ReFrame Documentation

Release 3.6.0

CSCS

May 04, 2021

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] @ 6th EasyBuild User Meeting 2021.
- 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.
- The required Python packages are the following:

```
argcomplete==1.12.3
coverage==5.5
importlib_metadata==4.0.1; python_version < '3.8'
jsonschema==3.2.0
lxml==4.6.3
pytest==6.2.3
pytest-forked==1.3.0
pytest-parallel==0.1.0
PyYAML==5.4.1
requests==2.25.1
semver==2.13.0
setuptools==56.0.0
wcwidth==0.2.5
#+pygelf%pygelf==0.4.0
```

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

2.1.2 Getting the Framework

Stable ReFrame releases are available through different channels.

Spack

ReFrame is available as a Spack package:

```
spack install reframe
```

There are the following variants available:

- `+docs`: This will install the man pages of ReFrame.
- `+gelf`: This will install the bindings for handling Graylog log messages.

EasyBuild

ReFrame is available as an EasyBuild package:

```
eb ReFrame-VERSION.eb -r
```

This will install the man pages as well as the Graylog bindings.

PyPI

ReFrame is available as a PyPI package:

```
pip install reframe-hpc
```

This is a bare installation of the framework. It will not install the documentation, the tutorial examples or the bindings for handling Graylog log messages.

Github

Any ReFrame version can be very easily installed directly from Github:

```
pushd /path/to/install/prefix
git clone -q --depth 1 --branch VERSION_TAG https://github.com/eth-cscs/reframe.git
pushd reframe && ./bootstrap.sh && popd
export PATH=$(pwd)/bin:$PATH
popd
```

The `VERSION_TAG` is the version number prefixed by `v`, e.g., `v3.5.0`. The `./bootstrap.sh` script will fetch ReFrame's requirements under its installation prefix. It will not set the `PYTHONPATH`, so it will not affect the user's Python installation. The `./bootstrap.sh` has two additional variant options:

- `+docs`: This will also build the documentation.
- `+pygelf`: This will install the bindings for handling Graylog log messages.

Note: New in version 3.1: The bootstrap script for ReFrame was added. For previous ReFrame versions you should install its requirements using `pip install -r requirements.txt` in a Python virtual environment.

2.1.3 Enabling auto-completion

New in version 3.4.1.

You can enable auto-completion for ReFrame by sourcing in your shell the corresponding script in `<install_prefix>/share/completions/reframe.<shell>`. Auto-completion is supported for Bash, Tcsh and Fish shells.

Note: Changed in version 3.4.2: The shell completion scripts have been moved under `share/completions/`.

2.1.4 Where to Go from Here

The easiest way to start with ReFrame is to go through *Tutorial 1: Getting Started with ReFrame*, which will guide you step-by-step in both writing your first tests and in configuring ReFrame. The *Configuring ReFrame for Your Site* page provides more details on the basic configuration aspects of ReFrame. *Advanced Topics* explain different aspects of the framework whereas *ReFrame Manuals* provide complete reference guides for the command line interface, the configuration parameters and the programming APIs for writing tests.

2.2 ReFrame Tutorials

2.2.1 Tutorial 1: Getting Started with ReFrame

New in version 3.1.

This tutorial will give you a first overview of ReFrame and will acquaint you with its basic concepts. We will start with a simple "Hello, World!" test running with the default configuration and we will expand the example along the way. We will also explore performance tests and port our tests to an HPC cluster. The examples of this tutorial can be found under `tutorials/basics/`.

Getting Ready

All you need to start off with this tutorial is to have installed ReFrame. If you haven't done so yet, all you need is Python 3.6 and above and to follow the steps below:

```
git clone https://github.com/eth-cscs/reframe.git
cd reframe
./bootstrap.sh
./bin/reframe -V
```

We're now good to go!

The "Hello, World!" test

As simple as it may sound, a series of "naive" "Hello, World!" tests can reveal lots of regressions in the programming environment of HPC clusters, but the bare minimum of those also serves perfectly the purpose of starting this tutorial. Here is its C version:

```
cat tutorials/basics/hello/src/hello.c
```

```
#include <stdio.h>


int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

And here is the ReFrame version of it:

```
cat tutorials/basics/hello/hello1.py
```

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


@rfm.simple_test
class HelloTest(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    sourcepath = 'hello.c'
    executable_opts = ['> hello.out']
    sanity_patterns = sn.assert_found(r'Hello, World\!', 'hello.out')
```

Regression tests in ReFrame are specially decorated classes that ultimately derive from *RegressionTest*. The *@simple_test* decorator registers a test class with ReFrame and makes it available to the framework. The test variables are essentially attributes of the test class and can be defined directly in the class body. Each test must always set the *valid_systems* and *valid_prog_environs* attributes. These define the systems and/or system partitions that this test is allowed to run on, as well as the programming environments that it is valid for. A programming environment is essentially a compiler toolchain. We will see later on in the tutorial how a programming environment can be defined. The generic configuration of ReFrame assumes a single programming environment named `builtin` which comprises a C compiler that can be invoked with `cc`. In this particular test we set both these attributes to `['*']`, essentially allowing this test to run everywhere.

A ReFrame test must either define an executable to execute or a source file (or source code) to be compiled. In this example, it is enough to define the source file of our hello program. ReFrame knows the executable that was produced and will use that to run the test. In this example, we redirect the executable's output into a file by defining the optional variable *executable_opts*. This output redirection is not strictly necessary and it is just done here to keep this first example as intuitive as possible.

Finally, each regression test must always define the *sanity_patterns* attribute. This is a lazily evaluated expression that asserts the sanity of the test. In this particular case, we ask ReFrame to check that the executable has produced the desired phrase into the output file `hello.out`. Note that ReFrame does not determine the success of a test by its exit code. Instead, the assessment of success is responsibility of the test itself.

Before running the test let's inspect the directory structure surrounding it:

```
tutorials/basics/hello
├── hello1.py
└── src
    └── hello.c
```

Our test is `hello1.py` and its resources, i.e., the `hello.c` source file, are located inside the `src/` subdirectory. If not specified otherwise, the *`sourcepath`* attribute is always resolved relative to `src/`. There is full flexibility in organizing the tests. Multiple tests may be defined in a single file or they may be split in multiple files. Similarly, several tests may share the same resources directory or they can simply have their own.

Now it's time to run our first test:

```
./bin/reframe -c tutorials/basics/hello/hello1.py -r
```

```
[ReFrame Setup]
  version:           3.3-dev0 (rev: 5d246bff)
  command:           './bin/reframe -c tutorials/basics/hello/hello1.py -r'
  launched by:       user@tresa.local
  working directory: '/Users/user/Repositories/reframe'
  settings file:     '<builtin>'
  check search path: '/Users/user/Repositories/reframe/tutorials/basics/hello/hello1.
↪py'
  stage directory:   '/Users/user/Repositories/reframe/stage'
  output directory:  '/Users/user/Repositories/reframe/output'

[==========] Running 1 check(s)
[==========] Started on Mon Oct 12 18:23:30 2020

[----------] started processing HelloTest (HelloTest)
[ RUN      ] HelloTest on generic:default using builtin
[----------] finished processing HelloTest (HelloTest)

[----------] waiting for spawned checks to finish
[       OK ] (1/1) HelloTest on generic:default using builtin [compile: 0.389s run: 0.
↪406s total: 0.811s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 1 test case(s) from 1 check(s) (0 failure(s))
[==========] Finished on Mon Oct 12 18:23:31 2020
Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-001f_tbi.
↪log'
```

Perfect! We have verified that we have a functioning C compiler in our system.

When ReFrame runs a test, it copies all its resources to a stage directory and performs all test-related operations (compilation, run, sanity checking etc.) from that directory. On successful outcome of the test, the stage directory is removed by default, but interesting files are copied to an output directory for archiving and later inspection. The prefixes of these directories are printed in the first section of the output. Let's inspect what files ReFrame produced for this test:

```
ls output/generic/default/builtin/HelloTest/
```

```
rfm_HelloTest_build.err rfm_HelloTest_build.sh  rfm_HelloTest_job.out
rfm_HelloTest_build.out rfm_HelloTest_job.err   rfm_HelloTest_job.sh
```

ReFrame stores in the output directory of the test the build and run scripts it generated for building and running the code along with their standard output and error. All these files are prefixed with `rfm_`.

ReFrame also generates a detailed JSON report for the whole regression testing session. By default, this is stored inside the `${HOME}/.reframe/reports` directory and a new report file is generated every time ReFrame is run, but you can control this through the *`--report-file`* command-line option.

Here are the contents of the report file for our first ReFrame run:

```
cat ~/.reframe/reports/run-report.json
```


```
{
  "session_info": {
    "cmdline": "./bin/reframe -c tutorials/basics/hello/hello1.py -r",
    "config_file": "<builtin>",
    "data_version": "1.0",
    "hostname": "dhcp-133-44.cscs.ch",
    "prefix_output": "/Users/user/Repositories/reframe/output",
    "prefix_stage": "/Users/user/Repositories/reframe/stage",
    "user": "user",
    "version": "3.1-dev2 (rev: 272e1aae)",
    "workdir": "/Users/user/Repositories/reframe",
    "time_start": "2020-07-24T11:05:46+0200",
    "time_end": "2020-07-24T11:05:47+0200",
    "time_elapsed": 0.7293069362640381,
    "num_cases": 1,
    "num_failures": 0
  },
  "runs": [
    {
      "num_cases": 1,
      "num_failures": 0,
      "runid": 0,
      "testcases": [
        {
          "build_stderr": "rfm_HelloTest_build.err",
          "build_stdout": "rfm_HelloTest_build.out",
          "description": "HelloTest",
          "environment": "builtin",
          "fail_reason": null,
          "fail_phase": null,
          "jobid": 85063,
          "job_stderr": "rfm_HelloTest_job.err",
          "job_stdout": "rfm_HelloTest_job.out",
          "name": "HelloTest",
          "maintainers": [],
          "nodelist": [
            "dhcp-133-44.cscs.ch"
          ],
          "outputdir": "/Users/user/Repositories/reframe/output/generic/default/
↪builtin/HelloTest",
          "perfvars": null,
          "result": "success",
          "stagedir": null,
          "scheduler": "local",
          "system": "generic:default",
          "tags": [],
          "time_compile": 0.3776402473449707,
          "time_performance": 4.506111145019531e-05,
          "time_run": 0.2992382049560547,
          "time_sanity": 0.0005609989166259766,
```


(continues on next page)

(continued from previous page)

```
          "time_setup": 0.0031709671020507812,
          "time_total": 0.7213571071624756
        }
      ]
    }
  ]
}
```

More of "Hello, World!"

