conflicting names when loading.

Associated command line option	*--ignore-check-conflicts*
Associated configuration parameter	`ignore_check_conflicts` general configuration parameter

RFM_IGNORE_REQNODENOTAVAIL
Do not treat specially jobs in pending state with the reason `ReqNodeNotAvail` (Slurm only).

Associated command line option	N/A
Associated configuration parameter	`ignore_reqnodenotavail` scheduler configuration parameter

RFM_KEEP_STAGE_FILES
Keep test stage directories even for tests that finish successfully.

Associated command line option	*--keep-stage-files*
Associated configuration parameter	`keep_stage_files` general configuration parameter

RFM_MODULE_MAP_FILE
A file containing module mappings.

Associated command line option	*--module-mappings*
Associated configuration parameter	`module_map_file` general configuration parameter

RFM_MODULE_MAPPINGS
A comma-separated list of module mappings.

Associated command line option	*-M*
Associated configuration parameter	`module_mappings` general configuration parameter

RFM_NON_DEFAULT_CRAYPE
Test a non-default Cray Programming Environment.

Associated command line option	*--non-default-craype*
Associated configuration parameter	`non_default_craype` general configuration parameter

RFM_OUTPUT_DIR
Directory prefix for test output files.

Associated command line option	*-o*
Associated configuration parameter	`outputdir` system configuration parameter

RFM_PERFLOG_DIR
Directory prefix for logging performance data.

Associated command line option	*--perflogdir*
Associated configuration parameter	`basedir` logging handler configuration parameter

RFM_PREFIX
General directory prefix for ReFrame-generated directories.

Associated command line option	*--prefix*
Associated configuration parameter	`prefix` system configuration parameter

RFM_PURGE_ENVIRONMENT
Unload all environment modules before acting on any tests.

Associated command line option	*--purge-env*
Associated configuration parameter	`purge_environment` general configuration parameter

RFM_SAVE_LOG_FILES
Save ReFrame log files in the output directory before exiting.

Associated command line option	*--save-log-files*
Associated configuration parameter	`save_log_files` general configuration parameter

RFM_STAGE_DIR
Directory prefix for staging test resources.

Associated command line option	*-s*
Associated configuration parameter	`stagedir` system configuration parameter

RFM_SYSTEM
Set the current system.

Associated command line option	*--system*
Associated configuration parameter	N/A

RFM_TIMESTAMP_DIRS
Append a timestamp to the output and stage directory prefixes.

Associated command line option	*--timestamp*
Associated configuration parameter	`timestamp_dirs` general configuration parameter.

RFM_UNLOAD_MODULES
A comma-separated list of environment modules to be unloaded before acting on any tests.

Associated command line option	*-u*
Associated configuration parameter	`unload_modules` general configuration parameter

RFM_USE_LOGIN_SHELL
Use a login shell for the generated job scripts.

Associated command line option	N/A
Associated configuration parameter	`use_login_shell` general configuration parameter

RFM_USER_MODULES
A comma-separated list of environment modules to be loaded before acting on any tests.

Associated command line option	*-m*
Associated configuration parameter	`user_modules` general configuration parameter

RFM_VERBOSE
Increase verbosity level of output.

Associated command line option	*-v*
Associated configuration parameter	`verbose` general configuration parameter

Configuration File

The configuration file of ReFrame defines the systems and environments to test as well as parameters controlling its behavior. Upon start up ReFrame checks for configuration files in the following locations in that order:

1. `$HOME/.reframe/settings.{py,json}`
2. `$RFM_INSTALL_PREFIX/settings.{py,json}`
3. `/etc/reframe.d/settings.{py,json}`

ReFrame accepts configuration files either in Python or JSON syntax. If both are found in the same location, the Python file will be preferred.

The `RFM_INSTALL_PREFIX` environment variable refers to the installation directory of ReFrame. Users have no control over this variable. It is always set by the framework upon startup.

If no configuration file can be found in any of the predefined locations, ReFrame will fall back to a generic configuration that allows it to run on any system. This configuration file is located in `reframe/core/settings.py`. Users may *not* modify this file.

For a complete reference of the configuration, please refer to `reframe.settings(8)` man page.

Reporting Bugs

For bugs, feature request, help, please open an issue on Github: <https://github.com/eth-cscs/reframe>

See Also

See full documentation online: <https://reframe-hpc.readthedocs.io/>

2.7.2 Configuration Reference

ReFrame's behavior can be configured through its configuration file (see *Configuring ReFrame for Your Site*), environment variables and command-line options. An option can be specified via multiple paths (e.g., a configuration file parameter and an environment variable), in which case command-line options precede environment variables, which in turn precede configuration file options. This section provides a complete reference guide of the configuration options of ReFrame that can be set in its configuration file or specified using environment variables.

ReFrame's configuration is in JSON syntax. The full schema describing it can be found in `reframe/schemas/config.json` file. Any configuration file given to ReFrame is validated against this schema.

The syntax we use in the following to describe the different configuration object attributes is a valid query string for the `jq(1)` command-line processor.

Top-level Configuration

The top-level configuration object is essentially the full configuration of ReFrame. It consists of the following properties:

`.systems`

Required Yes

A list of *system configuration objects*.

`.environments`

Required Yes

A list of *environment configuration objects*.

`.logging`

Required Yes

A list of *logging configuration objects*.

`.schedulers`

Required No

A list of *scheduler configuration objects*.

`.modes`

Required No

A list of *execution mode configuration objects*.

`.general`

Required No

A list of *general configuration objects*.

System Configuration

`.systems[].`**`name`**

Required Yes

The name of this system. Only alphanumeric characters, dashes (−) and underscores (_) are allowed.

`.systems[].`**`descr`**

Required No

Default `""`

The description of this system.

`.systems[].`**`hostnames`**

Required Yes

A list of hostname regular expression patterns in Python syntax, which will be used by the framework in order to automatically select a system configuration. For the auto-selection process, see here.

`.systems[].modules_system`

Required No

Default `"nomod"`

The modules system that should be used for loading environment modules on this system. Available values are the following:

- `tmod`: The classic Tcl implementation of the environment modules (version 3.2).
- `tmod31`: The classic Tcl implementation of the environment modules (version 3.1). A separate backend is required for Tmod 3.1, because Python bindings are different from Tmod 3.2.
- `tmod32`: A synonym of `tmod`.
- `tmod4`: The new environment modules implementation (versions older than 4.1 are not supported).
- `lmod`: The Lua implementation of the environment modules.
- `nomod`: This is to denote that no modules system is used by this system.

`.systems[].modules`

Required No

Default `[]`

Environment modules to be loaded always when running on this system. These modules modify the ReFrame environment. This is useful in cases where a particular module is needed, for example, to submit jobs on a specific system.

`.systems[].variables`

Required No

Default `[]`

A list of environment variables to be set always when running on this system. Each environment variable is specified as a two-element list containing the variable name and its value. You may reference other environment variables when defining an environment variable here. ReFrame will expand its value. Variables are set after the environment modules are loaded.

`.systems[].prefix`

Required No

Default `"."`

Directory prefix for a ReFrame run on this system. Any directories or files produced by ReFrame will use this prefix, if not specified otherwise.

`.systems[].stagedir`

Required No

Default `"${RFM_PREFIX}/stage"`

Stage directory prefix for this system. This is the directory prefix, where ReFrame will create the stage directories for each individual test case.

`.systems[].outputdir`

Required No

Default `"${RFM_PREFIX}/output"`

Output directory prefix for this system. This is the directory prefix, where ReFrame will save information about the successful tests.

`.systems[].resourcesdir`

Required No

Default `"."`

Directory prefix where external test resources (e.g., large input files) are stored. You may reference this prefix from within a regression test by accessing the *`reframe.core.systems.System.resourcesdir`* attribute of the current system.

`.systems[].partitions`

Required Yes

A list of *system partition configuration objects*. This list must have at least one element.

System Partition Configuration

`.systems[].partitions[].name`

Required Yes

The name of this partition. Only alphanumeric characters, dashes (−) and underscores (_) are allowed.

`.systems[].partitions[].descr`

Required No

Default `""`

The description of this partition.

`.systems[].partitions[].scheduler`

Required Yes

The job scheduler that will be used to launch jobs on this partition. Supported schedulers are the following:

- `local`: Jobs will be launched locally without using any job scheduler.
- `pbs`: Jobs will be launched using the PBS Pro scheduler.
- `torque`: Jobs will be launched using the Torque scheduler.
- `slurm`: Jobs will be launched using the Slurm scheduler. This backend requires job accounting to be enabled in the target system. If not, you should consider using the `squeue` backend below.
- `squeue`: Jobs will be launched using the Slurm scheduler. This backend does not rely on job accounting to retrieve job statuses, but ReFrame does its best to query the job state as reliably as possible.

`.systems[].partitions[].launcher`

Required Yes

The parallel job launcher that will be used in this partition to launch parallel programs. Available values are the following:

- `alps`: Parallel programs will be launched using the Cray ALPS `aprun` command.
- `ibrun`: Parallel programs will be launched using the `ibrun` command. This is a custom parallel program launcher used at TACC.
- `local`: No parallel program launcher will be used. The program will be launched locally.

- `mpirun`: Parallel programs will be launched using the `mpirun` command.
- `mpiexec`: Parallel programs will be launched using the `mpiexec` command.
- `srun`: Parallel programs will be launched using Slurm's `srun` command.
- `srunalloc`: Parallel programs will be launched using Slurm's `srun` command, but job allocation options will also be emitted. This can be useful when combined with the `local` job scheduler.
- `ssh`: Parallel programs will be launched using SSH. This launcher uses the partition's *`access`* property in order to determine the remote host and any additional options to be passed to the SSH client. The ssh command will be launched in "batch mode," meaning that password-less access to the remote host must be configured. Here is an example configuration for the ssh launcher:

```
{
    'name': 'foo'
    'scheduler': 'local',
    'launcher': 'ssh'
    'access': ['-l admin', 'remote.host'],
    'environs': ['builtin'],
}
```

- `upcrun`: Parallel programs will be launched using the UPC `upcrun` command.
- `upcxx-run`: Parallel programs will be launched using the UPC++ `upcxx-run` command.

`.systems[].partitions[].`**`access`**

Required No

Default `[]`

A list of job scheduler options that will be passed to the generated job script for gaining access to that logical partition.

`.systems[].partitions[].`**`environs`**

required No

default `[]`

A list of environment names that ReFrame will use to run regression tests on this partition. Each environment must be defined in the *`environments`* section of the configuration and the definition of the environment must be valid for this partition.

`.systems[].partitions[].`**`container_platforms`**

Required No

Default `[]`

A list for *container platform configuration objects*. This will allow launching regression tests that use containers on this partition.

`.systems[].partitions[].`**`modules`**

Required No

Default `[]`

A list of environment modules to be loaded before running a regression test on this partition.

`.systems[].partitions[].`**`variables`**

Required No

Default `[]`

A list of environment variables to be set before running a regression test on this partition. Each environment variable is specified as a two-element list containing the variable name and its value. You may reference other environment variables when defining an environment variable here. ReFrame will expand its value. Variables are set after the environment modules are loaded.