We want to extend our test and run a C++ "Hello, World!" as well. We could simply copy paste the `hello1.py` and change the source file extension to refer to the C++ source code. But this duplication is something that we generally want to avoid. ReFrame allows you to avoid this in several ways but the most compact is to define the new test as follows:

```
cat tutorials/basics/hello/hello2.py
```

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


@rfm.simple_test
class HelloMultiLangTest(rfm.RegressionTest):
    lang = parameter(['c', 'cpp'])

    valid_systems = ['*']
    valid_prog_environs = ['*']
    executable_opts = ['> hello.out']
    sanity_patterns = sn.assert_found(r'Hello, World\!', 'hello.out')

    @rfm.run_before('compile')
    def set_sourcepath(self):
        self.sourcepath = f'hello.{self.lang}'
```

This test extends the `hello1.py` test by defining the `lang` parameter with the *`parameter()`* built-in. This parameter will cause as many instantiations as parameter values available, each one setting the `lang` attribute to one single value. Hence, this example will create two test instances, one with `lang='c'` and another with `lang='cpp'`. The parameter is available as an attribute of the test instance and, in this example, we use it to set the extension of the source file. However, at the class level, a test parameter holds all the possible values for itself, and this is only assigned a single value after the class is instantiated. Therefore, the variable `sourcepath`, which depends on this parameter, also needs to be set after the class instantiation. The simplest way to do this would be to move the `sourcepath` assignment into the `__init__()` method as shown in the code snippet below, but this has some disadvantages when writing larger tests.

```
def __init__(self):
    self.sourcepath = f'hello.{self.lang}'
```

For example, when writing a base class for a test with a large amount of code into the `__init__()` method, the derived class may want to do a partial override of the code in this function. This would force us to understand the full implementation of the base class' `__init__()` despite that we may just be interested in overriding a small part of it. Doable, but not ideal. Instead, through pipeline hooks, ReFrame provides a mechanism to attach independent functions to execute at a given time before the data they set is required by the test. This is exactly what we want to do here, and we know that the test sources are needed to compile the code. Hence, we move the `sourcepath` assignment into a pre-compile hook.

```
    @rfm.run_before('compile')
    def set_sourcepath(self):
        self.sourcepath = f'hello.{self.lang}'
```

The use of hooks is covered in more detail later on, but for now, let's just think of them as a way to defer the execution of a function to a given stage of the test's pipeline. By using hooks, any user could now derive from this class and attach other hooks (for example, adding some compiler flags) without having to worry about overriding the base method that sets the `sourcepath` variable.

Let's run the test now:

```
./bin/reframe -c tutorials/basics/hello/hello2.py -r
```

```
[ReFrame Setup]
  version:           3.6.0-dev.0+a3d0b0cd
  command:           './bin/reframe -c tutorials/basics/hello/hello2.py -r'
  launched by:       user@tresa.local
  working directory: '/Users/user/Repositories/reframe'
  settings file:     '<builtin>'
  check search path: '/Users/user/Repositories/reframe/tutorials/basics/hello/hello2.
↪py'
  stage directory:   '/Users/user/Repositories/reframe/stage'
  output directory:  '/Users/user/Repositories/reframe/output'

[==========] Running 2 check(s)
[==========] Started on Tue Mar  9 23:25:22 2021

[----------] started processing HelloMultiLangTest_c (HelloMultiLangTest_c)
[ RUN      ] HelloMultiLangTest_c on generic:default using builtin
[----------] finished processing HelloMultiLangTest_c (HelloMultiLangTest_c)

[----------] started processing HelloMultiLangTest_cpp (HelloMultiLangTest_cpp)
[ RUN      ] HelloMultiLangTest_cpp on generic:default using builtin
[     FAIL ] (1/2) HelloMultiLangTest_cpp on generic:default using builtin [compile:␣
↪0.006s run: n/a total: 0.023s]
==> test failed during 'compile': test staged in '/Users/user/Repositories/reframe/
↪stage/generic/default/builtin/HelloMultiLangTest_cpp'
[----------] finished processing HelloMultiLangTest_cpp (HelloMultiLangTest_cpp)

[----------] waiting for spawned checks to finish
[       OK ] (2/2) HelloMultiLangTest_c on generic:default using builtin [compile: 0.
↪981s run: 0.468s total: 1.475s]
[----------] all spawned checks have finished

[  FAILED  ] Ran 2/2 test case(s) from 2 check(s) (1 failure(s))
[==========] Finished on Tue Mar  9 23:25:23 2021

==============================================================================
SUMMARY OF FAILURES
------------------------------------------------------------------------------
FAILURE INFO for HelloMultiLangTest_cpp
  * Test Description: HelloMultiLangTest_cpp
  * System partition: generic:default
  * Environment: builtin
  * Stage directory: /Users/user/Repositories/reframe/stage/generic/default/builtin/
↪HelloMultiLangTest_cpp
  * Node list: None
```

(continues on next page)

(continued from previous page)

```
  * Job type: local (id=None)
  * Dependencies (conceptual): []
  * Dependencies (actual): []
  * Maintainers: []
  * Failing phase: compile
  * Rerun with '-n HelloMultiLangTest_cpp -p builtin --system generic:default -r'
  * Reason: build system error: I do not know how to compile a C++ program
----------------------------------------------------------------------------
Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-wemvsvs2.
↪log'
```

Oops! The C++ test has failed. ReFrame complains that it does not know how to compile a C++ program. Remember our discussion above that the default configuration of ReFrame defines a minimal programming environment named `builtin` which only knows of a `cc` compiler. We will fix that in a moment, but before doing that it's worth looking into the failure information provided for the test. For each failed test, ReFrame will print a short summary with information about the system partition and the programming environment that the test failed for, its job or process id (if any), the nodes it was running on, its stage directory, the phase that failed etc.

When a test fails its stage directory is kept intact, so that users can inspect the failure and try to reproduce it manually. In this case, the stage directory contains only the "Hello, World" source files, since ReFrame could not produce a build script for the C++ test, as it doesn't know to compile a C++ program for the moment.

```
ls stage/generic/default/builtin/HelloMultiLangTest_cpp
```

```
hello.c   hello.cpp
```

Let's go on and fix this failure by defining a new system and programming environments for the machine we are running on. We start off by copying the generic configuration file that ReFrame uses. Note that you should *not* edit this configuration file in place.

```
cp reframe/core/settings.py tutorials/config/mysettings.py
```

Here is how the new configuration file looks like with the needed additions highlighted:

```
site_configuration = {
    'systems': [
        {
            'name': 'catalina',
            'descr': 'My Mac',
            'hostnames': ['tresa'],
            'modules_system': 'nomod',
            'partitions': [
                {
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['gnu', 'clang'],
                }
            ]
        },
        {
            'name': 'generic',
            'descr': 'Generic example system',
            'hostnames': ['.*'],
            'partitions': [
                {
```

(continues on next page)

(continued from previous page)

```
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['builtin']
                }
            ]
        },
    ],
    'environments': [
        {
            'name': 'gnu',
            'cc': 'gcc-9',
            'cxx': 'g++-9',
            'ftn': 'gfortran-9'
        },
        {
            'name': 'clang',
            'cc': 'clang',
            'cxx': 'clang++',
            'ftn': ''
        },
        {
            'name': 'builtin',
            'cc': 'cc',
            'cxx': '',
            'ftn': ''
        },
        {
            'name': 'builtin',
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        }
    ],
    'logging': [
        {
            'level': 'debug',
            'handlers': [
                {
                    'type': 'stream',
                    'name': 'stdout',
                    'level': 'info',
                    'format': '%(message)s'
                },
                {
                    'type': 'file',
                    'level': 'debug',
                    'format': '[%(asctime)s] %(levelname)s: %(check_info)s:
↪%(message)s',    # noqa: E501
                    'append': False
                }
            ],
            'handlers_perflog': [
                {
                    'type': 'filelog',
                    'prefix': '%(check_system)s/%(check_partition)s',
```

(continues on next page)

(continued from previous page)

```
                    '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)|'
                        '%(check_perf_unit)s'
                    ),
                    'append': True
                }
            ]
        }
    ],
}
```

Here we define a system named `catalina` that has one partition named `default`. This partition makes no use of any workload manager, but instead launches any jobs locally as OS processes. Two programming environments are relevant for that partition, namely `gnu` and `clang`, which are defined in the section `environments` of the configuration file. The `gnu` programming environment provides GCC 9, whereas the `clang` one provides the Clang compiler from the system. Notice, how you can define the actual commands for invoking the C, C++ and Fortran compilers in each programming environment. As soon as a programming environment defines the different compilers, ReFrame will automatically pick the right compiler based on the source file extension. In addition to C, C++ and Fortran programs, ReFrame will recognize the `.cu` extension as well and will try to invoke the `nvcc` compiler for CUDA programs.

Finally, the new system that we defined may be identified by the hostname `tresa` (see the `hostnames` configuration parameter) and it will not use any environment modules system (see the `modules_system` configuration parameter). The `hostnames` attribute will help ReFrame to automatically pick the right configuration when running on it. Notice, how the `generic` system matches any hostname, so that it acts as a fallback system.

Note: The different systems in the configuration file are tried in order and the first match is picked. This practically means that the more general the selection pattern for a system is, the lower in the list of systems it should be.

The *Configuring ReFrame for Your Site* page describes the configuration file in more detail and the *Configuration Reference* provides a complete reference guide of all the configuration options of ReFrame.

Let's now rerun our "Hello, World!" tests:

```
./bin/reframe -C tutorials/config/mysettings.py -c tutorials/basics/hello/hello2.py -r
```

```
[ReFrame Setup]
  version:           3.6.0-dev.0+a3d0b0cd
  command:           './bin/reframe -C tutorials/config/mysettings.py -c tutorials/
↪basics/hello/hello2.py -r'
  launched by:       user@tresa.local
  working directory: '/Users/user/Repositories/reframe'
  settings file:     'tutorials/config/settings.py'
  check search path: '/Users/user/Repositories/reframe/tutorials/basics/hello/hello2.
↪py'
  stage directory:   '/Users/user/Repositories/reframe/stage'
  output directory:  '/Users/user/Repositories/reframe/output'
```

(continues on next page)

(continued from previous page)

```
[==========] Running 2 check(s)
[==========] Started on Tue Mar  9 23:28:00 2021

[----------] started processing HelloMultiLangTest_c (HelloMultiLangTest_c)
[ RUN      ] HelloMultiLangTest_c on catalina:default using gnu
[ RUN      ] HelloMultiLangTest_c on catalina:default using clang
[----------] finished processing HelloMultiLangTest_c (HelloMultiLangTest_c)

[----------] started processing HelloMultiLangTest_cpp (HelloMultiLangTest_cpp)
[ RUN      ] HelloMultiLangTest_cpp on catalina:default using gnu
[ RUN      ] HelloMultiLangTest_cpp on catalina:default using clang
[----------] finished processing HelloMultiLangTest_cpp (HelloMultiLangTest_cpp)

[----------] waiting for spawned checks to finish
[       OK ] (1/4) HelloMultiLangTest_cpp on catalina:default using gnu [compile: 0.
↪768s run: 1.115s total: 1.909s]
[       OK ] (2/4) HelloMultiLangTest_c on catalina:default using gnu [compile: 0.
↪600s run: 2.230s total: 2.857s]
[       OK ] (3/4) HelloMultiLangTest_c on catalina:default using clang [compile: 0.
↪238s run: 2.129s total: 2.393s]
[       OK ] (4/4) HelloMultiLangTest_cpp on catalina:default using clang [compile: 1.
↪006s run: 0.427s total: 1.456s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 4/4 test case(s) from 2 check(s) (0 failure(s))
[==========] Finished on Tue Mar  9 23:28:03 2021
Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-dnubkvfi.
↪log'
```

Notice how the same tests are now tried with both the `gnu` and `clang` programming environments, without having to touch them at all! That's one of the powerful features of ReFrame and we shall see later on, how easily we can port our tests to an HPC cluster with minimal changes. In order to instruct ReFrame to use our configuration file, we use the `-C` command line option. Since we don't want to type it throughout the tutorial, we will now set it in the environment:

```
export RFM_CONFIG_FILE=$(pwd)/tutorials/config/mysettings.py
```

A Multithreaded "Hello, World!"

We extend our C++ "Hello, World!" example to print the greetings from multiple threads:

```
cat tutorials/basics/hellomp/src/hello_threads.cpp
```

```
#include <iomanip>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>


#ifdef SYNC_MESSAGES
std::mutex hello_mutex;
#endif
```

(continues on next page)

(continued from previous page)

```
void greetings(int tid)
{
#ifdef SYNC_MESSAGES
    const std::lock_guard<std::mutex> lock(hello_mutex);
#endif
    std::cout << "[" << std::setw(2) << tid << "] " << "Hello, World!\n";
}


int main(int argc, char *argv[])
{
    int nr_threads = 1;
    if (argc > 1) {
        nr_threads = std::atoi(argv[1]);
    }

    if (nr_threads <= 0) {
        std::cerr << "thread count must a be positive integer\n";
        return 1;
    }

    std::vector<std::thread> threads;
    for (auto i = 0; i < nr_threads; ++i) {
        threads.push_back(std::thread(greetings, i));
    }

    for (auto &t : threads) {
        t.join();
    }

    return 0;
}
```

This program takes as argument the number of threads it will create and it uses `std::thread`, which is a C++11 addition, meaning that we will need to pass `-std=c++11` to our compilers. Here is the corresponding ReFrame test, where the new concepts introduced are highlighted:

```
cat tutorials/basics/hellomp/hellomp1.py
```

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


@rfm.simple_test
class HelloThreadedTest(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    sourcepath = 'hello_threads.cpp'
    build_system = 'SingleSource'
    executable_opts = ['16']

    @rfm.run_before('compile')
    def set_compilation_flags(self):
        self.build_system.cxxflags = ['-std=c++11', '-Wall']
        environ = self.current_environ.name
        if environ in {'clang', 'gnu'}:
```

(continues on next page)

(continued from previous page)

```
            self.build_system.cxxflags += ['-pthread']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(r'Hello, World\!', self.stdout)
```

ReFrame delegates the compilation of a test to a *build_system*, which is an abstraction of the steps needed to compile the test. Build systems take also care of interactions with the programming environment if necessary. Compilation flags are a property of the build system. If not explicitly specified, ReFrame will try to pick the correct build system (e.g., CMake, Autotools etc.) by inspecting the test resources, but in cases as the one presented here where we need to set the compilation flags, we need to specify a build system explicitly. In this example, we instruct ReFrame to compile a single source file using the `-std=c++11 -pthread -Wall` compilation flags. However, the flag `-pthread` is only needed to compile applications using `std::thread` with the GCC and Clang compilers. Hence, since this flag may not be valid for other compilers, we need to include it only in the tests that use either GCC or Clang. Similarly to the `lang` parameter in the previous example, the information regarding which compiler is being used is only available after the class is instantiated (after completion of the `setup` pipeline stage), so we also defer the addition of this optional compiler flag with a pipeline hook. In this case, we set the `set_compile_flags()` hook to run before the ReFrame pipeline stage `compile`.

Note: The pipeline hooks, as well as the regression test pipeline itself, are covered in more detail later on in the tutorial.

In this example, the generated executable takes a single argument which sets the number of threads that will be used. As seen in the previous examples, executable options are defined with the *executable_opts* variable, and here is set to `'16'`. Also, the reader may notice that this example no longer redirects the standard output of the executable into a file as the previous examples did. Instead, just with the purpose of keeping the *executable_opts* simple, we use ReFrame's internal mechanism to process the standard output of the executable. Similarly to the parameters and the compiler settings, the output of a test is private to each of the instances of the `HelloThreadedTest` class. So, instead of inspecting an external file to evaluate the sanity of the test, we can just set our sanity function to inspect this attribute that contains the test's standard output. This output is stored under `self.stdout` and is populated only after the executable has run. Therefore, we can set the *sanity_patterns* with the `set_sanity_patterns()` pipeline hook that is scheduled to run before the `sanity` pipeline stage. Again, pipeline stages will be covered detail further on, so for now, just think of this `sanity` stage as a step that occurs after the test's executable is run.

Let's run the test now:

```
./bin/reframe -c tutorials/basics/hellomp/hellomp1.py -r
```

```
[ReFrame Setup]
  version:           3.3-dev0 (rev: 5d246bff)
  command:           './bin/reframe -c tutorials/basics/hellomp/hellomp1.py -r'
  launched by:       user@tresa.local
  working directory: '/Users/user/Repositories/reframe'
  settings file:     '/Users/user/Repositories/reframe/tutorials/config/settings.py'
  check search path: '/Users/user/Repositories/reframe/tutorials/basics/hellomp/
↪hellomp1.py'
  stage directory:   '/Users/user/Repositories/reframe/stage'
  output directory:  '/Users/user/Repositories/reframe/output'

[==========] Running 1 check(s)
[==========] Started on Mon Oct 12 20:02:37 2020

[----------] started processing HelloThreadedTest (HelloThreadedTest)
```

(continues on next page)

(continued from previous page)

```
[ RUN      ] HelloThreadedTest on catalina:default using gnu
[ RUN      ] HelloThreadedTest on catalina:default using clang
[----------] finished processing HelloThreadedTest (HelloThreadedTest)

[----------] waiting for spawned checks to finish
[       OK ] (1/2) HelloThreadedTest on catalina:default using gnu [compile: 1.591s␣
↪run: 1.205s total: 2.816s]
[       OK ] (2/2) HelloThreadedTest on catalina:default using clang [compile: 1.141s␣
↪run: 0.309s total: 1.465s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 2 test case(s) from 1 check(s) (0 failure(s))
[==========] Finished on Mon Oct 12 20:02:40 2020
Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-h_itoc1k.
↪log'
```

Everything looks fine, but let's inspect the actual output of one of the tests:

```
cat output/catalina/default/clang/HelloThreadedTest/rfm_HelloThreadedTest_job.out
```

```
[[[[    8] Hello, World!
1] Hello, World!
5[[0[ 7] Hello, World!
] ] Hello, World!
[ Hello, World!
6[] Hello, World!
9] Hello, World!
 2 ] Hello, World!
4] [[10 3] Hello, World!
] Hello, World!
[Hello, World!
11] Hello, World!
[12] Hello, World!
[13] Hello, World!
[14] Hello, World!
[15] Hello, World!
```

Not exactly what we were looking for! In the following we write a more robust sanity check that can catch this havoc.

More advanced sanity checking

Sanity checking of a test's outcome is quite powerful in ReFrame. So far, we have seen only a `grep`-like search for a string in the output, but ReFrame's `sanity_patterns` are much more capable than this. In fact, you can practically do almost any operation in the output and process it as you would like before assessing the test's sanity. The syntax feels also quite natural since it is fully integrated in Python.

In the following we extend the sanity checking of the multithreaded "Hello, World!", such that not only the output pattern we are looking for is more restrictive, but also we check that all the threads produce a greetings line. See the highlighted lines in the modified version of the `set_sanity_patterns` pipeline hook.

```
cat tutorials/basics/hellomp/hellomp2.py
```

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

(continues on next page)

(continued from previous page)

```
@rfm.simple_test
class HelloThreadedExtendedTest(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    sourcepath = 'hello_threads.cpp'
    build_system = 'SingleSource'
    executable_opts = ['16']

    @rfm.run_before('compile')
    def set_compilation_flags(self):
        self.build_system.cxxflags = ['-std=c++11', '-Wall']
        environ = self.current_environ.name
        if environ in {'clang', 'gnu'}:
            self.build_system.cxxflags += ['-pthread']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        num_messages = sn.len(sn.findall(r'\[\s?\d+\] Hello, World\!',
                                         self.stdout))
        self.sanity_patterns = sn.assert_eq(num_messages, 16)
```

The sanity checking is straightforward. We find all the matches of the required pattern, we count them and finally we check their number. Both statements here are lazily evaluated. They will not be executed where they appear, but rather at the sanity checking phase. ReFrame provides lazily evaluated counterparts for most of the builtin Python functions, such the `len()` function here. Also whole expressions can be lazily evaluated if one of the operands is deferred, as is the case in this example with the assignment to `num_messages`. This makes the sanity checking mechanism quite powerful and straightforward to reason about, without having to rely on complex pattern matching techniques. *Sanity Functions Reference* provides a complete reference of the sanity functions provided by ReFrame, but users can also define their own, as described in *Understanding the Mechanism of Sanity Functions*.

Let's run this version of the test now and see if it fails:

```
./bin/reframe -c tutorials/basics/hellomp/hellomp2.py -r
```

```
[ReFrame Setup]
  version:           3.3-dev0 (rev: 5d246bff)
  command:           './bin/reframe -c tutorials/basics/hellomp/hellomp2.py -r'
  launched by:       user@tresa.local
  working directory: '/Users/user/Repositories/reframe'
  settings file:     '/Users/user/Repositories/reframe/tutorials/config/settings.py'
  check search path: '/Users/user/Repositories/reframe/tutorials/basics/hellomp/
↪hellomp2.py'
  stage directory:   '/Users/user/Repositories/reframe/stage'
  output directory:  '/Users/user/Repositories/reframe/output'

[==========] Running 1 check(s)
[==========] Started on Mon Oct 12 20:04:59 2020

[----------] started processing HelloThreadedExtendedTest (HelloThreadedExtendedTest)
[ RUN      ] HelloThreadedExtendedTest on catalina:default using gnu
[ RUN      ] HelloThreadedExtendedTest on catalina:default using clang
[----------] finished processing HelloThreadedExtendedTest (HelloThreadedExtendedTest)

[----------] waiting for spawned checks to finish
```

(continues on next page)

(continued from previous page)

```
[     FAIL ] (1/2) HelloThreadedExtendedTest on catalina:default using gnu [compile:␣
↪1.222s run: 0.891s total: 2.130s]
[     FAIL ] (2/2) HelloThreadedExtendedTest on catalina:default using clang␣
↪[compile: 0.835s run: 0.167s total: 1.018s]
[----------] all spawned checks have finished

[  FAILED  ] Ran 2 test case(s) from 1 check(s) (2 failure(s))
[==========] Finished on Mon Oct 12 20:05:02 2020

==============================================================================
SUMMARY OF FAILURES
------------------------------------------------------------------------------
FAILURE INFO for HelloThreadedExtendedTest
  * Test Description: HelloThreadedExtendedTest
  * System partition: catalina:default
  * Environment: gnu
  * Stage directory: /Users/user/Repositories/reframe/stage/catalina/default/gnu/
↪HelloThreadedExtendedTest
  * Node list: tresa.local
  * Job type: local (id=60355)
  * Maintainers: []
  * Failing phase: sanity
  * Rerun with '-n HelloThreadedExtendedTest -p gnu --system catalina:default'
  * Reason: sanity error: 12 != 16
------------------------------------------------------------------------------
FAILURE INFO for HelloThreadedExtendedTest
  * Test Description: HelloThreadedExtendedTest
  * System partition: catalina:default
  * Environment: clang
  * Stage directory: /Users/user/Repositories/reframe/stage/catalina/default/clang/
↪HelloThreadedExtendedTest
  * Node list: tresa.local
  * Job type: local (id=60366)
  * Maintainers: []
  * Failing phase: sanity
  * Rerun with '-n HelloThreadedExtendedTest -p clang --system catalina:default'
  * Reason: sanity error: 6 != 16
------------------------------------------------------------------------------
Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-zz7x_5c8.
↪log'
```