`.systems[].partitions[].max_jobs`

Required No

Default `1`

The maximum number of concurrent regression tests that may be active (i.e., not completed) on this partition. This option is relevant only when ReFrame executes with the asynchronous execution policy.

`.systems[].partitions[].resources`

Required No

Default `[]`

A list of job scheduler resource specification objects.

Container Platform Configuration

ReFrame can launch containerized applications, but you need to configure properly a system partition in order to do that by defining a container platform configuration.

`.systems[].partitions[].container_platforms[].type`

Required Yes

The type of the container platform. Available values are the following:

- `Docker`: The Docker container runtime.
- `Sarus`: The Sarus container runtime.
- `Shifter`: The Shifter container runtime.
- `Singularity`: The Singularity container runtime.

`.systems[].partitions[].container_platforms[].modules`

required No

default `[]`

List of environment modules to be loaded when running containerized tests using this container platform.

`.systems[].partitions[].container_platforms[].variables`

Required No

Default `[]`

List of environment variables to be set when running containerized tests using this container platform. Each environment variable is specified as a two-element list containing the variable name and its value. You may reference other environment variables when defining an environment variable here. ReFrame will expand its value. Variables are set after the environment modules are loaded.

Custom Job Scheduler Resources

ReFrame allows you to define custom scheduler resources for each partition that you can then transparently access through the `extra_resources` attribute of a regression test.

`.systems[].partitions[].resources[].`**`name`**

required Yes

The name of this resources. This name will be used to request this resource in a regression test's `extra_resources`.

`.systems[].partitions[].resources[].`**`options`**

required No

default `[]`

A list of options to be passed to this partition's job scheduler. The option strings can contain placeholders of the form `{placeholder_name}`. These placeholders may be replaced with concrete values by a regression test through the `extra_resources` attribute.

For example, one could define a `gpu` resource for a multi-GPU system that uses Slurm as follows:

```
'resources': [
    {
        'name': 'gpu',
        'options': ['--gres=gpu:{num_gpus_per_node}']
    }
]
```

A regression test then may request this resource as follows:

```
self.extra_resources = {'gpu': {'num_gpus_per_node': '8'}}
```

And the generated job script will have the following line in its preamble:

```
#SBATCH --gres=gpu:8
```

A resource specification may also start with `#PREFIX`, in which case `#PREFIX` will replace the standard job script prefix of the backend scheduler of this partition. This is useful in cases of job schedulers like Slurm, that allow alternative prefixes for certain features. An example is the DataWarp functionality of Slurm which is supported by the `#DW` prefix. One could then define DataWarp related resources as follows:

```
'resources': [
    {
        'name': 'datawarp',
        'options': [
            '#DW jobdw capacity={capacity} access_mode={mode} type=scratch
↪',
            '#DW stage_out source={out_src} destination={out_dst} type=
↪{stage_filetype}'
        ]
    }
]
```

A regression test that wants to make use of that resource, it can set its `extra_resources` as follows:

```
self.extra_resources = {
    'datawarp': {
        'capacity': '100GB',
        'mode': 'striped',
        'out_src': '$DW_JOB_STRIPED/name',
        'out_dst': '/my/file',
        'stage_filetype': 'file'
    }
}
```

Note: For the `pbs` and `torque` backends, options accepted in the *`access`* and *`resources`* attributes may either refer to actual `qsub` options or may be just resources specifications to be passed to the `-l` option. The backend assumes a `qsub` option, if the options passed in these attributes start with a `-`.

Environment Configuration

Environments defined in this section will be used for running regression tests. They are associated with *system partitions*.

`.environments[].`**`name`**

Required Yes

The name of this environment.

`.environments[].`**`modules`**

Required No

Default `[]`

A list of environment modules to be loaded when this environment is loaded.

`.environments[].`**`variables`**

Required No

Default `[]`

A list of environment variables to be set when loading this environment. Each environment variable is specified as a two-element list containing the variable name and its value. You may reference other environment variables when defining an environment variable here. ReFrame will expand its value. Variables are set after the environment modules are loaded.

`.environments[].`**`cc`**

Required No

Default `"cc"`

The C compiler to be used with this environment.

`.environments[].`**`cxx`**

Required No

Default `"CC"`

The C++ compiler to be used with this environment.

`.environments[].`**`ftn`**

Required No

Default `"ftn"`

The Fortran compiler to be used with this environment.

`.environments[].`**`cppflags`**

Required No

Default `[]`

A list of C preprocessor flags to be used with this environment by default.

`.environments[].`**`cflags`**

Required No

Default `[]`

A list of C flags to be used with this environment by default.

`.environments[].`**`cxxflags`**

Required No

Default `[]`

A list of C++ flags to be used with this environment by default.

`.environments[].`**`fflags`**

Required No

Default `[]`

A list of Fortran flags to be used with this environment by default.

`.environments[].`**`ldflags`**

Required No

Default `[]`

A list of linker flags to be used with this environment by default.

`.environments[].`**`target_systems`**

Required No

Default `["*"]`

A list of systems or system/partitions combinations that this environment definition is valid for. A `*` entry denotes any system. In case of multiple definitions of an environment, the most specific to the current system partition will be used. For example, if the current system/partition combination is `daint:mc`, the second definition of the `PrgEnv-gnu` environment will be used:

```
'environments': [
    {
        'name': 'PrgEnv-gnu',
        'modules': ['PrgEnv-gnu']
    },
    {
        'name': 'PrgEnv-gnu',
        'modules': ['PrgEnv-gnu', 'openmpi'],
        'cc':  'mpicc',
        'cxx': 'mpicxx',
```

(continues on next page)

(continued from previous page)

```
        'ftn': 'mpif90',
        'target_systems': ['daint:mc']
    }
]
```

However, if the current system was `daint:gpu`, the first definition would be selected, despite the fact that the second definition is relevant for another partition of the same system. To better understand this, ReFrame resolves definitions in a hierarchical way. It first looks for definitions for the current partition, then for the containing system and, finally, for global definitions (the `*` pseudo-system).

Logging Configuration

Logging in ReFrame is handled by logger objects which further delegate message to *logging handlers* which are eventually responsible for emitting or sending the log records to their destinations. You may define different logger objects per system but *not* per partition.

`.logging[].level`

Required No

Default `"debug"`

The level associated with this logger object. There are the following levels in decreasing severity order:

- `critical`: Catastrophic errors; the framework cannot proceed with its execution.
- `error`: Normal errors; the framework may or may not proceed with its execution.
- `warning`: Warning messages.
- `info`: Informational messages.
- `verbose`: More informational messages.
- `debug`: Debug messages.

If a message is logged by the framework, its severity level will be checked by the logger and if it is higher from the logger's level, it will be passed down to its handlers.

`.logging[].handlers`

Required Yes

A list of logging handlers responsible for handling normal framework output.

`.logging[].handlers_perflog`

Required Yes

A list of logging handlers responsible for handling performance data from tests.

`.logging[].target_systems`

Required No

Default `["*"]`

A list of systems or system/partitions combinations that this logging configuration is valid for. For a detailed description of this property, you may refer *here*.

Common logging handler properties

All logging handlers share the following set of common attributes:

`.logging[].handlers[].type`

`.logging[].handlers_perflog[].type`

Required Yes

The type of handler. There are the following types available:

- `file`: This handler sends log records to file. See *here* for more details.
- `filelog`: This handler sends performance log records to files. See *here* for more details.
- `graylog`: This handler sends performance log records to Graylog. See *here* for more details.
- `stream`: This handler sends log records to a file stream. See *here* for more details.
- `syslog`: This handler sends log records to a Syslog facility. See *here* for more details.

`.logging[].handlers[].level`

`.logging[].handlers_perflog[].level`

Required No

Default `"info"`

The *log level* associated with this handler.

`.logging[].handlers[].format`

`.logging[].handlers_perflog[].format`

Required No

Default `"%(message)s"`

Log record format string. ReFrame accepts all log record attributes from Python's logging mechanism and adds the following:

- `%(check_environ)s`: The name of the *environment* that the current test is being executing for.
- `%(check_info)s`: General information of the currently executing check. By default this field has the form `%(check_name)s on %(check_system)s:%(check_partition)s using %(check_environ)s`. It can be configured on a per test basis by overriding the *`info`* method of a specific regression test.
- `%(check_jobid)s`: The job or process id of the job or process associated with the currently executing regression test. If a job or process is not yet created, `-1` will be printed.
- `%(check_job_completion_time)s`: The completion time of the job spawned by this regression test. This timestamp will be formatted according to *`datefmt`* handler property. The accuracy of this timestamp depends on the backend scheduler. The `slurm` scheduler *backend* relies on job accounting and returns the actual termination time of the job. The rest of the backends report as completion time the moment when the framework realizes that the spawned job has finished. In this case, the accuracy depends on the execution policy used. If tests are executed with the serial execution policy, this is close to the real completion time, but if the asynchronous execution policy is used, it can differ significantly. If the job completion time cannot be retrieved, `None` will be printed.
- `%(check_job_completion_time_unix)s`: The completion time of the job spawned by this regression test expressed as UNIX time. This is a raw time field and will not be formatted according to `datefmt`. If specific formatting is desired, the `check_job_completion_time` should be used instead.

- `%(check_name)s`: The name of the regression test on behalf of which ReFrame is currently executing. If ReFrame is not executing in the context of a regression test, `reframe` will be printed instead.
- `%(check_num_tasks)s`: The number of tasks assigned to the regression test.
- `%(check_outputdir)s`: The output directory associated with the currently executing test.
- `%(check_partition)s`: The system partition where this test is currently executing.
- `%(check_stagedir)s`: The stage directory associated with the currently executing test.
- `%(check_system)s`: The system where this test is currently executing.
- `%(check_tags)s`: The tags associated with this test.
- `%(check_perf_lower_thres)s`: The lower threshold of the performance difference from the reference value expressed as a fractional value. See the *`reframe.core.pipeline.RegressionTest.reference`* attribute of regression tests for more details.
- `%(check_perf_ref)s`: The reference performance value of a certain performance variable.
- `%(check_perf_unit)s`: The unit of measurement for the measured performance variable.
- `%(check_perf_upper_thres)s`: The upper threshold of the performance difference from the reference value expressed as a fractional value. See the *`reframe.core.pipeline.RegressionTest.reference`* attribute of regression tests for more details.
- `%(check_perf_value)s`: The performance value obtained for a certain performance variable.
- `%(check_perf_var)s`: The name of the performance variable being logged.
- `%(osuser)s`: The name of the OS user running ReFrame.
- `%(osgroup)s`: The name of the OS group running ReFrame.
- `%(version)s`: The ReFrame version.

`.logging[].handlers[].`**`datefmt`**

`.logging[].handlers_perflog[].`**`datefmt`**

Required No

Default `"%FT%T"`

Time format to be used for printing timestamps fields. There are two timestamp fields available: `%(asctime)s` and `%(check_job_completion_time)s`. In addition to the format directives supported by the standard library's time.strftime() function, ReFrame allows you to use the `%:z` directive – a GNU `date` extension – that will print the time zone difference in a RFC3339 compliant way, i.e., `+/-HH:MM` instead of `+/-HHMM`.