As expected, only some of lines are printed correctly which makes the test fail. To fix this test, we need to compile with `-DSYNC_MESSAGES`, which will synchronize the printing of messages.

```
cat tutorials/basics/hellomp/hellomp3.py
```

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


@rfm.simple_test
class HelloThreadedExtended2Test(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    sourcepath = 'hello_threads.cpp'
    build_system = 'SingleSource'
```

(continues on next page)

(continued from previous page)

```
    executable_opts = ['16']

    @rfm.run_before('compile')
    def set_compilation_flags(self):
        self.build_system.cppflags = ['-DSYNC_MESSAGES']
        self.build_system.cxxflags = ['-std=c++11', '-Wall']
        environ = self.current_environ.name
        if environ in {'clang', 'gnu'}:
            self.build_system.cxxflags += ['-pthread']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        num_messages = sn.len(sn.findall(r'\[\s?\d+\] Hello, World\!',
                                         self.stdout))
        self.sanity_patterns = sn.assert_eq(num_messages, 16)
```

Writing A Performance Test

An important aspect of regression testing is checking for performance regressions. In this example, we will write a test that downloads the STREAM benchmark, compiles it, runs it and records its performance. In the test below, we highlight the lines that introduce new concepts.

```
cat tutorials/basics/stream/stream1.py
```

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


@rfm.simple_test
class StreamTest(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['gnu']
    prebuild_cmds = [
        'wget http://www.cs.virginia.edu/stream/FTP/Code/stream.c',
    ]
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    variables = {
        'OMP_NUM_THREADS': '4',
        'OMP_PLACES': 'cores'
    }

    @rfm.run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = ['-DSTREAM_ARRAY_SIZE=$((1 << 25))']
        self.build_system.cflags = ['-fopenmp', '-O3', '-Wall']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(r'Solution Validates',
                                               self.stdout)

    @rfm.run_before('performance')
    def set_perf_patterns(self):
        self.perf_patterns = {
```

(continues on next page)

(continued from previous page)

```
            'Copy': sn.extractsingle(r'Copy:\s+(\S+)\s+.*',
                                     self.stdout, 1, float),
            'Scale': sn.extractsingle(r'Scale:\s+(\S+)\s+.*',
                                      self.stdout, 1, float),
            'Add': sn.extractsingle(r'Add:\s+(\S+)\s+.*',
                                    self.stdout, 1, float),
            'Triad': sn.extractsingle(r'Triad:\s+(\S+)\s+.*',
                                      self.stdout, 1, float)
        }
```

First of all, notice that we restrict the programming environments to `gnu` only, since this test requires OpenMP, which our installation of Clang does not have. The next thing to notice is the *`prebuild_cmds`* attribute, which provides a list of commands to be executed before the build step. These commands will be executed from the test's stage directory. In this case, we just fetch the source code of the benchmark. For running the benchmark, we need to set the OpenMP number of threads and pin them to the right CPUs through the `OMP_NUM_THREADS` and `OMP_PLACES` environment variables. You can set environment variables in a ReFrame test through the *`variables`* dictionary.

What makes a ReFrame test a performance test is the definition of the *`perf_patterns`* attribute. This is a dictionary where the keys are *performance variables* and the values are lazily evaluated expressions for extracting the performance variable values from the test's output. In this example, we extract four performance variables, namely the memory bandwidth values for each of the "Copy", "Scale", "Add" and "Triad" sub-benchmarks of STREAM and we do so by using the *`extractsingle()`* sanity function. For each of the sub-benchmarks we extract the "Best Rate MB/s" column of the output (see below) and we convert that to a float.

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           24939.4     0.021905     0.021527     0.022382
Scale:          16956.3     0.031957     0.031662     0.032379
Add:            18648.2     0.044277     0.043184     0.046349
Triad:          19133.4     0.042935     0.042089     0.044283
```

Let's run the test now:

```
./bin/reframe -c tutorials/basics/stream/stream1.py -r --performance-report
```

The *`--performance-report`* will generate a short report at the end for each performance test that has run.

```
[ReFrame Setup]
  version:           3.3-dev0 (rev: 5d246bff)
  command:           './bin/reframe -c tutorials/basics/stream/stream1.py -r --
↪performance-report'
  launched by:       user@tresa.local
  working directory: '/Users/user/Repositories/reframe'
  settings file:     '/Users/user/Repositories/reframe/tutorials/config/settings.py'
  check search path: '/Users/user/Repositories/reframe/tutorials/basics/stream/
↪stream1.py'
  stage directory:   '/Users/user/Repositories/reframe/stage'
  output directory:  '/Users/user/Repositories/reframe/output'

[==========] Running 1 check(s)
[==========] Started on Mon Oct 12 20:06:09 2020

[----------] started processing StreamTest (StreamTest)
[ RUN      ] StreamTest on catalina:default using gnu
[----------] finished processing StreamTest (StreamTest)

[----------] waiting for spawned checks to finish
```

(continues on next page)

(continued from previous page)

```
[       OK ] (1/1) StreamTest on catalina:default using gnu [compile: 1.386s run: 2.
↪377s total: 3.780s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 1 test case(s) from 1 check(s) (0 failure(s))
[==========] Finished on Mon Oct 12 20:06:13 2020
================================================================================
PERFORMANCE REPORT
--------------------------------------------------------------------------------
StreamTest
- catalina:default
   - gnu
      * num_tasks: 1
      * Copy: 24326.7 None
      * Scale: 16664.2 None
      * Add: 18398.7 None
      * Triad: 18930.6 None
--------------------------------------------------------------------------------
Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-gczplnic.
↪log'
```

Adding reference values

A performance test would not be so meaningful, if we couldn't test the obtained performance against a reference value. ReFrame offers the possibility to set references for each of the performance variables defined in a test and also set different references for different systems. In the following example, we set the reference values for all the STREAM sub-benchmarks for the system we are currently running on.

Note: Optimizing STREAM benchmark performance is outside the scope of this tutorial.

```
cat tutorials/basics/stream/stream2.py
```

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


@rfm.simple_test
class StreamWithRefTest(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['gnu']
    prebuild_cmds = [
        'wget http://www.cs.virginia.edu/stream/FTP/Code/stream.c',
    ]
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    variables = {
        'OMP_NUM_THREADS': '4',
        'OMP_PLACES': 'cores'
    }
    reference = {
        'catalina': {
            'Copy':  (25200, -0.05, 0.05, 'MB/s'),
```

(continues on next page)

(continued from previous page)

```
            'Scale': (16800, -0.05, 0.05, 'MB/s'),
            'Add':   (18500, -0.05, 0.05, 'MB/s'),
            'Triad': (18800, -0.05, 0.05, 'MB/s')
        }
    }

    @rfm.run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = ['-DSTREAM_ARRAY_SIZE=$((1 << 25))']
        self.build_system.cflags = ['-fopenmp', '-O3', '-Wall']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(r'Solution Validates',
                                               self.stdout)

    @rfm.run_before('performance')
    def set_perf_patterns(self):
        self.perf_patterns = {
            'Copy': sn.extractsingle(r'Copy:\s+(\S+)\s+.*',
                                     self.stdout, 1, float),
            'Scale': sn.extractsingle(r'Scale:\s+(\S+)\s+.*',
                                      self.stdout, 1, float),
            'Add': sn.extractsingle(r'Add:\s+(\S+)\s+.*',
                                    self.stdout, 1, float),
            'Triad': sn.extractsingle(r'Triad:\s+(\S+)\s+.*',
                                      self.stdout, 1, float)
        }
```

The performance reference tuple consists of the reference value, the lower and upper thresholds expressed as fractional numbers relative to the reference value, and the unit of measurement. If any of the thresholds is not relevant, `None` may be used instead.

If any obtained performance value is beyond its respective thresholds, the test will fail with a summary as shown below:

```
./bin/reframe -c tutorials/basics/stream/stream2.py -r --performance-report
```

```
FAILURE INFO for StreamWithRefTest
  * Test Description: StreamWithRefTest
  * System partition: catalina:default
  * Environment: gnu
  * Stage directory: /Users/user/Repositories/reframe/stage/catalina/default/gnu/
↪StreamWithRefTest
  * Node list: tresa.local
  * Job type: local (id=62114)
  * Maintainers: []
  * Failing phase: performance
  * Rerun with '-n StreamWithRefTest -p gnu --system catalina:default'
  * Reason: performance error: failed to meet reference: Copy=24586.5, expected 55200␣
↪(l=52440.0, u=57960.0)
```

Examining the performance logs

ReFrame has a powerful mechanism for logging its activities as well as performance data. It supports different types of log channels and it can send data simultaneously in any number of them. For example, performance data might be logged in files and the same time being sent to Syslog or to a centralized log management server. By default (i.e., starting off from the builtin configuration file), ReFrame sends performance data to files per test under the `perflogs/` directory:

```
perflogs
└── catalina
    └── default
        ├── StreamTest.log
        └── StreamWithRefTest.log
```

ReFrame creates a log file per test per system and per partition and appends to it every time the test is run on that system/partition combination. Let's inspect the log file from our last test:

```
tail perflogs/catalina/default/StreamWithRefTest.log
```

```
2020-06-24T00:27:06|reframe 3.1-dev0 (rev: 9d92d0ec)|StreamWithRefTest on␣
↪catalina:default using gnu|jobid=58384|Copy=24762.2|ref=25200 (l=-0.05, u=0.05)|MB/s
2020-06-24T00:27:06|reframe 3.1-dev0 (rev: 9d92d0ec)|StreamWithRefTest on␣
↪catalina:default using gnu|jobid=58384|Scale=16784.6|ref=16800 (l=-0.05, u=0.05)|MB/
↪s
2020-06-24T00:27:06|reframe 3.1-dev0 (rev: 9d92d0ec)|StreamWithRefTest on␣
↪catalina:default using gnu|jobid=58384|Add=18553.8|ref=18500 (l=-0.05, u=0.05)|MB/s
2020-06-24T00:27:06|reframe 3.1-dev0 (rev: 9d92d0ec)|StreamWithRefTest on␣
↪catalina:default using gnu|jobid=58384|Triad=18679.0|ref=18800 (l=-0.05, u=0.05)|MB/
↪s
2020-06-24T12:42:07|reframe 3.1-dev0 (rev: 138cbd68)|StreamWithRefTest on␣
↪catalina:default using gnu|jobid=62114|Copy=24586.5|ref=55200 (l=-0.05, u=0.05)|MB/s
2020-06-24T12:42:07|reframe 3.1-dev0 (rev: 138cbd68)|StreamWithRefTest on␣
↪catalina:default using gnu|jobid=62114|Scale=16880.6|ref=16800 (l=-0.05, u=0.05)|MB/
↪s
2020-06-24T12:42:07|reframe 3.1-dev0 (rev: 138cbd68)|StreamWithRefTest on␣
↪catalina:default using gnu|jobid=62114|Add=18570.4|ref=18500 (l=-0.05, u=0.05)|MB/s
2020-06-24T12:42:07|reframe 3.1-dev0 (rev: 138cbd68)|StreamWithRefTest on␣
↪catalina:default using gnu|jobid=62114|Triad=19048.3|ref=18800 (l=-0.05, u=0.05)|MB/
↪s
```

Several information are printed for each run, such as the performance variables, their value, their references and thresholds etc. The default format is in a form suitable for easy parsing, but you may fully control not only the format, but also what is being logged from the configuration file. *Configuring ReFrame for Your Site* and *Configuration Reference* cover logging in ReFrame in much more detail.