The `file` log handler

This log handler handles output to normal files. The additional properties for the `file` handler are the following:

`.logging[].handlers[].`**`name`**

`.logging[].handlers_perflog[].`**`name`**

Required Yes

The name of the file where this handler will write log records.

`.logging[].handlers[].`**`append`**

`.logging[].handlers_perflog[].`**`append`**

Required No

Default `false`

Controls whether this handler should append to its file or not.

`.logging[].handlers[].`**`timestamp`**

`.logging[].handlers_perflog[].timestamp`

Required No

Default `false`

Append a timestamp to this handler's log file. This property may also accept a date format as described in the *`datefmt`* property. If the handler's *`name`* property is set to `filename.log` and this property is set to `true` or to a specific timestamp format, the resulting log file will be `filename_<timestamp>.log`.

The `filelog` log handler

This handler is meant primarily for performance logging and logs the performance of a regression test in one or more files. The additional properties for the `filelog` handler are the following:

`.logging[].handlers[].`**`basedir`**

`.logging[].handlers_perflog[].basedir`

Required No

Default `"./perflogs"`

The base directory of performance data log files.

`.logging[].handlers[].`**`prefix`**

`.logging[].handlers_perflog[].prefix`

Required Yes

This is a directory prefix (usually dynamic), appended to the *`basedir`*, where the performance logs of a test will be stored. This attribute accepts any of the check-specific *formatting placeholders*. This allows to create dynamic paths based on the current system, partition and/or programming environment a test executes with. For example, a value of `%(check_system)s/%(check_partition)s` would generate the following structure of performance log files:

```
{basedir}/
   system1/
       partition1/
           test_name.log
       partition2/
           test_name.log
       ...
   system2/
   ...
```

`.logging[].handlers[].append`

`.logging[].handlers_perflog[].append`

Required No

Default `true`

Open each log file in append mode.

The `graylog` log handler

This handler sends log records to a Graylog server. The additional properties for the `graylog` handler are the following:

`.logging[].handlers[].address`

`.logging[].handlers_perflog[].address`

Required Yes

The address of the Graylog server defined as `host:port`.

`.logging[].handlers[].extras`

`.logging[].handlers_perflog[].extras`

Required No

Default `{}`

A set of optional key/value pairs to be passed with each log record to the server. These may depend on the server configuration.

This log handler uses internally pygelf. If `pygelf` is not available, this log handler will be ignored. GELF is a format specification for log messages that are sent over the network. The `graylog` handler sends log messages in JSON format using an HTTP POST request to the specified address. More details on this log format may be found here. An example configuration of this handler for performance logging is shown here:

```
{
    'type': 'graylog',
    'address': 'graylog-server:12345',
    'level': 'info',
    'format': '%(message)s',
    'extras': {
        'facility': 'reframe',
        'data-version': '1.0'
    }
}
```

Although the `format` is defined for this handler, it is not only the log message that will be transmitted the Graylog server. This handler transmits the whole log record, meaning that all the information will be available and indexable at the remote end.

The `stream` log handler

This handler sends log records to a file stream. The additional properties for the `stream` handler are the following:

`.logging[].handlers[].name`

`.logging[].handlers_perflog[].name`

Required No

Default `"stdout"`

The name of the file stream to send records to. There are only two available streams:

- `stdout`: the standard output.
- `stderr`: the standard error.

The `syslog` log handler

This handler sends log records to UNIX syslog. The additional properties for the `syslog` handler are the following:

`.logging[].handlers[].`**`socktype`**

`.logging[].handlers_perflog[].socktype`

Required No

Default `"udp"`

The socket type where this handler will send log records to. There are two socket types:

- `udp`: A UDP datagram socket.
- `tcp`: A TCP stream socket.

`.logging[].handlers[].`**`facility`**

`.logging[].handlers_perflog[].facility`

Required No

Default `"user"`

The Syslog facility where this handler will send log records to. The list of supported facilities can be found here.

`.logging[].handlers[].address`

`.logging[].handlers_perflog[].address`

Required Yes

The socket address where this handler will connect to. This can either be of the form `<host>:<port>` or simply a path that refers to a Unix domain socket.

Scheduler Configuration

A scheduler configuration object contains configuration options specific to the scheduler's behavior.

Common scheduler options

`.schedulers[].`**`name`**

Required Yes

The name of the scheduler that these options refer to. It can be any of the supported job scheduler *backends*.

`.schedulers[].`**`job_submit_timeout`**

Required No

Default 60

Timeout in seconds for the job submission command. If timeout is reached, the regression test issuing that command will be marked as a failure.

`.schedulers[].`**`target_systems`**

Required No

Default `["*"]`

A list of systems or system/partitions combinations that this scheduler configuration is valid for. For a detailed description of this property, you may refer *here*.

`.schedulers[].use_nodes_option`

Required No

Default `false`

Always emit the `--nodes` Slurm option in the preamble of the job script. This option is relevant to Slurm backends only.

`.schedulers[].ignore_reqnodenotavail`

Required No

Default `false`

This option is relevant to the Slurm backends only.

If a job associated to a test is in pending state with the Slurm reason `ReqNodeNotAvail` and a list of unavailable nodes is also specified, ReFrame will check the status of the nodes and, if all of them are indeed down, it will cancel the job. Sometimes, however, when Slurm's backfill algorithm takes too long to compute, Slurm will set the pending reason to `ReqNodeNotAvail` and mark all system nodes as unavailable, causing ReFrame to kill the job. In such cases, you may set this parameter to `true` to avoid this.

Execution Mode Configuration

ReFrame allows you to define groups of command line options that are collectively called *execution modes*. An execution mode can then be selected from the command line with the `-mode` option. The options of an execution mode will be passed to ReFrame as if they were specified in the command line.

`.modes[].name`

Required Yes

The name of this execution mode. This can be used with the `-mode` command line option to invoke this mode.

`.modes[].options`

Required No

Default `[]`

The command-line options associated with this execution mode.

`.modes[].target_systems`

Required No

Default `["*"]`

A list of systems or system/partitions combinations that this execution mode is valid for. For a detailed description of this property, you may refer *here*.

General Configuration

`.general[].check_search_path`

Required No

Default `["${RFM_INSTALL_PREFIX}/checks/"]`

A list of paths (files or directories) where ReFrame will look for regression test files. If the search path is set through the environment variable, it should be a colon separated list. If specified from command line, the search path is constructed by specifying multiple times the command line option.

`.general[].check_search_recursive`

Required No

Default `false`

Search directories in the *search path* recursively.

`.general[].colorize`

Required No

Default `true`

Use colors in output. The command-line option sets the configuration option to `false`.

`.general[].ignore_check_conflicts`

Required No

Default `false`

Ignore test name conflicts when loading tests.

`.general[].keep_stage_files`

Required No

Default `false`

Keep stage files of tests even if they succeed.

`.general[].module_map_file`

Required No

Default `""`

File containing module mappings.

`.general[].module_mappings`

Required No

Default `[]`

A list of module mappings. If specified through the environment variable, the mappings must be separated by commas. If specified from command line, multiple module mappings are defined by passing the command line option multiple times.

`.general[].non_default_craype`

Required No

Default `false`

Test a non-default Cray Programming Environment. This will emit some special instructions in the generated build and job scripts. See also *--non-default-craype* for more details.

`.general[].purge_environment`

Required No

Default `false`

Purge any loaded environment modules before running any tests.

`.general[].save_log_files`

Required No

Default `false`

Save any log files generated by ReFrame to its output directory

`.general[].target_systems`

Required No

Default `["*"]`

A list of systems or system/partitions combinations that these general options are valid for. For a detailed description of this property, you may refer *here*.

`.general[].timestamp_dirs`

Required No

Default `""`

Append a timestamp to ReFrame directory prefixes. Valid formats are those accepted by the time.strftime() function. If specified from the command line without any argument, `"%FT%T"` will be used as a time format.

`.general[].unload_modules`

Required No

Default `[]`

A list of environment modules to unload before executing any test. If specified using an the environment variable, a space separated list of modules is expected. If specified from the command line, multiple modules can be passed by passing the command line option multiple times.

`.general[].use_login_shell`

Required No

Default `false`

Use a login shell for the generated job scripts. This option will cause ReFrame to emit `-l` in the shebang of shell scripts. This option, if set to `true`, may cause ReFrame to fail, if the shell changes permanently to a different directory during its start up.

`.general[].user_modules`

Required No

Default `[]`

A list of environment modules to be loaded before executing any test. If specified using an the environment variable, a space separated list of modules is expected. If specified from the command line, multiple modules can be passed by passing the command line option multiple times.

`.general[].verbose`

Required No

Default 0

Increase the verbosity level of the output. The higher the number, the more verbose the output will be. If specified from the command line, the command line option must be specified multiple times to increase the verbosity level more than once.

2.7.3 ReFrame Programming APIs

This page provides a reference guide of the ReFrame API for writing regression tests covering all the relevant details. Internal data structures and APIs are covered only to the extent that this might be helpful to the final user of the framework.

Regression Test Base Classes

class reframe.core.pipeline.**CompileOnlyRegressionTest**

Bases: *reframe.core.pipeline.RegressionTest*

Base class for compile-only regression tests.

These tests are by default local and will skip the run phase of the regression test pipeline.

The standard output and standard error of the test will be set to those of the compilation stage.

This class is also directly available under the top-level `reframe` module.

run()

The run stage of the regression test pipeline.

Implemented as no-op.

setup(*partition*, *environ*, ***job_opts*)

The setup stage of the regression test pipeline.

Similar to the *RegressionTest.setup()*, except that no job descriptor is set up for this test.

property stderr

The name of the file containing the standard error of the test.

This is set during the *setup()* phase.

This attribute is evaluated lazily, so it can by used inside sanity expressions.

Type str.

property stdout

The name of the file containing the standard output of the test.

This is set during the *setup()* phase.

This attribute is evaluated lazily, so it can by used inside sanity expressions.

Type str.

wait()

Wait for this test to finish.

Implemented as no-op

class reframe.core.pipeline.**RegressionTest**
Bases: object

Base class for regression tests.

All regression tests must eventually inherit from this class. This class provides the implementation of the pipeline phases that the regression test goes through during its lifetime.

Note: Changed in version 2.19: Base constructor takes no arguments.

---

build_system
New in version 2.14.

The build system to be used for this test. If not specified, the framework will try to figure it out automatically based on the value of *sourcepath*.

This field may be set using either a string referring to a concrete build system class name (see *build systems*) or an instance of *reframe.core.buildsystems.BuildSystem*. The former is the recommended way.

Type str or *reframe.core.buildsystems.BuildSystem*.

Default None.

check_performance()
The performance checking phase of the regression test pipeline.

Raises **reframe.core.exceptions.SanityError** – If the performance check fails.

> **Warning:** Changed in version 3.0: You may not override this method directly unless you are in special test. See here for more details.

check_sanity()
The sanity checking phase of the regression test pipeline.

Raises **reframe.core.exceptions.SanityError** – If the sanity check fails.

> **Warning:** Changed in version 3.0: You may not override this method directly unless you are in special test. See here for more details.

cleanup(*remove_files=False*)
The cleanup phase of the regression test pipeline.

Parameters **remove_files** – If True, the stage directory associated with this test will be removed.

> **Warning:** Changed in version 3.0: You may not override this method directly unless you are in special test. See here for more details.

compile()
The compilation phase of the regression test pipeline.

Raises **reframe.core.exceptions.ReframeError** – In case of errors.

> **Warning:** Changed in version 3.0: You may not override this method directly unless you are in special test. See here for more details.

compile_wait()
Wait for compilation phase to finish.

New in version 2.13.

> **Warning:** Changed in version 3.0: You may not override this method directly unless you are in special test. See here for more details.

container_platform
New in version 2.20.

The container platform to be used for launching this test.

If this field is set, the test will run inside a container using the specified container runtime. Container-specific options must be defined additionally after this field is set:

```
self.container_platform = 'Singularity'
self.container_platform.image = 'docker://ubuntu:18.04'
self.container_platform.commands = ['cat /etc/os-release']
```

If this field is set, *executable* and *executable_opts* attributes are ignored. The container platform's *commands* will be used instead.

Type str or *reframe.core.containers.ContainerPlatform*.

Default None.

property current_environ
The programming environment that the regression test is currently executing with.

This is set by the framework during the *setup()* phase.

Type *reframe.core.environments.ProgEnvironment*.

property current_partition
The system partition the regression test is currently executing on.

This is set by the framework during the *setup()* phase.

Type *reframe.core.systems.SystemPartition*.

property current_system
The system the regression test is currently executing on.

This is set by the framework during the initialization phase.

Type *reframe.core.systems.System*.

depends_on(*target*, *how=2*, *subdeps=None*)
Add a dependency to target in this test.

Parameters

- **target** – The name of the target test.
- **how** – How the dependency should be mapped in the test cases space. This argument can accept any of the three constants *DEPEND_EXACT*, *DEPEND_BY_ENV* (default), *DEPEND_FULLY*.

- **subdeps** – An adjacency list representation of how this test's test cases depend on those of the target test. This is only relevant if `how == DEPEND_EXACT`. The value of this argument is a dictionary having as keys the names of this test's supported programming environments. The values are lists of the programming environments names of the target test that this test's test cases will depend on. In the following example, this test's `E0` programming environment case will depend on both `E0` and `E1` test cases of the target test `T0`, but its `E1` case will depend only on the `E1` test case of `T0`:

```
self.depends_on('T0', how=rfm.DEPEND_EXACT,
                subdeps={'E0': ['E0', 'E1'], 'E1': ['E1']})
```

For more details on how test dependencies work in ReFrame, please refer to How Test Dependencies Work In ReFrame.

New in version 2.21.

descr
A detailed description of the test.

Type `str`

Default `self.name`

exclusive_access
Specify whether this test needs exclusive access to nodes.

Type boolean

Default `False`

executable
The name of the executable to be launched during the run phase.

Type `str`

Default `os.path.join('.', self.name)`

executable_opts
List of options to be passed to the *executable*.

Type `List[str]`

Default `[]`

extra_resources
New in version 2.8.

Extra resources for this test.

This field is for specifying custom resources needed by this test. These resources are defined in the configuration of a system partition. For example, assume that two additional resources, named `gpu` and `datawarp`, are defined in the configuration file as follows:

```
'resources': {
    'gpu': [
        '--gres=gpu:{num_gpus_per_node}'
    ],
    'datawarp': [
        '#DW jobdw capacity={capacity}',
        '#DW stage_in source={stagein_src}'
    ]
}
```

A regression test then may instantiate the above resources by setting the *extra_resources* attribute as follows:

```
self.extra_resources = {
    'gpu': {'num_gpus_per_node': 2}
    'datawarp': {
        'capacity': '100GB',
        'stagein_src': '/foo'
    }
}
```

The generated batch script (for Slurm) will then contain the following lines:

```
#SBATCH --gres=gpu:2
#DW jobdw capacity=100GB
#DW stage_in source=/foo
```

Notice that if the resource specified in the configuration uses an alternative directive prefix (in this case `#DW`), this will replace the standard prefix of the backend scheduler (in this case `#SBATCH`)

If the resource name specified in this variable does not match a resource name in the partition configuration, it will be simply ignored. The *num_gpus_per_node* attribute translates internally to the `_rfm_gpu` resource, so that setting `self.num_gpus_per_node = 2` is equivalent to the following:

```
self.extra_resources = {'_rfm_gpu': {'num_gpus_per_node': 2}}
```

Type `Dict[str, Dict[str, object]]`

Default `{}`

Note: Changed in version 2.9: A new more powerful syntax was introduced that allows also custom job script directive prefixes.

---

getdep(*target*, *environ=None*)

Retrieve the test case of a target dependency.

This is a low-level method. The *@require_deps* decorators should be preferred.

Parameters

- **target** – The name of the target dependency to be retrieved.
- **environ** – The name of the programming environment that will be used to retrieve the test case of the target test. If `None`, *RegressionTest.current_environ* will be used.

New in version 2.21.

info()

Provide live information for this test.

This method is used by the front-end to print the status message during the test's execution. This function is also called to provide the message for the check_info logging attribute. By default, it returns a message reporting the test name, the current partition and the current programming environment that the test is currently executing on.

New in version 2.10.

Returns a string with an informational message about this test

Note: When overriding this method, you should pay extra attention on how you use the *RegressionTest*'s attributes, because this method may be called at any point of the test's lifetime.

is_local()
Check if the test will execute locally.

A test executes locally if the *local* attribute is set or if the current partition's scheduler does not support job submission.

property job
The job descriptor associated with this test.

This is set by the framework during the *setup()* phase.

Type *reframe.core.schedulers.Job*.

keep_files
List of files to be kept after the test finishes.

By default, the framework saves the standard output, the standard error and the generated shell script that was used to run this test.

These files will be copied over to the framework's output directory during the *cleanup()* phase.

Directories are also accepted in this field.

Relative path names are resolved against the stage directory.

Type `List[str]`

Default `[]`

local
Always execute this test locally.

Type boolean

Default `False`

property logger
A logger associated with this test.

You can use this logger to log information for your test.

maintainers
List of people responsible for this test.

When the test fails, this contact list will be printed out.

Type `List[str]`

Default `[]`

max_pending_time
New in version 3.0.

The maximum time a job can be pending before starting running.

Time duration is specified as of the *time_limit* attribute.

Type `str` or `datetime.timedelta`

Default `None`

modules
List of modules to be loaded before running this test.

These modules will be loaded during the *setup()* phase.

Type List[str]

Default []

name
The name of the test.

Type string that can contain any character except /

num_cpus_per_task
Number of CPUs per task required by this test.

Ignored if None.

Type integral or None

Default None

num_gpus_per_node
Number of GPUs per node required by this test.

Type integral

Default 0

num_tasks
Number of tasks required by this test.

If the number of tasks is set to a number <=0, ReFrame will try to flexibly allocate the number of tasks, based on the command line option --flex-alloc-nodes. A negative number is used to indicate the minimum number of tasks required for the test. In this case the minimum number of tasks is the absolute value of the number, while Setting *num_tasks* to 0 is equivalent to setting it to *-num_tasks_per_node*.

Type integral

Default 1

Note: Changed in version 2.15: Added support for flexible allocation of the number of tasks if the number of tasks is set to 0.

Changed in version 2.16: Negative *num_tasks* is allowed for specifying the minimum number of required tasks by the test.

---

num_tasks_per_core
Number of tasks per core required by this test.

Ignored if None.

Type integral or None

Default None

num_tasks_per_node
Number of tasks per node required by this test.

Ignored if None.

Type integral or None

Default `None`

num_tasks_per_socket
Number of tasks per socket required by this test.

Ignored if `None`.

Type integral or `None`

Default `None`

property outputdir
The output directory of the test.

This is set during the *setup()* phase.

New in version 2.13.

Type `str`.

perf_patterns
Patterns for verifying the performance of this test.

Refer to the *ReFrame Tutorials* for concrete usage examples.

If set to `None`, no performance checking will be performed.

Type A dictionary with keys of type `str` and deferrable expressions (i.e., the result of a *sanity function*) as values. `None` is also allowed.

Default `None`

poll()
Poll the test's state.

Returns

`True` if the associated job has finished, `False` otherwise.

If no job descriptor is yet associated with this test, `True` is returned.

Raises **reframe.core.exceptions.ReframeError** – In case of errors.

> **Warning:** Changed in version 3.0: You may not override this method directly unless you are in special test. See here for more details.

post_run
Deprecated since version 3.0.

Use *postrun_cmds* instead.

postbuild_cmd
Deprecated since version 3.0.

Use *postbuild_cmds* instead.

postbuild_cmds
New in version 3.0.

List of shell commands to be executed after a successful compilation.

These commands are emitted in the script after the actual build commands generated by the selected *build system*.

Type `List[str]`

Default `[]`

postrun_cmds
New in version 3.0.

List of shell commands to execute after launching this job.

See *prerun_cmds* for a more detailed description of the semantics.

Type `List[str]`

Default `[]`

pre_run
Deprecated since version 3.0.

Use *prerun_cmds* instead.

prebuild_cmd
Deprecated since version 3.0.

Use *prebuild_cmds* instead.

prebuild_cmds
New in version 3.0.

List of shell commands to be executed before compiling.

These commands are emitted in the build script before the actual build commands generated by the selected *build system*.

Type `List[str]`

Default `[]`

property prefix
The prefix directory of the test.

Type `str`.

prerun_cmds
New in version 3.0.

List of shell commands to execute before launching this job.

These commands do not execute in the context of ReFrame. Instead, they are emitted in the generated job script just before the actual job launch command.

Type `List[str]`

Default `[]`

readonly_files
List of files or directories (relative to the *sourcesdir*) that will be symlinked in the stage directory and not copied.

You can use this variable to avoid copying very large files to the stage directory.

Type `List[str]`

Default `[]`

reference
The set of reference values for this test.

The reference values are specified as a scoped dictionary keyed on the performance variables defined in *perf_patterns* and scoped under the system/partition combinations. The reference itself is a four-tuple that contains the reference value, the lower and upper thresholds and the measurement unit.

An example follows:

```
self.reference = {
    'sys0:part0': {
        'perfvar0': (50, -0.1, 0.1, 'Gflop/s'),
        'perfvar1': (20, -0.1, 0.1, 'GB/s')
    },
    'sys0:part1': {
        'perfvar0': (100, -0.1, 0.1, 'Gflop/s'),
        'perfvar1': (40, -0.1, 0.1, 'GB/s')
    }
}
```

Type A scoped dictionary with system names as scopes or `None`

Default `{}`

Note: Changed in version 3.0: The measurement unit is required. The user should explicitly specify `None` if no unit is available.

---

run()

The run phase of the regression test pipeline.

This call is non-blocking. It simply submits the job associated with this test and returns.