Porting The Tests to an HPC cluster

It's now time to port our tests to an HPC cluster. Obviously, HPC clusters are much more complex than our laptop or PC. Usually there are many more compilers, the user environment is handled in a different way, and the way to launch the tests varies significantly, since you have to go through a workload manager in order to access the actual compute nodes. Besides that, there might be multiple types of compute nodes that we would like to run our tests on, but each type might be accessed in a different way. It is already apparent that porting even an as simple as a "Hello, World" test to such a system is not that straightforward. As we shall see in this section, ReFrame makes that pretty easy.

Adapting the configuration

Our target system is the Piz Daint supercomputer at CSCS, but you can adapt the process to your target HPC system. In ReFrame, all the details of the various interactions of a test with the system environment are handled transparently and are set up in its configuration file. Let's extend our configuration file for Piz Daint.

```
site_configuration = {
    'systems': [
        {
            'name': 'catalina',
            'descr': 'My Mac',
            'hostnames': ['tresa'],
            'modules_system': 'nomod',
            'partitions': [
                {
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['gnu', 'clang'],
                }
            ]
        },
        {
            'name': 'daint',
            'descr': 'Piz Daint Supercomputer',
            'hostnames': ['daint'],
            'modules_system': 'tmod32',
            'partitions': [
                {
                    'name': 'login',
                    'descr': 'Login nodes',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['builtin', 'gnu', 'intel', 'pgi', 'cray'],
                },
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C gpu', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                },
                {
                    'name': 'mc',
                    'descr': 'Multicore nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C mc', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                }
            ]
        },
        {
            'name': 'generic',
```

(continues on next page)

(continued from previous page)

```
            'descr': 'Generic example system',
            'hostnames': ['.*'],
            'partitions': [
                {
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['builtin']
                }
            ]
        },
    ],
    'environments': [
        {
            'name': 'gnu',
            'cc': 'gcc-9',
            'cxx': 'g++-9',
            'ftn': 'gfortran-9'
        },
        {
            'name': 'gnu',
            'modules': ['PrgEnv-gnu'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'cray',
            'modules': ['PrgEnv-cray'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'intel',
            'modules': ['PrgEnv-intel'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'pgi',
            'modules': ['PrgEnv-pgi'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'clang',
            'cc': 'clang',
            'cxx': 'clang++',
            'ftn': ''
        },
```

(continues on next page)

(continued from previous page)

```
        {
            'name': 'builtin',
            'cc': 'cc',
            'cxx': '',
            'ftn': ''
        },
        {
            'name': 'builtin',
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        }
    ],
    'logging': [
        {
            'level': 'debug',
            'handlers': [
                {
                    'type': 'stream',
                    'name': 'stdout',
                    'level': 'info',
                    'format': '%(message)s'
                },
                {
                    'type': 'file',
                    'level': 'debug',
                    'format': '[%(asctime)s] %(levelname)s: %(check_info)s:
↪%(message)s',   # noqa: E501
                    '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)|'
                        '%(check_perf_unit)s'
                    ),
                    'append': True
                }
            ]
        }
    ],
}
```

First of all, we need to define a new system and set the list of hostnames that will help ReFrame identify it. We also set the `modules_system` configuration parameter to instruct ReFrame that this system makes use of the environment modules for managing the user environment. Then we define the system partitions that we want to test. In this case, we define three partitions:

1. the login nodes,
2. the multicore partition (2x Broadwell CPUs per node) and
3. the hybrid partition (1x Haswell CPU + 1x Pascal GPU).

The login nodes are pretty much similar to the `catalina:default` partition which corresponded to our laptop: tests will be launched and run locally. The other two partitions are handled by Slurm and parallel jobs are launched using the `srun` command. Additionally, in order to access the different types of nodes represented by those partitions, users have to specify either `-C mc` or `-C gpu` options along with their account. This is what we do exactly with the `access` partition configuration option.

Note: System partitions in ReFrame do not necessarily correspond to real job scheduler partitions.

Piz Daint's programming environment offers four compilers: Cray, GNU, Intel and PGI. We want to test all of them, so we include them in the `environs` lists. Notice that we do not include Clang in the list, since there is no such compiler on this particular system. On the other hand, we include a different version of the `builtin` environment, which corresponds to the default login environment without loading any modules. It is generally useful to define such an environment so as to use it for tests that are running simple utilities and don't need to compile anything.

Before looking into the definition of the new environments for the four compilers, it is worth mentioning the `max_jobs` parameter. This parameter specifies the maximum number of ReFrame test jobs that can be simultaneously in flight. ReFrame will try to keep concurrency close to this limit (but not exceeding it). By default, this is set to `8`, so you are advised to set it to a higher number if you want to increase the throughput of completed tests.

The new environments are defined similarly to the ones we had for our local system, except that now we set two more parameters: the `modules` and the `target_systems`. The `modules` parameter is a list of environment modules that needs to be loaded, in order to make available this compiler. The `target_systems` parameter restricts the environment definition to a list of specific systems or system partitions. This allows us to redefine environments for different systems, as for example the `gnu` environment in this case. ReFrame will always pick the definition that is a closest match for the current system. In this example, it will pick the second definition for `gnu` whenever it runs on the system named `daint`, and the first in every other occasion.

Running the tests

We are now ready to run our tests on Piz Daint. We will only do so with the final versions of the tests from the previous section, which we will select using `-n` option.

```
export RFM_CONFIG_FILE=$(pwd)/tutorials/config/mysettings.py
./bin/reframe -c tutorials/basics/ -R -n
↪'HelloMultiLangTest|HelloThreadedExtended2Test|StreamWithRefTest' --performance-
↪report -r
```

```
[ReFrame Setup]
  version:           3.4-dev2 (rev: f102d4bb)
  command:           './bin/reframe -c tutorials/basics/ -R -n␣
↪HelloMultiLangTest|HelloThreadedExtended2Test|StreamWithRefTest --performance-
↪report -r'
  launched by:       user@dom101
  working directory: '/users/user/Devel/reframe'
  settings file:     '/users/user/Devel/reframe/tutorials/config/settings.py'
  check search path: (R) '/users/user/Devel/reframe/tutorials/basics'
  stage directory:   '/users/user/Devel/reframe/stage'
  output directory:  '/users/user/Devel/reframe/output'
```

(continues on next page)

(continued from previous page)

```
[==========] Running 4 check(s)
[==========] Started on Mon Jan 25 00:34:32 2021

[----------] started processing HelloMultiLangTest_c (HelloMultiLangTest_c)
[ RUN      ] HelloMultiLangTest_c on daint:login using builtin
[ RUN      ] HelloMultiLangTest_c on daint:login using gnu
[ RUN      ] HelloMultiLangTest_c on daint:login using intel
[ RUN      ] HelloMultiLangTest_c on daint:login using pgi
[ RUN      ] HelloMultiLangTest_c on daint:login using cray
[ RUN      ] HelloMultiLangTest_c on daint:gpu using gnu
[ RUN      ] HelloMultiLangTest_c on daint:gpu using intel
[ RUN      ] HelloMultiLangTest_c on daint:gpu using pgi
[ RUN      ] HelloMultiLangTest_c on daint:gpu using cray
[ RUN      ] HelloMultiLangTest_c on daint:mc using gnu
[ RUN      ] HelloMultiLangTest_c on daint:mc using intel
[ RUN      ] HelloMultiLangTest_c on daint:mc using pgi
[ RUN      ] HelloMultiLangTest_c on daint:mc using cray
[----------] finished processing HelloMultiLangTest_c (HelloMultiLangTest_c)

[----------] started processing HelloMultiLangTest_cpp (HelloMultiLangTest_cpp)
[ RUN      ] HelloMultiLangTest_cpp on daint:login using builtin
[ RUN      ] HelloMultiLangTest_cpp on daint:login using gnu
[ RUN      ] HelloMultiLangTest_cpp on daint:login using intel
[ RUN      ] HelloMultiLangTest_cpp on daint:login using pgi
[ RUN      ] HelloMultiLangTest_cpp on daint:login using cray
[ RUN      ] HelloMultiLangTest_cpp on daint:gpu using gnu
[ RUN      ] HelloMultiLangTest_cpp on daint:gpu using intel
[ RUN      ] HelloMultiLangTest_cpp on daint:gpu using pgi
[ RUN      ] HelloMultiLangTest_cpp on daint:gpu using cray
[ RUN      ] HelloMultiLangTest_cpp on daint:mc using gnu
[ RUN      ] HelloMultiLangTest_cpp on daint:mc using intel
[ RUN      ] HelloMultiLangTest_cpp on daint:mc using pgi
[ RUN      ] HelloMultiLangTest_cpp on daint:mc using cray
[----------] finished processing HelloMultiLangTest_cpp (HelloMultiLangTest_cpp)

[----------] started processing HelloThreadedExtended2Test␣
↪(HelloThreadedExtended2Test)
[ RUN      ] HelloThreadedExtended2Test on daint:login using builtin
[ RUN      ] HelloThreadedExtended2Test on daint:login using gnu
[ RUN      ] HelloThreadedExtended2Test on daint:login using intel
[ RUN      ] HelloThreadedExtended2Test on daint:login using pgi
[ RUN      ] HelloThreadedExtended2Test on daint:login using cray
[ RUN      ] HelloThreadedExtended2Test on daint:gpu using gnu
[ RUN      ] HelloThreadedExtended2Test on daint:gpu using intel
[ RUN      ] HelloThreadedExtended2Test on daint:gpu using pgi
[ RUN      ] HelloThreadedExtended2Test on daint:gpu using cray
[ RUN      ] HelloThreadedExtended2Test on daint:mc using gnu
[ RUN      ] HelloThreadedExtended2Test on daint:mc using intel
[ RUN      ] HelloThreadedExtended2Test on daint:mc using pgi
[ RUN      ] HelloThreadedExtended2Test on daint:mc using cray
[----------] finished processing HelloThreadedExtended2Test␣
↪(HelloThreadedExtended2Test)

[----------] started processing StreamWithRefTest (StreamWithRefTest)
[ RUN      ] StreamWithRefTest on daint:login using gnu
[ RUN      ] StreamWithRefTest on daint:gpu using gnu
[ RUN      ] StreamWithRefTest on daint:mc using gnu
```

(continues on next page)