Warning: Changed in version 3.0: You may not override this method directly unless you are in special test. See here for more details.

sanity_patterns

Refer to the *ReFrame Tutorials* for concrete usage examples.

If set to `None`, a sanity error will be raised during sanity checking.

Type A deferrable expression (i.e., the result of a *sanity function*) or `None`

Default `None`

Note: Changed in version 2.9: The default behaviour has changed and it is now considered a sanity failure if this attribute is set to `None`.

If a test doesn't care about its output, this must be stated explicitly as follows:

```
self.sanity_patterns = sn.assert_found(r'.*', self.stdout)
```

---

setup(*partition*, *environ*, ***job_opts*)

The setup phase of the regression test pipeline.

Parameters

- **partition** – The system partition to set up this test for.
- **environ** – The environment to set up this test for.

- **job_opts** – Options to be passed through to the backend scheduler. When overriding this method users should always pass through `job_opts` to the base class method.

Raises **reframe.core.exceptions.ReframeError** – In case of errors.

> **Warning:** Changed in version 3.0: You may not override this method directly unless you are in special test. See here for more details.

sourcepath

The path to the source file or source directory of the test.

It must be a path relative to the *sourcesdir*, pointing to a subfolder or a file contained in *sourcesdir*. This applies also in the case where *sourcesdir* is a Git repository.

If it refers to a regular file, this file will be compiled using the *SingleSource* build system. If it refers to a directory, ReFrame will try to infer the build system to use for the project and will fall back in using the *Make* build system, if it cannot find a more specific one.

Type `str`

Default `''`

sourcesdir

The directory containing the test's resources.

This directory may be specified with an absolute path or with a path relative to the location of the test. Its contents will always be copied to the stage directory of the test.

This attribute may also accept a URL, in which case ReFrame will treat it as a Git repository and will try to clone its contents in the stage directory of the test.

If set to `None`, the test has no resources an no action is taken.

Type `str` or `None`

Default `'src'` if such a directory exists at the test level, otherwise `None`

Note: Changed in version 2.9: Allow `None` values to be set also in regression tests with a compilation phase

Changed in version 2.10: Support for Git repositories was added.

Changed in version 3.0: Default value is now conditionally set to either `'src'` or `None`.

---

property stagedir

The stage directory of the test.

This is set during the *setup()* phase.

Type `str`.

property stderr

The name of the file containing the standard error of the test.

This is set during the *setup()* phase.

This attribute is evaluated lazily, so it can by used inside sanity expressions.

Type `str`.

property stdout
The name of the file containing the standard output of the test.

This is set during the *setup()* phase.

This attribute is evaluated lazily, so it can by used inside sanity expressions.

Type str.

strict_check
Mark this test as a strict performance test.

If a test is marked as non-strict, the performance checking phase will always succeed, unless the `--strict` command-line option is passed when invoking ReFrame.

Type boolean

Default `True`

tags
Set of tags associated with this test.

This test can be selected from the frontend using any of these tags.

Type `Set[str]`

Default an empty set

time_limit
Time limit for this test.

Time limit is specified as a string in the form `<days>d<hours>h<minutes>m<seconds>s`. If set to `None`, no time limit will be set. The default time limit of the system partition's scheduler will be used.

The value is internaly kept as a `datetime.timedelta` object. For example '2h30m' is represented as `datetime.timedelta(hours=2, minutes=30)`

Type str or `datetime.timedelta`

Default `'10m'`

Note: Changed in version 2.15: This attribute may be set to `None`.

Warning: Changed in version 3.0: The old syntax using a `(h, m, s)` tuple is deprecated.

use_multithreading
Specify whether this tests needs simultaneous multithreading enabled.

Ignored if `None`.

Type boolean or `None`

Default `None`

valid_prog_environs
List of programming environments supported by this test.

If $\star$ is in the list then all programming environments are supported by this test.

Type `List[str]`

Default `[]`

Note: Changed in version 2.12: Programming environments can now be specified using wildcards.

Changed in version 2.17: Support for wildcards is dropped.

valid_systems
List of systems supported by this test. The general syntax for systems is `<sysname>[:<partname>]`. Both <sysname> and <partname> accept the value `*` to mean any value. `*` is an alias of `*:*`

Type `List[str]`

Default `[]`

variables
Environment variables to be set before running this test.

These variables will be set during the *setup()* phase.

Type `Dict[str, str]`

Default `{}`

wait()
Wait for this test to finish.

Raises **reframe.core.exceptions.ReframeError** – In case of errors.

Warning: Changed in version 3.0: You may not override this method directly unless you are in special test. See here for more details.

class reframe.core.pipeline.**RunOnlyRegressionTest**
Bases: *reframe.core.pipeline.RegressionTest*

Base class for run-only regression tests.

This class is also directly available under the top-level `reframe` module.

compile()
The compilation phase of the regression test pipeline.

This is a no-op for this type of test.

compile_wait()
Wait for compilation phase to finish.

This is a no-op for this type of test.

run()
The run phase of the regression test pipeline.

The resources of the test are copied to the stage directory and the rest of execution is delegated to the *RegressionTest.run()*.

reframe.core.pipeline.**DEPEND_BY_ENV = 2**
Constant to be passed as the `how` argument of the *RegressionTest.depends_on()* method. It denotes that the test cases of the current test will depend only on the corresponding test cases of the target test that use the same programming environment.

This constant is directly available under the `reframe` module.

reframe.core.pipeline.**DEPEND_EXACT = 1**
Constant to be passed as the how argument of the *RegressionTest.depends_on()* method. It denotes that test case dependencies will be explicitly specified by the user.

This constant is directly available under the reframe module.

reframe.core.pipeline.**DEPEND_FULLY = 3**
Constant to be passed as the how argument of the *RegressionTest.depends_on()* method. It denotes that each test case of this test depends on all the test cases of the target test.

This constant is directly available under the reframe module.

Regression Test Class Decorators

@reframe.core.decorators.**parameterized_test**(**inst*)
Class decorator for registering multiple instantiations of a test class.

The decorated class must derive from *reframe.core.pipeline.RegressionTest*. This decorator is also available directly under the reframe module.

Parameters **inst** – The different instantiations of the test. Each instantiation argument may be either a sequence or a mapping.

New in version 2.13.

Note: This decorator does not instantiate any test. It only registers them. The actual instantiation happens during the loading phase of the test.

@reframe.core.decorators.**required_version**(**versions*)
Class decorator for specifying the required ReFrame versions for the following test.

If the test is not compatible with the current ReFrame version it will be skipped.

New in version 2.13.

Parameters **versions** – A list of ReFrame version specifications that this test is allowed to run. A version specification string can have one of the following formats:

1. VERSION: Specifies a single version.
2. {OP}VERSION, where {OP} can be any of >, >=, <, <=, == and !=. For example, the version specification string '>=2.15' will allow the following test to be loaded only by ReFrame 2.15 and higher. The ==VERSION specification is the equivalent of VERSION.
3. V1..V2: Specifies a range of versions.

You can specify multiple versions with this decorator, such as @required_version('2.13', '>=2.16'), in which case the test will be selected if *any* of the versions is satisfied, even if the versions specifications are conflicting.

@reframe.core.decorators.**simple_test**
Class decorator for registering parameterless tests with ReFrame.

The decorated class must derive from *reframe.core.pipeline.RegressionTest*. This decorator is also available directly under the reframe module.

New in version 2.13.

Pipeline Hooks

`@reframe.core.decorators.`**`run_after`**(*stage*)
Decorator for attaching a test method to a pipeline stage.

This is completely analogous to the *reframe.core.decorators.run_before*.

New in version 2.20.

`@reframe.core.decorators.`**`run_before`**(*stage*)
Decorator for attaching a test method to a pipeline stage.

The method will run just before the specified pipeline stage and it should not accept any arguments except `self`.

This decorator can be stacked, in which case the function will be attached to multiple pipeline stages.

The `stage` argument can be any of `'setup'`, `'compile'`, `'run'`, `'sanity'`, `'performance'` or `'cleanup'`.

New in version 2.20.

`@reframe.core.decorators.`**`require_deps`**
Denote that the decorated test method will use the test dependencies.

The arguments of the decorated function must be named after the dependencies that the function intends to use. The decorator will bind the arguments to a partial realization of the *reframe.core.pipeline.RegressionTest.getdep()* function, such that conceptually the new function arguments will be the following:

```
new_arg = functools.partial(getdep, orig_arg_name)
```

The converted arguments are essentially functions accepting a single argument, which is the target test's programming environment.

This decorator is also directly available under the `reframe` module.

New in version 2.21.

Environments and Systems

`class` `reframe.core.environments.`**`Environment`**(*name*, *modules=None*, *variables=None*)
Bases: `object`

This class abstracts away an environment to run regression tests.

It is simply a collection of modules to be loaded and environment variables to be set when this environment is loaded by the framework.

> **Warning:** Users may not create *Environment* objects directly.

`property modules`
The modules associated with this environment.

Type `List[str]`

`property name`
The name of this environment.

Type `str`

property variables
The environment variables associated with this environment.

Type OrderedDict[str, str]

class reframe.core.environments.**ProgEnvironment**(*name*, *modules=None*, *variables=None*, *cc='cc'*, *cxx='CC'*, *ftn='ftn'*, *nvcc='nvcc'*, *cppflags=None*, *cflags=None*, *cxxflags=None*, *fflags=None*, *ldflags=None*, ***kwargs*)

Bases: *reframe.core.environments.Environment*

A class representing a programming environment.

This type of environment adds also properties for retrieving the compiler and compilation flags.

Warning: Users may not create *ProgEnvironment* objects directly.

property cc
The C compiler of this programming environment.

Type str

property cflags
The C compiler flags of this programming environment.

Type List[str]

property cppflags
The preprocessor flags of this programming environment.

Type List[str]

property cxx
The C++ compiler of this programming environment.

Type str

property cxxflags
The C++ compiler flags of this programming environment.

Type List[str]

property fflags
The Fortran compiler flags of this programming environment.

Type List[str]

property ftn
The Fortran compiler of this programming environment.

Type str

property ldflags
The linker flags of this programming environment.

Type List[str]

class reframe.core.environments.**_EnvironmentSnapshot**(*name='env_snapshot'*)

Bases: *reframe.core.environments.Environment*

An environment snapshot.

restore()
Restore this environment snapshot.

reframe.core.environments.**snapshot**()
Create an environment snapshot

Returns An instance of *_EnvironmentSnapshot*.

class reframe.core.systems.**System**(*name*, *descr*, *hostnames*, *modules_system*, *preload_env*, *prefix*, *outputdir*, *resourcesdir*, *stagedir*, *partitions*)
Bases: object

A representation of a system inside ReFrame.

Warning: Users may not create *System* objects directly.

property descr
The description of this system.

Type str

property hostnames
The hostname patterns associated with this system.

Type List[str]

json()
Return a JSON object representing this system.

property modules_system
The modules system name associated with this system.

Type *reframe.core.modules.ModulesSystem*

property name
The name of this system.

Type str

property outputdir
The ReFrame output directory prefix associated with this system.

Type str

property partitions
The system partitions associated with this system.

Type List[SystemPartition]

property prefix
The ReFrame prefix associated with this system.

Type str

property preload_environ
The environment to load whenever ReFrame runs on this system.

New in version 2.19.

Type *reframe.core.environments.Environment*

property resourcesdir
Global resources directory for this system.

This directory may be used for storing large files related to regression tests. The value of this directory is controlled by the resourcesdir configuration parameter.

Type str

property stagedir
The ReFrame stage directory prefix associated with this system.

Type str

class reframe.core.systems.**SystemPartition**(*parent*, *name*, *scheduler*, *launcher*, *descr*, *access*, *container_environs*, *resources*, *local_env*, *environs*, *max_jobs*)

Bases: object

A representation of a system partition inside ReFrame.

Warning: Users may not create *SystemPartition* objects directly.

property access
The scheduler options for accessing this system partition.

Type List[str]

property container_environs
Environments associated with the different container platforms.

Type Dict[str, Environment]

property descr
The description of this partition.

Type str

environment(*name*)
Return the partition environment named name.

property environs
The programming environments associated with this system partition.

Type List[ProgEnvironment]

property fullname
Return the fully-qualified name of this partition.

The fully-qualified name is of the form <parent-system-name>:<partition-name>.

Type str

json()
Return a JSON object representing this system partition.

property launcher
The type of the backend launcher of this partition.

New in version 2.8.

Returns a subclass of *reframe.core.launchers.JobLauncher*.

property local_env
The local environment associated with this partition.

Type Environment

property max_jobs
The maximum number of concurrent jobs allowed on this partition.

Type integral

property name
The name of this partition.

Type str

property resources
The resources template strings associated with this partition.

This is a dictionary, where the key is the name of a resource and the value is the scheduler options or directives associated with this resource.

Type Dict[str, List[str]]

property scheduler
The type of the backend scheduler of this partition.

Returns a subclass of *reframe.core.schedulers.JobScheduler*.

Note: Changed in version 2.8: Prior versions returned a string representing the scheduler and job launcher combination.

Job Schedulers and Parallel Launchers

class reframe.core.schedulers.**Job**(*name*, *workdir='.'*, *script_filename=None*, *stdout=None*, *stderr=None*, *max_pending_time=None*, *sched_flex_alloc_nodes=None*, *sched_access=[]*, *sched_account=None*, *sched_partition=None*, *sched_reservation=None*, *sched_nodelist=None*, *sched_exclude_nodelist=None*, *sched_exclusive_access=None*, *sched_options=None*)

Bases: object

A job descriptor.

A job descriptor is created by the framework after the "setup" phase and is associated with the test.

Warning: Users may not create a job descriptor directly.

exitcode
New in version 2.21.

The exit code of the job.

This may or may not be set depending on the scheduler backend.

Type int or None.

jobid
New in version 2.21.

The ID of the current job.

Type int or None.

launcher
The (parallel) program launcher that will be used to launch the (parallel) executable of this job.

Users are allowed to explicitly set the current job launcher, but this is only relevant in rare situations, such as when you want to wrap the current launcher command. For this specific scenario, you may have a look at the *reframe.core.launchers.LauncherWrapper* class.

The following example shows how you can replace the current partition's launcher for this test with the "local" launcher:

```
from reframe.core.backends import getlauncher

@rfm.run_after('setup')
def set_launcher(self):
    self.job.launcher = getlauncher('local')()
```

Type *reframe.core.launchers.JobLauncher*

nodelist
New in version 2.17.

The list of node names assigned to this job.

This attribute is `None` if no nodes are assigned to the job yet. This attribute is set reliably only for the `slurm` backend, i.e., Slurm *with* accounting enabled. The `squeue` scheduler backend, i.e., Slurm *without* accounting, might not set this attribute for jobs that finish very quickly. For the `local` scheduler backend, this returns an one-element list containing the hostname of the current host.

This attribute might be useful in a flexible regression test for determining the actual nodes that were assigned to the test. For more information on flexible node allocation, see the `--flex-alloc-nodes` command-line option

This attribute is *not* supported by the `pbs` scheduler backend.

options
Options to be passed to the backend job scheduler.

Type `List[str]`

Default `[]`

state
New in version 2.21.

The state of the job.

The value of this field is scheduler-specific.

Type `str` or `None`.

class `reframe.core.schedulers.`**JobScheduler**
Bases: `abc.ABC`

Abstract base class for job scheduler backends.

class `reframe.core.launchers.`**JobLauncher**
Bases: `abc.ABC`

Abstract base class for job launchers.

A job launcher is the executable that actually launches a distributed program to multiple nodes, e.g., `mpirun`, `srun` etc.

Warning: Users may not create job launchers directly.

Note: Changed in version 2.8: Job launchers do not get a reference to a job during their initialization.

options
List of options to be passed to the job launcher invocation.

Type `List[str]`

Default `[]`

class reframe.core.launchers.**LauncherWrapper**(*target_launcher*, *wrapper_command*, *wrapper_options=[]*)

Bases: *reframe.core.launchers.JobLauncher*

Wrap a launcher object so as to modify its invocation.

This is useful for parallel debuggers. For example, to launch a regression test using the ARM DDT debugger, you can do the following:

```
@rfm.run_after('setup')
def set_launcher(self):
    self.job.launcher = LauncherWrapper(self.job.launcher, 'ddt',
                                        ['--offline'])
```

If the current system partition uses native Slurm for job submission, this setup will generate the following command in the submission script:

```
ddt --offline srun <test_executable>
```

If the current partition uses `mpirun` instead, it will generate

```
ddt --offline mpirun -np <num_tasks> ... <test_executable>
```

Parameters

- **target_launcher** – The launcher to wrap.
- **wrapper_command** – The wrapper command.
- **wrapper_options** – List of options to pass to the wrapper command.

reframe.core.backends.**getlauncher**(*name*)
Retrieve the *reframe.core.launchers.JobLauncher* concrete implementation for a parallel launcher backend.

Parameters **name** – The registered name of the launcher backend.

reframe.core.backends.**getscheduler**(*name*)
Retrieve the *reframe.core.schedulers.JobScheduler* concrete implementation for a scheduler backend.

Parameters **name** – The registered name of the scheduler backend.

Runtime Services

class reframe.core.runtime.**RuntimeContext**(*site_config*)
Bases: object

The runtime context of the framework.

There is a single instance of this class globally in the framework.

New in version 2.13.

get_option(*option*)
Get a configuration option.

Parameters **option** – The option to be retrieved.

Returns The value of the option.

property modules_system
The environment modules system used in the current host.

Type *reframe.core.modules.ModulesSystem*.

property output_prefix
The output directory prefix.

Type str

property stage_prefix
The stage directory prefix.

Type str

property system
The current host system.

Type *reframe.core.systems.System*

reframe.core.runtime.**is_env_loaded**(*environ*)
Check if environment is loaded.

Parameters **environ** (Environment) – Environment to check for.

Returns True if this environment is loaded, False otherwise.

reframe.core.runtime.**loadenv**(**environs*)
Load environments in the current Python context.

Parameters **environs** (*List[Environment]*) – A list of environments to load.

Returns A tuple containing snapshot of the current environment upon entry to this function and a list of shell commands required to load the environments.

Return type Tuple[*_EnvironmentSnapshot*, List[str]]

class reframe.core.runtime.**module_use**(**paths*)
Bases: object

Context manager for temporarily modifying the module path.

reframe.core.runtime.**runtime**()
Get the runtime context of the framework.

New in version 2.13.

Returns A *reframe.core.runtime.RuntimeContext* object.

class reframe.core.runtime.**temp_environment**(*modules=[]*, *variables=[]*)
Bases: object

Context manager to temporarily change the environment.

Modules Systems

class reframe.core.modules.**ModulesSystem**(*backend*)
Bases: object

A modules system.

conflicted_modules(*name*)
Return the list of the modules conflicting with module name.

If module name resolves to multiple real modules, then the returned list will be the concatenation of the conflict lists of all the real modules.

Return type List[str]

emit_load_commands(*name*)
Return the appropriate shell command for loading module name.

Return type List[str]

emit_unload_commands(*name*)
Return the appropriate shell command for unloading module name.

Return type List[str]

is_module_loaded(*name*)
Check if module name is loaded.

If module name refers to multiple real modules, this method will return True only if all the referees are loaded.

load_module(*name*, *force=False*)
Load the module name.

If force is set, forces the loading, unloading first any conflicting modules currently loaded. If module name refers to multiple real modules, all of the target modules will be loaded.

Returns the list of unloaded modules as strings.

Return type List[str]

loaded_modules()
Return a list of loaded modules.

Return type List[str]

property name
The name of this module system.

property searchpath
The module system search path as a list of directories.

searchpath_add(**dirs*)
Add dirs to the module system search path.

searchpath_remove(**dirs*)
Remove dirs from the module system search path.

unload_all()
Unload all loaded modules.

unload_module(*name*)
Unload module `name`.

If module `name` refers to multiple real modules, all the referred to modules will be unloaded in reverse order.

property version
The version of this module system.

Build Systems

New in version 2.14.

ReFrame delegates the compilation of the regression test to a *build system*. Build systems in ReFrame are entities that are responsible for generating the necessary shell commands for compiling a code. Each build system defines a set of attributes that users may set in order to customize their compilation. An example usage is the following:

```
self.build_system = 'SingleSource'
self.build_system.cflags = ['-fopenmp']
```

Users simply set the build system to use in their regression tests and then they configure it. If no special configuration is needed for the compilation, users may completely ignore the build systems. ReFrame will automatically pick one based on the regression test attributes and will try to compile the code.

All build systems in ReFrame derive from the abstract base class *reframe.core.buildsystems.BuildSystem*. This class defines a set of common attributes, such us compilers, compilation flags etc. that all subclasses inherit. It is up to the concrete build system implementations on how to use or not these attributes.

class reframe.core.buildsystems.**Autotools**
Bases: *reframe.core.buildsystems.ConfigureBasedBuildSystem*

A build system for compiling Autotools-based projects.

This build system will emit the following commands:

1. Create a build directory if `builddir` is not `None` and change to it.
2. Invoke `configure` to configure the project by setting the corresponding flags for compilers and compiler flags.
3. Issue `make` to compile the code.

emit_build_commands(*environ*)
Return the list of commands for building using this build system.

The build commands may always assume to be issued from the top-level directory of the code that is to be built.

Parameters **environ** (*reframe.core.environments.ProgEnvironment*) – The programming environment for which to emit the build instructions. The framework passes here the current programming environment.

Raises `BuildSystemError` in case of errors when generating the build instructions.

Note: This method is relevant only to developers of new build systems.

class reframe.core.buildsystems.**BuildSystem**

Bases: abc.ABC

The abstract base class of any build system.

Concrete build systems inherit from this class and must override the *emit_build_commands()* abstract function.

cc

The C compiler to be used. If empty and *flags_from_environ* is `True`, the compiler defined in the current programming environment will be used.

Type str

Default ''

cflags

The C compiler flags to be used. If empty and *flags_from_environ* is `True`, the corresponding flags defined in the current programming environment will be used.