(continued from previous page)

```
[----------] finished processing StreamWithRefTest (StreamWithRefTest)

[----------] waiting for spawned checks to finish
[       OK ] ( 1/42) HelloThreadedExtended2Test on daint:login using cray [compile: 0.
↪959s run: 56.203s total: 57.189s]
[       OK ] ( 2/42) HelloThreadedExtended2Test on daint:login using intel [compile:␣
↪2.096s run: 61.438s total: 64.062s]
[       OK ] ( 3/42) HelloMultiLangTest_cpp on daint:login using cray [compile: 0.
↪479s run: 98.909s total: 99.406s]
[       OK ] ( 4/42) HelloMultiLangTest_c on daint:login using pgi [compile: 1.342s␣
↪run: 137.250s total: 138.609s]
[       OK ] ( 5/42) HelloThreadedExtended2Test on daint:gpu using cray [compile: 0.
↪792s run: 33.748s total: 34.558s]
[       OK ] ( 6/42) HelloThreadedExtended2Test on daint:gpu using intel [compile: 2.
↪257s run: 48.545s total: 50.825s]
[       OK ] ( 7/42) HelloMultiLangTest_cpp on daint:gpu using cray [compile: 0.469s␣
↪run: 85.383s total: 85.873s]
[       OK ] ( 8/42) HelloMultiLangTest_c on daint:gpu using cray [compile: 0.132s␣
↪run: 124.678s total: 124.827s]
[       OK ] ( 9/42) HelloThreadedExtended2Test on daint:mc using cray [compile: 0.
↪775s run: 15.569s total: 16.362s]
[       OK ] (10/42) HelloThreadedExtended2Test on daint:mc using intel [compile: 2.
↪814s run: 24.600s total: 27.438s]
[       OK ] (11/42) HelloMultiLangTest_cpp on daint:mc using cray [compile: 0.474s␣
↪run: 70.035s total: 70.528s]
[       OK ] (12/42) HelloMultiLangTest_c on daint:mc using cray [compile: 0.138s␣
↪run: 110.807s total: 110.963s]
[       OK ] (13/42) HelloThreadedExtended2Test on daint:login using builtin␣
↪[compile: 0.790s run: 67.313s total: 68.124s]
[       OK ] (14/42) HelloMultiLangTest_cpp on daint:login using pgi [compile: 1.799s␣
↪run: 100.490s total: 102.683s]
[       OK ] (15/42) HelloMultiLangTest_cpp on daint:login using builtin [compile: 0.
↪497s run: 108.380s total: 108.895s]
[       OK ] (16/42) HelloMultiLangTest_c on daint:login using gnu [compile: 1.337s␣
↪run: 142.017s total: 143.373s]
[       OK ] (17/42) HelloMultiLangTest_cpp on daint:gpu using pgi [compile: 1.851s␣
↪run: 88.935s total: 90.805s]
[       OK ] (18/42) HelloMultiLangTest_cpp on daint:gpu using gnu [compile: 1.640s␣
↪run: 97.855s total: 99.513s]
[       OK ] (19/42) HelloMultiLangTest_c on daint:gpu using intel [compile: 1.578s␣
↪run: 131.689s total: 133.287s]
[       OK ] (20/42) HelloMultiLangTest_cpp on daint:mc using pgi [compile: 1.917s␣
↪run: 73.276s total: 75.213s]
[       OK ] (21/42) HelloMultiLangTest_cpp on daint:mc using gnu [compile: 1.727s␣
↪run: 82.213s total: 83.960s]
[       OK ] (22/42) HelloMultiLangTest_c on daint:mc using intel [compile: 1.573s␣
↪run: 117.806s total: 119.402s]
[       OK ] (23/42) HelloMultiLangTest_cpp on daint:login using gnu [compile: 1.644s␣
↪run: 106.956s total: 108.618s]
[       OK ] (24/42) HelloMultiLangTest_c on daint:login using cray [compile: 0.146s␣
↪run: 137.301s total: 137.466s]
[       OK ] (25/42) HelloMultiLangTest_c on daint:login using intel [compile: 1.613s␣
↪run: 140.058s total: 141.689s]
[       OK ] (26/42) HelloMultiLangTest_c on daint:login using builtin [compile: 0.
↪122s run: 143.692s total: 143.833s]
[       OK ] (27/42) HelloMultiLangTest_c on daint:gpu using pgi [compile: 1.361s␣
↪run: 127.958s total: 129.341s]
```

(continues on next page)

(continued from previous page)

```
[       OK ] (28/42) HelloMultiLangTest_c on daint:gpu using gnu [compile: 1.337s␣
↪run: 136.031s total: 137.386s]
[       OK ] (29/42) HelloMultiLangTest_c on daint:mc using pgi [compile: 1.410s run:␣
↪113.998s total: 115.428s]
[       OK ] (30/42) HelloMultiLangTest_c on daint:mc using gnu [compile: 1.344s run:␣
↪122.086s total: 123.453s]
[       OK ] (31/42) HelloThreadedExtended2Test on daint:login using pgi [compile: 2.
↪733s run: 60.105s total: 62.951s]
[       OK ] (32/42) HelloMultiLangTest_cpp on daint:login using intel [compile: 2.
↪780s run: 104.916s total: 107.716s]
[       OK ] (33/42) HelloThreadedExtended2Test on daint:gpu using pgi [compile: 2.
↪373s run: 39.144s total: 41.545s]
[       OK ] (34/42) HelloMultiLangTest_cpp on daint:gpu using intel [compile: 1.835s␣
↪run: 95.042s total: 96.896s]
[       OK ] (35/42) HelloThreadedExtended2Test on daint:mc using pgi [compile: 2.
↪686s run: 20.751s total: 23.457s]
[       OK ] (36/42) HelloMultiLangTest_cpp on daint:mc using intel [compile: 1.862s␣
↪run: 79.275s total: 81.170s]
[       OK ] (37/42) HelloThreadedExtended2Test on daint:login using gnu [compile: 2.
↪106s run: 67.284s total: 69.409s]
[       OK ] (38/42) HelloThreadedExtended2Test on daint:gpu using gnu [compile: 2.
↪471s run: 56.360s total: 58.871s]
[       OK ] (39/42) HelloThreadedExtended2Test on daint:mc using gnu [compile: 2.
↪007s run: 32.300s total: 34.330s]
[       OK ] (40/42) StreamWithRefTest on daint:login using gnu [compile: 1.941s run:␣
↪14.373s total: 16.337s]
[       OK ] (41/42) StreamWithRefTest on daint:gpu using gnu [compile: 1.954s run:␣
↪11.815s total: 13.791s]
[       OK ] (42/42) StreamWithRefTest on daint:mc using gnu [compile: 2.513s run: 10.
↪672s total: 13.213s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 42 test case(s) from 4 check(s) (0 failure(s))
[==========] Finished on Mon Jan 25 00:37:02 2021
================================================================================
PERFORMANCE REPORT
--------------------------------------------------------------------------------
StreamWithRefTest
- daint:login
   - gnu
      * num_tasks: 1
      * Copy: 72923.3 MB/s
      * Scale: 45663.4 MB/s
      * Add: 49417.7 MB/s
      * Triad: 49426.4 MB/s
- daint:gpu
   - gnu
      * num_tasks: 1
      * Copy: 50638.7 MB/s
      * Scale: 35186.0 MB/s
      * Add: 38564.4 MB/s
      * Triad: 38771.1 MB/s
- daint:mc
   - gnu
      * num_tasks: 1
      * Copy: 19072.5 MB/s
      * Scale: 10395.6 MB/s
```

(continues on next page)

(continued from previous page)

```
      * Add: 11041.0 MB/s
      * Triad: 11079.2 MB/s
------------------------------------------------------------------------------
Log file(s) saved in: '/tmp/rfm-r4yjva71.log'
```

There it is! Without any change in our tests, we could simply run them in a HPC cluster with all of its intricacies. Notice how our original four tests expanded to more than 40 test cases on that particular HPC cluster! One reason we could run immediately our tests on a new system was that we have not been restricting neither the valid system they can run nor the valid programming environments they can run with (except for the STREAM test). Otherwise we would have to add `daint` and its corresponding programming environments in `valid_systems` and `valid_prog_environs` lists respectively.

Tip: A quick way to try a test on a new system, if it's not generic, is to pass the *`--skip-system-check`* and the *`--skip-prgenv-check`* command line options which will cause ReFrame to skip any test validity checks for systems or programming environments.

Although the tests remain the same, ReFrame has generated completely different job scripts for each test depending on where it was going to run. Let's check the job script generated for the `StreamWithRefTest`:

```
cat output/daint/gpu/gnu/StreamWithRefTest/rfm_StreamWithRefTest_job.sh
```

```
#!/bin/bash
#SBATCH --job-name="rfm_StreamWithRefTest_job"
#SBATCH --ntasks=1
#SBATCH --output=rfm_StreamWithRefTest_job.out
#SBATCH --error=rfm_StreamWithRefTest_job.err
#SBATCH --time=0:10:0
#SBATCH -A csstaff
#SBATCH --constraint=gpu
module unload PrgEnv-cray
module load PrgEnv-gnu
export OMP_NUM_THREADS=4
export OMP_PLACES=cores
srun ./StreamWithRefTest
```

Whereas the exact same test running on our laptop was as simple as the following:

```
#!/bin/bash
export OMP_NUM_THREADS=4
export OMP_PLACES=cores
 ./StreamWithRefTest
```

In ReFrame, you don't have to care about all the system interaction details, but rather about the logic of your tests as we shall see in the next section.

Adapting a test to new systems and programming environments

Unless a test is rather generic, you will need to make some adaptations for the system that you port it to. In this case, we will adapt the STREAM benchmark so as to run it with multiple compiler and adjust its execution based on the target architecture of each partition. Let's see and comment the changes:

```
cat tutorials/basics/stream/stream3.py
```

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


@rfm.simple_test
class StreamMultiSysTest(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['cray', 'gnu', 'intel', 'pgi']
    prebuild_cmds = [
        'wget http://www.cs.virginia.edu/stream/FTP/Code/stream.c',
    ]
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    variables = {
        'OMP_NUM_THREADS': '4',
        'OMP_PLACES': 'cores'
    }
    reference = {
        'catalina': {
            'Copy':  (25200, -0.05, 0.05, 'MB/s'),
            'Scale': (16800, -0.05, 0.05, 'MB/s'),
            'Add':   (18500, -0.05, 0.05, 'MB/s'),
            'Triad': (18800, -0.05, 0.05, 'MB/s')
        }
    }

    # Flags per programming environment
    flags = variable(dict, value={
        'cray':  ['-fopenmp', '-O3', '-Wall'],
        'gnu':   ['-fopenmp', '-O3', '-Wall'],
        'intel': ['-qopenmp', '-O3', '-Wall'],
        'pgi':   ['-mp', '-O3']
    })

    # Number of cores for each system
    cores = variable(dict, value={
        'catalina:default': 4,
        'daint:gpu': 12,
        'daint:mc': 36,
        'daint:login': 10
    })

    @rfm.run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = ['-DSTREAM_ARRAY_SIZE=$((1 << 25))']
        environ = self.current_environ.name
        self.build_system.cflags = self.flags.get(environ, [])

    @rfm.run_before('run')
```

(continues on next page)

(continued from previous page)