Type List[str]

Default []

cppflags

The preprocessor flags to be used. If empty and *flags_from_environ* is `True`, the corresponding flags defined in the current programming environment will be used.

Type List[str]

Default []

cxx

The C++ compiler to be used. If empty and *flags_from_environ* is `True`, the compiler defined in the current programming environment will be used.

Type str

Default ''

cxxflags

The C++ compiler flags to be used. If empty and *flags_from_environ* is `True`, the corresponding flags defined in the current programming environment will be used.

Type List[str]

Default []

abstract emit_build_commands(*environ*)

Return the list of commands for building using this build system.

The build commands may always assume to be issued from the top-level directory of the code that is to be built.

Parameters **environ** (*reframe.core.environments.ProgEnvironment*) – The programming environment for which to emit the build instructions. The framework passes here the current programming environment.

Raises BuildSystemError in case of errors when generating the build instructions.

Note: This method is relevant only to developers of new build systems.

---

fflags
The Fortran compiler flags to be used. If empty and *flags_from_environ* is `True`, the corresponding flags defined in the current programming environment will be used.

Type `List[str]`

Default `[]`

flags_from_environ
Set compiler and compiler flags from the current programming environment if not specified otherwise.

Type `bool`

Default `True`

ftn
The Fortran compiler to be used. If empty and *flags_from_environ* is `True`, the compiler defined in the current programming environment will be used.

Type `str`

Default `''`

ldflags
The linker flags to be used. If empty and *flags_from_environ* is `True`, the corresponding flags defined in the current programming environment will be used.

Type `List[str]`

Default `[]`

nvcc
The CUDA compiler to be used. If empty and *flags_from_environ* is `True`, the compiler defined in the current programming environment will be used.

Type `str`

Default `''`

class `reframe.core.buildsystems.`**CMake**
Bases: *reframe.core.buildsystems.ConfigureBasedBuildSystem*

A build system for compiling CMake-based projects.

This build system will emit the following commands:

1. Create a build directory if `builddir` is not `None` and change to it.
2. Invoke `cmake` to configure the project by setting the corresponding CMake flags for compilers and compiler flags.
3. Issue `make` to compile the code.

emit_build_commands(*environ*)
Return the list of commands for building using this build system.

The build commands may always assume to be issued from the top-level directory of the code that is to be built.

Parameters **environ** (*reframe.core.environments.ProgEnvironment*) – The programming environment for which to emit the build instructions. The framework passes here the current programming environment.

Raises `BuildSystemError` in case of errors when generating the build instructions.

Note: This method is relevant only to developers of new build systems.

class reframe.core.buildsystems.**ConfigureBasedBuildSystem**

Bases: *reframe.core.buildsystems.BuildSystem*

Abstract base class for configured-based build systems.

builddir

The CMake build directory, where all the generated files will be placed.

Type str

Default None

config_opts

Additional configuration options to be passed to the CMake invocation.

Type List[str]

Default []

make_opts

Options to be passed to the subsequent make invocation.

Type List[str]

Default []

max_concurrency

Same as for the *Make* build system.

Type integer

Default 1

srcdir

The top-level directory of the code.

This is set automatically by the framework based on the *reframe.core.pipeline.RegressionTest.sourcepath* attribute.

Type str

Default None

class reframe.core.buildsystems.**Make**

Bases: *reframe.core.buildsystems.BuildSystem*

A build system for compiling codes using make.

The generated build command has the following form:

```
make -j [N] [-f MAKEFILE] [-C SRCDIR] CC="X" CXX="X" FC="X" NVCC="X" CPPFLAGS="X"␣
↪CFLAGS="X" CXXFLAGS="X" FCFLAGS="X" LDFLAGS="X" OPTIONS
```

The compiler and compiler flags variables will only be passed if they are not None. Their value is determined by the corresponding attributes of *BuildSystem*. If you want to completely disable passing these variables to the make invocation, you should make sure not to set any of the correspoding attributes and set also the *BuildSystem.flags_from_environ* flag to False.

emit_build_commands(*environ*)

Return the list of commands for building using this build system.

The build commands may always assume to be issued from the top-level directory of the code that is to be built.

Parameters **environ** (*reframe.core.environments.ProgEnvironment*) – The programming environment for which to emit the build instructions. The framework passes here the current programming environment.

Raises `BuildSystemError` in case of errors when generating the build instructions.

Note: This method is relevant only to developers of new build systems.

---

makefile
Instruct build system to use this Makefile. This option is useful when having non-standard Makefile names.

Type `str`

Default `None`

max_concurrency
Limit concurrency for `make` jobs. This attribute controls the `-j` option passed to `make`. If not `None`, `make` will be invoked as `make -j max_concurrency`. Otherwise, it will invoked as `make -j`.

Type integer

Default `1`

Note: Changed in version 2.19: The default value is now `1`

---

options
Append these options to the `make` invocation. This variable is also useful for passing variables or targets to `make`.

Type `List[str]`

Default `[]`

srcdir
The top-level directory of the code.

This is set automatically by the framework based on the *reframe.core.pipeline.RegressionTest.sourcepath* attribute.

Type `str`

Default `None`

class reframe.core.buildsystems.**SingleSource**
Bases: *reframe.core.buildsystems.BuildSystem*

A build system for compiling a single source file.

The generated build command will have the following form:

```
COMP CPPFLAGS XFLAGS SRCFILE -o EXEC LDFLAGS
```

- `COMP` is the required compiler for compiling `SRCFILE`. This build system will automatically detect the programming language of the source file and pick the correct compiler. See also the *SingleSource.lang* attribute.
- `CPPFLAGS` are the preprocessor flags and are passed to any compiler.

- `XFLAGS` is any of `CFLAGS`, `CXXFLAGS` or `FCFLAGS` depending on the programming language of the source file.
- `SRCFILE` is the source file to be compiled. This is set up automatically by the framework. See also the *`SingleSource.srcfile`* attribute.
- `EXEC` is the executable to be generated. This is also set automatically by the framework. See also the *`SingleSource.executable`* attribute.
- `LDFLAGS` are the linker flags.

For CUDA codes, the language assumed is C++ (for the compilation flags) and the compiler used is *`BuildSystem.nvcc`*.

`emit_build_commands`(*environ*)
Return the list of commands for building using this build system.

The build commands may always assume to be issued from the top-level directory of the code that is to be built.

Parameters **`environ`** (*`reframe.core.environments.ProgEnvironment`*) – The programming environment for which to emit the build instructions. The framework passes here the current programming environment.

Raises `BuildSystemError` in case of errors when generating the build instructions.

Note: This method is relevant only to developers of new build systems.

`executable`
The executable file to be generated.

This is set automatically by the framework based on the *`reframe.core.pipeline.RegressionTest.executable`* attribute.

Type `str` or `None`

`include_path`
The include path to be used for this compilation.

All the elements of this list will be appended to the *`BuildSystem.cppflags`*, by prepending to each of them the `-I` option.

Type `List[str]`

Default `[]`

`lang`
The programming language of the file that needs to be compiled. If not specified, the build system will try to figure it out automatically based on the extension of the source file. The automatically detected extensions are the following:

- C: *.c* and *.upc*.
- C++: *.cc*, *.cp*, *.cxx*, *.cpp*, *.CPP*, *.c++* and *.C*.
- Fortran: *.f*, *.for*, *.ftn*, *.F*, *.FOR*, *.fpp*, *.FPP*, *.FTN*, *.f90*, *.f95*, *.f03*, *.f08*, *.F90*, *.F95*, *.F03* and *.F08*.
- CUDA: *.cu*.

Type `str` or `None`

srcfile
The source file to compile. This is automatically set by the framework based on the *reframe.core.pipeline.RegressionTest.sourcepath* attribute.

Type str or None

Container Platforms

New in version 2.20.

class reframe.core.containers.**ContainerPlatform**
Bases: abc.ABC

The abstract base class of any container platform.

commands
The commands to be executed within the container.

Type list[str]

Default []

image
The container image to be used for running the test.

Type str or None

Default None

mount_points
List of mount point pairs for directories to mount inside the container.

Each mount point is specified as a tuple of (/path/in/host, /path/in/container).

Type list[tuple[str, str]]

Default []

options
Additional options to be passed to the container runtime when executed.

Type list[str]

Default []

workdir
The working directory of ReFrame inside the container.

This is the directory where the test's stage directory is mounted inside the container. This directory is always mounted regardless if *mount_points* is set or not.

Type str

Default /rfm_workdir

class reframe.core.containers.**Docker**
Bases: *reframe.core.containers.ContainerPlatform*

Container platform backend for running containers with Docker.

class reframe.core.containers.**Sarus**
Bases: *reframe.core.containers.ContainerPlatform*

Container platform backend for running containers with Sarus.

with_mpi
Enable MPI support when launching the container.

Type boolean

Default `False`

class reframe.core.containers.**Shifter**
Bases: *reframe.core.containers.Sarus*

Container platform backend for running containers with Shifter.

class reframe.core.containers.**Singularity**
Bases: *reframe.core.containers.ContainerPlatform*

Container platform backend for running containers with Singularity.

with_cuda
Enable CUDA support when launching the container.

Type boolean

Default `False`

The `reframe` module

The `reframe` module offers direct access to the basic test classes, constants and decorators.

class reframe.**CompileOnlyRegressionTest**
See *reframe.core.pipeline.CompileOnlyRegressionTest*.

class reframe.**RegressionTest**
See *reframe.core.pipeline.RegressionTest*.

class reframe.**RunOnlyRegressionTest**
See *reframe.core.pipeline.RunOnlyRegressionTest*.

reframe.**DEPEND_BY_ENV**
See *reframe.core.pipeline.DEPEND_BY_ENV*.

reframe.**DEPEND_EXACT**
See *reframe.core.pipeline.DEPEND_EXACT*.

reframe.**DEPEND_FULLY**
See *reframe.core.pipeline.DEPEND_FULLY*.

@reframe.**parameterized_test**
See *@reframe.core.decorators.parameterized_test*.

@reframe.**require_deps**
See *@reframe.core.decorators.require_deps*.

@reframe.**required_version**
See *@reframe.core.decorators.required_version*.

@reframe.**run_after**
See *@reframe.core.decorators.run_after*.

@reframe.**run_before**
See *@reframe.core.decorators.run_before*.

@reframe.**simple_test**
See *@reframe.core.decorators.simple_test*.

2.7.4 Sanity Functions Reference

Sanity functions are the functions used with the *sanity_patterns* and *perf_patterns*. The key characteristic of these functions is that they are not executed the time they are called. Instead they are evaluated at a later point by the framework (inside the *check_sanity* and *check_performance* methods). Any sanity function may be evaluated either explicitly or implicitly.

Explicit evaluation of sanity functions

Sanity functions may be evaluated at any time by calling `evaluate()` on their return value or by passing the result of a sanity function to the *reframe.utility.sanity.evaluate()* free function.

Implicit evaluation of sanity functions

Sanity functions may also be evaluated implicitly in the following situations:

- When you try to get their truthy value by either explicitly or implicitly calling `bool` on their return value. This implies that when you include the result of a sanity function in an `if` statement or when you apply the `and`, `or` or `not` operators, this will trigger their immediate evaluation.
- When you try to iterate over their result. This implies that including the result of a sanity function in a `for` statement will trigger its evaluation immediately.
- When you try to explicitly or implicitly get its string representation by calling `str` on its result. This implies that printing the return value of a sanity function will automatically trigger its evaluation.

Categories of sanity functions

Currently ReFrame provides three broad categories of sanity functions:

1. Deferrable replacements of certain Python built-in functions. These functions simply delegate their execution to the actual built-ins.
2. Assertion functions. These functions are used to assert certain conditions and they either return `True` or raise `reframe.core.exceptions.SanityError` with a message describing the error. Users may provide their own formatted messages through the `msg` argument. For example, in the following call to `assert_eq()` the `{0}` and `{1}` placeholders will obtain the actual arguments passed to the assertion function.

   ```
   assert_eq(a, 1, msg="{0} is not equal to {1}")
   ```

   If in the user provided message more placeholders are used than the arguments of the assert function (except the `msg` argument), no argument substitution will be performed in the user message.
3. Utility functions. The are functions that you will normally use when defining *sanity_patterns* and *perf_patterns*. They include, but are not limited to, functions to iterate over regex matches in a file, extracting and converting values from regex matches, computing statistical information on series of data etc.

Users can write their own sanity functions as well. The page "*Understanding the Mechanism of Sanity Functions*" explains in detail how sanity functions work and how users can write their own.

@sanity_function
Sanity function decorator.

The evaluation of the decorated function will be deferred and it will become suitable for use in the sanity and performance patterns of a regression test.

```
@sanity_function
def myfunc(*args):
    do_sth()
```

reframe.utility.sanity.**abs**(*x*)
Replacement for the built-in `abs()` function.

reframe.utility.sanity.**all**(*iterable*)
Replacement for the built-in `all()` function.

reframe.utility.sanity.**allx**(*iterable*)
Same as the built-in `all()` function, except that it returns `False` if `iterable` is empty.

New in version 2.13.

reframe.utility.sanity.**and_**(*a*, *b*)
Deferrable version of the `and` operator.

Returns `a and b`.

reframe.utility.sanity.**any**(*iterable*)
Replacement for the built-in `any()` function.

reframe.utility.sanity.**assert_bounded**(*val*, *lower=None*, *upper=None*, *msg=None*)
Assert that `lower <= val <= upper`.

Parameters

- **val** – The value to check.
- **lower** – The lower bound. If `None`, it defaults to `-inf`.
- **upper** – The upper bound. If `None`, it defaults to `inf`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_eq**(*a*, *b*, *msg=None*)
Assert that `a == b`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_false**(*x*, *msg=None*)
Assert that `x` is evaluated to `False`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_found**(*patt*, *filename*, *msg=None*, *encoding='utf-8'*)
Assert that regex pattern `patt` is found in the file `filename`.

Parameters

- **patt** – The regex pattern to search. Any standard Python regular expression is accepted.
- **filename** – The name of the file to examine. Any `OSError` raised while processing the file will be propagated as a `reframe.core.exceptions.SanityError`.
- **encoding** – The name of the encoding used to decode the file.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_ge**(*a*, *b*, *msg=None*)
Assert that `a >= b`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_gt**(*a*, *b*, *msg=None*)
Assert that `a > b`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_in**(*item*, *container*, *msg=None*)
Assert that `item` is in `container`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_le**(*a*, *b*, *msg=None*)
Assert that `a <= b`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_lt**(*a*, *b*, *msg=None*)
Assert that `a < b`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_ne**(*a*, *b*, *msg=None*)
Assert that `a != b`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_not_found**(*patt*, *filename*, *msg=None*, *encoding='utf-8'*)
Assert that regex pattern `patt` is not found in the file `filename`.

This is the inverse of *assert_found()*.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_not_in**(*item*, *container*, *msg=None*)
Assert that `item` is not in `container`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_reference**(*val*, *ref*, *lower_thres=None*, *upper_thres=None*, *msg=None*)
Assert that value `val` respects the reference value `ref`.

Parameters

- **val** – The value to check.
- **ref** – The reference value.

- **lower_thres** – The lower threshold value expressed as a negative decimal fraction of the reference value. Must be in [-1, 0] for ref >= 0.0 and in [-inf, 0] for ref < 0.0. If `None`, no lower thresholds is applied.
- **upper_thres** – The upper threshold value expressed as a decimal fraction of the reference value. Must be in [0, inf] for ref >= 0.0 and in [0, 1] for ref < 0.0. If `None`, no upper thresholds is applied.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails or if the lower and upper thresholds do not have appropriate values.

reframe.utility.sanity.**assert_true**(*x*, *msg=None*)
Assert that `x` is evaluated to `True`.

Returns `True` on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**avg**(*iterable*)
Return the average of all the elements of `iterable`.

reframe.utility.sanity.**chain**(**iterables*)
Replacement for the `itertools.chain()` function.

reframe.utility.sanity.**contains**(*seq*, *key*)
Deferrable version of the `in` operator.

Returns `key in seq`.

reframe.utility.sanity.**count**(*iterable*)
Return the element count of `iterable`.

This is similar to the built-in `len()`, except that it can also handle any argument that supports iteration, including generators.

reframe.utility.sanity.**count_uniq**(*iterable*)
Return the unique element count of `iterable`.

reframe.utility.sanity.**defer**(*x*)
Defer the evaluation of variable `x`.

New in version 2.21.

reframe.utility.sanity.**enumerate**(*iterable*, *start=0*)
Replacement for the built-in `enumerate()` function.

reframe.utility.sanity.**evaluate**(*expr*)
Evaluate a deferred expression.

If `expr` is not a deferred expression, it will be returned as is.

New in version 2.21.

reframe.utility.sanity.**extractall**(*patt*, *filename*, *tag=0*, *conv=None*, *encoding='utf-8'*)
Extract all values from the capturing group `tag` of a matching regex `patt` in the file `filename`.

Parameters

- **patt** – The regex pattern to search. Any standard Python regular expression is accepted.
- **filename** – The name of the file to examine.
- **encoding** – The name of the encoding used to decode the file.

- **tag** – The regex capturing group to be extracted. Group `0` refers always to the whole match. Since the file is processed line by line, this means that group `0` returns the whole line that was matched.
- **conv** – A callable that takes a single argument and returns a new value. If provided, it will be used to convert the extracted values before returning them.

Returns A list of the extracted values from the matched regex.

Raises **reframe.core.exceptions.SanityError** – In case of errors.

reframe.utility.sanity.**extractiter**(*patt*, *filename*, *tag=0*, *conv=None*, *encoding='utf-8'*)
Get an iterator over the values extracted from the capturing group `tag` of a matching regex `patt` in the file `filename`.

This function is equivalent to *extractall()* except that it returns a generator object, instead of a list, which you can use to iterate over the extracted values.

reframe.utility.sanity.**extractsingle**(*patt*, *filename*, *tag=0*, *conv=None*, *item=0*, *encoding='utf-8'*)
Extract a single value from the capturing group `tag` of a matching regex `patt` in the file `filename`.

This function is equivalent to `extractall(patt, filename, tag, conv)[item]`, except that it raises a `SanityError` if `item` is out of bounds.

Parameters

- **patt** – as in *extractall()*.
- **filename** – as in *extractall()*.
- **encoding** – as in *extractall()*.
- **tag** – as in *extractall()*.
- **conv** – as in *extractall()*.
- **item** – the specific element to extract.

Returns The extracted value.

Raises **reframe.core.exceptions.SanityError** – In case of errors.

reframe.utility.sanity.**filter**(*function*, *iterable*)
Replacement for the built-in `filter()` function.

reframe.utility.sanity.**findall**(*patt*, *filename*, *encoding='utf-8'*)
Get all matches of regex `patt` in `filename`.

Parameters

- **patt** – The regex pattern to search. Any standard Python regular expression is accepted.
- **filename** – The name of the file to examine.
- **encoding** – The name of the encoding used to decode the file.

Returns A list of raw regex match objects.

Raises **reframe.core.exceptions.SanityError** – In case an `OSError` is raised while processing `filename`.

reframe.utility.sanity.**finditer**(*patt*, *filename*, *encoding='utf-8'*)
Get an iterator over the matches of the regex `patt` in `filename`.

This function is equivalent to *findall()* except that it returns a generator object instead of a list, which you can use to iterate over the raw matches.

reframe.utility.sanity.**getattr**(*obj*, *attr*, **args*)
Replacement for the built-in `getattr()` function.

reframe.utility.sanity.**getitem**(*container*, *item*)
Get `item` from `container`.

`container` may refer to any container that can be indexed.

Raises **reframe.core.exceptions.SanityError** – In case `item` cannot be retrieved from `container`.

reframe.utility.sanity.**glob**(*pathname*, ***, *recursive=False*)
Replacement for the `glob.glob()` function.

reframe.utility.sanity.**hasattr**(*obj*, *name*)
Replacement for the built-in `hasattr()` function.

reframe.utility.sanity.**iglob**(*pathname*, *recursive=False*)
Replacement for the `glob.iglob()` function.

reframe.utility.sanity.**len**(*s*)
Replacement for the built-in `len()` function.

reframe.utility.sanity.**map**(*function*, **iterables*)
Replacement for the built-in `map()` function.

reframe.utility.sanity.**max**(**args*)
Replacement for the built-in `max()` function.

reframe.utility.sanity.**min**(**args*)
Replacement for the built-in `min()` function.

reframe.utility.sanity.**not_**(*a*)
Deferrable version of the `not` operator.

Returns `not a`.

reframe.utility.sanity.**or_**(*a*, *b*)
Deferrable version of the `or` operator.

Returns `a or b`.

reframe.utility.sanity.**print**(**objects*, *sep=' '*, *end='\n'*, *file=None*, *flush=False*)
Replacement for the built-in `print()` function.

The only difference is that this function returns the `objects`, so that you can use it transparently inside a complex sanity expression. For example, you could write the following to print the matches returned from the *`extractall()`* function:

```
self.sanity_patterns = sn.assert_eq(
    sn.count(sn.print(sn.extract_all(...))), 10
)
```

If `file` is None, *`print()`* will print its arguments to the standard output. Unlike the builtin `print()` function, we don't bind the `file` argument to `sys.stdout` by default. This would capture `sys.stdout` at the time this function is defined and would prevent it from seeing changes to `sys.stdout`, such as redirects, in the future.

reframe.utility.sanity.**reversed**(*seq*)
Replacement for the built-in `reversed()` function.

reframe.utility.sanity.**round**(*number*, **args*)
Replacement for the built-in `round()` function.

reframe.utility.sanity.**setattr**(*obj*, *name*, *value*)
Replacement for the built-in `setattr()` function.

reframe.utility.sanity.**sorted**(*iterable*, **args*)
Replacement for the built-in `sorted()` function.

reframe.utility.sanity.**sum**(*iterable*, **args*)
Replacement for the built-in `sum()` function.

reframe.utility.sanity.**zip**(**iterables*)
Replacement for the built-in `zip()` function.

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

O

P

R

S

T

U

V

W

Z