```
    def set_num_threads(self):
        num_threads = self.cores.get(self.current_partition.fullname, 1)
        self.num_cpus_per_task = num_threads
        self.variables = {
            'OMP_NUM_THREADS': str(num_threads),
            'OMP_PLACES': 'cores'
        }

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(r'Solution Validates',
                                               self.stdout)

    @rfm.run_before('performance')
    def set_perf_patterns(self):
        self.perf_patterns = {
            'Copy': sn.extractsingle(r'Copy:\s+(\S+)\s+.*',
                                     self.stdout, 1, float),
            'Scale': sn.extractsingle(r'Scale:\s+(\S+)\s+.*',
                                      self.stdout, 1, float),
            'Add': sn.extractsingle(r'Add:\s+(\S+)\s+.*',
                                    self.stdout, 1, float),
            'Triad': sn.extractsingle(r'Triad:\s+(\S+)\s+.*',
                                      self.stdout, 1, float)
        }
```

First of all, we need to add the new programming environments in the list of the supported ones. Now there is the problem that each compiler has its own flags for enabling OpenMP, so we need to differentiate the behavior of the test based on the programming environment. For this reason, we define the flags for each compiler in a separate dictionary (`flags` variable) and we set them in the `set_compiler_flags()` pipeline hook. We have first seen the pipeline hooks in the multithreaded "Hello, World!" example and now we explain them in more detail. When ReFrame loads a test file, it instantiates all the tests it finds in it. Based on the system ReFrame runs on and the supported environments of the tests, it will generate different test cases for each system partition and environment combination and it will finally send the test cases for execution. During its execution, a test case goes through the *regression test pipeline*, which is a series of well defined phases. Users can attach arbitrary functions to run before or after any pipeline stage and this is exactly what the `set_compiler_flags()` function is. We instruct ReFrame to run this function before the test enters the `compile` stage and set accordingly the compilation flags. The system partition and the programming environment of the currently running test case are available to a ReFrame test through the *`current_partition`* and *`current_environ`* attributes respectively. These attributes, however, are only set after the first stage (`setup`) of the pipeline is executed, so we can't use them inside the test's constructor.

We do exactly the same for setting the `OMP_NUM_THREADS` environment variables depending on the system partition we are running on, by attaching the `set_num_threads()` pipeline hook to the `run` phase of the test. In that same hook we also set the *`num_cpus_per_task`* attribute of the test, so as to instruct the backend job scheduler to properly assign CPU cores to the test. In ReFrame tests you can set a series of task allocation attributes that will be used by the backend schedulers to emit the right job submission script. The section *Mapping of Test Attributes to Job Scheduler Backends* of the *Regression Tests API* summarizes these attributes and the actual backend scheduler options that they correspond to.

For more information about the regression test pipeline and how ReFrame executes the tests in general, have a look at *How ReFrame Executes Tests*.

Note: ReFrame tests are ordinary Python classes so you can define your own attributes as we do with `flags` and `cores` in this example.

Let's run our adapted test now:

```
./bin/reframe -c tutorials/basics/stream/stream3.py -r --performance-report
```

```
[ReFrame Setup]
  version:           3.3-dev0 (rev: cb974c13)
  command:           './bin/reframe -C tutorials/config/settings.py -c tutorials/
↪basics/stream/stream3.py -r --performance-report'
  launched by:       user@dom101
  working directory: '/users/user/Devel/reframe'
  settings file:     'tutorials/config/settings.py'
  check search path: '/users/user/Devel/reframe/tutorials/basics/stream/stream3.py'
  stage directory:   '/users/user/Devel/reframe/stage'
  output directory:  '/users/user/Devel/reframe/output'

[==========] Running 1 check(s)
[==========] Started on Mon Oct 12 20:16:03 2020

[----------] started processing StreamMultiSysTest (StreamMultiSysTest)
[ RUN      ] StreamMultiSysTest on daint:login using gnu
[ RUN      ] StreamMultiSysTest on daint:login using intel
[ RUN      ] StreamMultiSysTest on daint:login using pgi
[ RUN      ] StreamMultiSysTest on daint:login using cray
[ RUN      ] StreamMultiSysTest on daint:gpu using gnu
[ RUN      ] StreamMultiSysTest on daint:gpu using intel
[ RUN      ] StreamMultiSysTest on daint:gpu using pgi
[ RUN      ] StreamMultiSysTest on daint:gpu using cray
[ RUN      ] StreamMultiSysTest on daint:mc using gnu
[ RUN      ] StreamMultiSysTest on daint:mc using intel
[ RUN      ] StreamMultiSysTest on daint:mc using pgi
[ RUN      ] StreamMultiSysTest on daint:mc using cray
[----------] finished processing StreamMultiSysTest (StreamMultiSysTest)

[----------] waiting for spawned checks to finish
[       OK ] ( 1/12) StreamMultiSysTest on daint:gpu using pgi [compile: 2.092s run:␣
↪11.201s total: 13.307s]
[       OK ] ( 2/12) StreamMultiSysTest on daint:gpu using gnu [compile: 2.349s run:␣
↪17.140s total: 19.509s]
[       OK ] ( 3/12) StreamMultiSysTest on daint:login using pgi [compile: 2.230s␣
↪run: 20.946s total: 23.189s]
[       OK ] ( 4/12) StreamMultiSysTest on daint:login using gnu [compile: 2.161s␣
↪run: 27.093s total: 29.266s]
[       OK ] ( 5/12) StreamMultiSysTest on daint:mc using gnu [compile: 1.954s run: 7.
↪904s total: 9.870s]
[       OK ] ( 6/12) StreamMultiSysTest on daint:gpu using intel [compile: 2.286s␣
↪run: 14.686s total: 16.984s]
[       OK ] ( 7/12) StreamMultiSysTest on daint:login using intel [compile: 2.520s␣
↪run: 24.427s total: 26.960s]
[       OK ] ( 8/12) StreamMultiSysTest on daint:mc using intel [compile: 2.312s run:␣
↪5.350s total: 7.678s]
[       OK ] ( 9/12) StreamMultiSysTest on daint:gpu using cray [compile: 0.672s run:␣
↪10.791s total: 11.476s]
[       OK ] (10/12) StreamMultiSysTest on daint:login using cray [compile: 0.706s␣
↪run: 20.505s total: 21.229s]
[       OK ] (11/12) StreamMultiSysTest on daint:mc using cray [compile: 0.674s run:␣
↪2.763s total: 3.453s]
[       OK ] (12/12) StreamMultiSysTest on daint:mc using pgi [compile: 2.088s run: 5.
↪124s total: 7.224s]
```

(continues on next page)

(continued from previous page)

```
[----------] all spawned checks have finished

[  PASSED  ] Ran 12 test case(s) from 1 check(s) (0 failure(s))
[==========] Finished on Mon Oct 12 20:16:36 2020
================================================================================
PERFORMANCE REPORT
--------------------------------------------------------------------------------
StreamMultiSysTest
- daint:login
   - gnu
      * num_tasks: 1
      * Copy: 95784.6 MB/s
      * Scale: 73747.3 MB/s
      * Add: 79138.3 MB/s
      * Triad: 81253.3 MB/s
   - intel
      * num_tasks: 1
      * Copy: 103540.5 MB/s
      * Scale: 109257.6 MB/s
      * Add: 112189.8 MB/s
      * Triad: 113440.8 MB/s
   - pgi
      * num_tasks: 1
      * Copy: 99071.7 MB/s
      * Scale: 74721.3 MB/s
      * Add: 81206.4 MB/s
      * Triad: 78328.9 MB/s
   - cray
      * num_tasks: 1
      * Copy: 96664.5 MB/s
      * Scale: 75637.4 MB/s
      * Add: 74759.3 MB/s
      * Triad: 73450.6 MB/s
- daint:gpu
   - gnu
      * num_tasks: 1
      * Copy: 42293.7 MB/s
      * Scale: 38095.1 MB/s
      * Add: 43080.7 MB/s
      * Triad: 43719.2 MB/s
   - intel
      * num_tasks: 1
      * Copy: 52563.0 MB/s
      * Scale: 54316.5 MB/s
      * Add: 59044.5 MB/s
      * Triad: 59165.5 MB/s
   - pgi
      * num_tasks: 1
      * Copy: 50710.5 MB/s
      * Scale: 39639.5 MB/s
      * Add: 44104.5 MB/s
      * Triad: 44143.7 MB/s
   - cray
      * num_tasks: 1
      * Copy: 51159.8 MB/s
      * Scale: 39176.0 MB/s
      * Add: 43588.8 MB/s
```

(continues on next page)

(continued from previous page)

```
      * Triad: 43866.8 MB/s
- daint:mc
   - gnu
      * num_tasks: 1
      * Copy: 48744.5 MB/s
      * Scale: 38774.7 MB/s
      * Add: 43760.0 MB/s
      * Triad: 44143.1 MB/s
   - intel
      * num_tasks: 1
      * Copy: 52707.0 MB/s
      * Scale: 49011.8 MB/s
      * Add: 57513.3 MB/s
      * Triad: 57678.3 MB/s
   - pgi
      * num_tasks: 1
      * Copy: 46274.3 MB/s
      * Scale: 40628.6 MB/s
      * Add: 44352.4 MB/s
      * Triad: 44630.2 MB/s
   - cray
      * num_tasks: 1
      * Copy: 46912.5 MB/s
      * Scale: 40076.9 MB/s
      * Add: 43639.0 MB/s
      * Triad: 44068.3 MB/s
------------------------------------------------------------------------------
Log file(s) saved in: '/tmp/rfm-odx7qewe.log'
```

Notice the improved performance of the benchmark in all partitions and the differences in performance between the different compilers.

This concludes our introductory tutorial to ReFrame!

2.2.2 Tutorial 2: Customizing Further a Regression Test

In this tutorial we will present common patterns that can come up when writing regression tests with ReFrame. All examples use the configuration file presented in *Tutorial 1: Getting Started with ReFrame*, which you can find in `tutorials/config/settings.py`. We also assume that the reader is already familiar with the concepts presented in the basic tutorial. Finally, to avoid specifying the tutorial configuration file each time, make sure to export it here:

```
export RFM_CONFIG_FILE=$(pwd)/tutorials/config/mysettings.py
```

Parameterizing a Regression Test

We have briefly looked into parameterized tests in *Tutorial 1: Getting Started with ReFrame* where we parameterized the "Hello, World!" test based on the programming language. Test parameterization in ReFrame is quite powerful since it allows you to create a multitude of similar tests automatically. In this example, we will parameterize the last version of the STREAM test from the *Tutorial 1: Getting Started with ReFrame* by changing the array size, so as to check the bandwidth of the different cache levels. Here is the adapted code with the relevant parts highlighted (for simplicity, we are interested only in the "Triad" benchmark):

```
cat tutorials/advanced/parameterized/stream.py
```

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


@rfm.simple_test
class StreamMultiSysTest(rfm.RegressionTest):
    num_bytes = parameter(1 << pow for pow in range(19, 30))
    array_size = variable(int)
    ntimes = variable(int)

    valid_systems = ['*']
    valid_prog_environs = ['cray', 'gnu', 'intel', 'pgi']
    prebuild_cmds = [
        'wget http://www.cs.virginia.edu/stream/FTP/Code/stream.c',
    ]
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    variables = {
        'OMP_NUM_THREADS': '4',
        'OMP_PLACES': 'cores'
    }
    reference = {
        '*': {
            'Triad': (0, None, None, 'MB/s'),
        }
    }

    # Flags per programming environment
    flags = variable(dict, value={
        'cray':  ['-fopenmp', '-O3', '-Wall'],
        'gnu':   ['-fopenmp', '-O3', '-Wall'],
        'intel': ['-qopenmp', '-O3', '-Wall'],
        'pgi':   ['-mp', '-O3']
    })

    # Number of cores for each system
    cores = variable(dict, value={
        'catalina:default': 4,
        'daint:gpu': 12,
        'daint:mc': 36,
        'daint:login': 10
    })

    @rfm.run_after('init')
    def set_variables(self):
        self.array_size = (self.num_bytes >> 3) // 3
        self.ntimes = 100*1024*1024 // self.array_size
        self.descr = (
            f'STREAM test (array size: {self.array_size}, '
            f'ntimes: {self.ntimes})'
        )

    @rfm.run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = [f'-DSTREAM_ARRAY_SIZE={self.array_size}',
```

(continues on next page)

(continued from previous page)

```
                                        f'-DNTIMES={self.ntimes}']
        environ = self.current_environ.name
        self.build_system.cflags = self.flags.get(environ, [])

    @rfm.run_before('run')
    def set_num_threads(self):
        num_threads = self.cores.get(self.current_partition.fullname, 1)
        self.num_cpus_per_task = num_threads
        self.variables = {
            'OMP_NUM_THREADS': str(num_threads),
            'OMP_PLACES': 'cores'
        }

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(r'Solution Validates',
                                               self.stdout)

    @rfm.run_before('performance')
    def set_perf_patterns(self):
        self.perf_patterns = {
            'Triad': sn.extractsingle(r'Triad:\s+(\S+)\s+.*',
                                      self.stdout, 1, float),
        }
```

Any ordinary ReFrame test becomes a parameterized one if the user defines parameters inside the class body of the test. This is done using the *`parameter()`* ReFrame built-in function, which accepts the list of parameter values. For each parameter value ReFrame will instantiate a different regression test by assigning the corresponding value to an attribute named after the parameter. So in this example, ReFrame will generate automatically 11 tests with different values for their `num_bytes` attribute. From this point on, you can adapt the test based on the parameter values, as we do in this case, where we compute the STREAM array sizes, as well as the number of iterations to be performed on each benchmark, and we also compile the code accordingly.

Let's try listing the generated tests:

```
./bin/reframe -c tutorials/advanced/parameterized/stream.py -l
```

```
[ReFrame Setup]
  version:           3.6.0-dev.0+2f8e5b3b
  command:           './bin/reframe -c tutorials/advanced/parameterized/stream.py -l'
  launched by:       user@tresa.local
  working directory: '/Users/user/Repositories/reframe'
  settings file:     'tutorials/config/settings.py'
  check search path: '/Users/user/Repositories/reframe/tutorials/advanced/
↪parameterized/stream.py'
  stage directory:   '/Users/user/Repositories/reframe/stage'
  output directory:  '/Users/user/Repositories/reframe/output'

[List of matched checks]
- StreamMultiSysTest_2097152 (found in '/Users/user/Repositories/reframe/tutorials/
↪advanced/parameterized/stream.py')
- StreamMultiSysTest_67108864 (found in '/Users/user/Repositories/reframe/tutorials/
↪advanced/parameterized/stream.py')
- StreamMultiSysTest_1048576 (found in '/Users/user/Repositories/reframe/tutorials/
↪advanced/parameterized/stream.py')
- StreamMultiSysTest_536870912 (found in '/Users/user/Repositories/reframe/tutorials/
↪advanced/parameterized/stream.py')
```

(continues on next page)

(continued from previous page)

```
- StreamMultiSysTest_4194304 (found in '/Users/user/Repositories/reframe/tutorials/
 ↪advanced/parameterized/stream.py')
- StreamMultiSysTest_33554432 (found in '/Users/user/Repositories/reframe/tutorials/
 ↪advanced/parameterized/stream.py')
- StreamMultiSysTest_8388608 (found in '/Users/user/Repositories/reframe/tutorials/
 ↪advanced/parameterized/stream.py')
- StreamMultiSysTest_268435456 (found in '/Users/user/Repositories/reframe/tutorials/
 ↪advanced/parameterized/stream.py')
- StreamMultiSysTest_16777216 (found in '/Users/user/Repositories/reframe/tutorials/
 ↪advanced/parameterized/stream.py')
- StreamMultiSysTest_524288 (found in '/Users/user/Repositories/reframe/tutorials/
 ↪advanced/parameterized/stream.py')
- StreamMultiSysTest_134217728 (found in '/Users/user/Repositories/reframe/tutorials/
 ↪advanced/parameterized/stream.py')
Found 11 check(s)

Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-s_ty1l50.
 ↪log'
```

ReFrame generates 11 tests from the single parameterized test that we have written and names them by appending a string representation of the parameter value.

Test parameterization in ReFrame is very powerful since you can parameterize your tests on anything and you can create complex parameterization spaces. A common pattern is to parameterize a test on the environment module that loads a software in order to test different versions of it. For this reason, ReFrame offers the *find_modules()* function, which allows you to parameterize a test on the available modules for a given programming environment and partition combination. The following example will create a test for each GROMACS module found on the software stack associated with a system partition and programming environment (toolchain):

```
import reframe as rfm
import reframe.utility as util


@rfm.simple_test
class MyTest(rfm.RegressionTest):
    module_info = parameter(util.find_modules('GROMACS'))

    @rfm.run_after('init')
    def process_module_info(self):
        s, e, m = self.module_info
        self.valid_systems = [s]
        self.valid_prog_environs = [e]
        self.modules = [m]
```

More On Building Tests

We have already seen how ReFrame can compile a test with a single source file. However, ReFrame can also build tests that use Make or a configure-Make approach. We are going to demonstrate this through a simple C++ program that computes a dot-product of two vectors and is being compiled through a Makefile. Additionally, we can select the type of elements for the vectors at compilation time. Here is the C++ program:

```
cat tutorials/advanced/makefiles/src/dotprod.cpp
```

```
#include <cassert>
#include <iostream>
#include <random>
#include <vector>

#ifndef ELEM_TYPE
#define ELEM_TYPE double
#endif

using elem_t = ELEM_TYPE;

template<typename T>
T dotprod(const std::vector<T> &x, const std::vector<T> &y)
{
    assert(x.size() == y.size());
    T sum = 0;
    for (std::size_t i = 0; i < x.size(); ++i) {
        sum += x[i] * y[i];
    }

    return sum;
}

template<typename T>
struct type_name {
    static constexpr const char *value = nullptr;
};

template<>
struct type_name<float> {
    static constexpr const char *value = "float";
};

template<>
struct type_name<double> {
    static constexpr const char *value = "double";
};

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cerr << argv[0] << ": too few arguments\n";
        std::cerr << "Usage: " << argv[0] << " DIM\n";
        return 1;
    }

    std::size_t N = std::atoi(argv[1]);
    if (N < 0) {
        std::cerr << argv[0]
                  << ": array dimension must a positive integer: " << argv[1]
                  << "\n";
        return 1;
    }

    std::vector<elem_t> x(N), y(N);
    std::random_device seed;
    std::mt19937 rand(seed());
```

(continues on next page)

(continued from previous page)

```
    std::uniform_real_distribution<> dist(-1, 1);
    for (std::size_t i = 0; i < N; ++i) {
        x[i] = dist(rand);
        y[i] = dist(rand);
    }

    std::cout << "Result (" << type_name<elem_t>::value << "): "
              << dotprod(x, y) << "\n";
    return 0;
}
```

The directory structure for this test is the following:

```
tutorials/makefiles/
├── maketest.py
└── src
    ├── Makefile
    └── dotprod.cpp
```

Let's have a look at the test itself:

```
cat tutorials/advanced/makefiles/maketest.py
```

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


@rfm.simple_test
class MakefileTest(rfm.RegressionTest):
    elem_type = parameter(['float', 'double'])

    descr = 'Test demonstrating use of Makefiles'
    valid_systems = ['*']
    valid_prog_environs = ['clang', 'gnu']
    executable = './dotprod'
    executable_opts = ['100000']
    build_system = 'Make'

    @rfm.run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = [f'-DELEM_TYPE={self.elem_type}']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(
            rf'Result \({self.elem_type}\):', self.stdout
        )
```

First, if you're using any build system other than `SingleSource`, you must set the `executable` attribute of the test, because ReFrame cannot know what is the actual executable to be run. We then set the build system to *`Make`* and set the preprocessor flags as we would do with the `SingleSource` build system.

Let's inspect the build script generated by ReFrame:

```
./bin/reframe -c tutorials/advanced/makefiles/maketest.py -r
cat output/catalina/default/clang/MakefileTest_float/rfm_MakefileTest_build.sh
```

```
#!/bin/bash

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

trap _onerror ERR

make -j 1 CPPFLAGS="-DELEM_TYPE=float"
```

The compiler variables (`CC`, `CXX` etc.) are set based on the corresponding values specified in the configuration of the current environment. We can instruct the build system to ignore the default values from the environment by setting its `flags_from_environ` attribute to false:

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

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

```
make -j 1 CPPFLAGS="-DELEM_TYPE=float"
```

Notice that the `-j 1` option is always generated. We can increase the build concurrency by setting the *`max_concurrency`* attribute. Finally, we may even use a custom Makefile by setting the *`makefile`* attribute:

```
self.build_system.max_concurrency = 4
self.build_system.makefile = 'Makefile_custom'
```

As a final note, as with the `SingleSource` build system, it wouldn't have been necessary to specify one in this test, if we wouldn't have to set the CPPFLAGS. ReFrame could automatically figure out the correct build system if *`sourcepath`* refers to a directory. ReFrame will inspect the directory and it will first try to determine whether this is a CMake or Autotools-based project.

More details on ReFrame's build systems can be found 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 a repository using the *`prebuild_cmds`*:

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

Alternatively, we can retrieve specifically a Git repository by assigning 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 proceed with the build procedure as usual.

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).

Adding 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 will then have the following lines:

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

Writing a Run-Only Regression Test

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

```
cat tutorials/advanced/runonly/echorand.py
```

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


@rfm.simple_test
class EchoRandTest(rfm.RunOnlyRegressionTest):
    descr = 'A simple test that echoes a random number'
    valid_systems = ['*']
    valid_prog_environs = ['*']
    lower = variable(int, value=90)
    upper = variable(int, value=100)
    executable = 'echo'
    executable_opts = [
        'Random: ',
        f'$((RANDOM%({upper}+1-{lower})+{lower}))'
    ]

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_bounded(
            sn.extractsingle(
                r'Random: (?P<number>\S+)', self.stdout, 'number', float
```

(continues on next page)

(continued from previous page)

```
            ),
            self.lower, self.upper
        )
```

There is nothing special for this test compared to those presented so far except that it derives from the *RunOnlyRegressionTest*. Run-only regression tests may also have resources, as for instance a pre-compiled executable or some input data. These resources may reside under the `src/` directory or under any directory specified in the *sourcesdir* attribute. These resources will be copied to the stage directory at the beginning of the run phase.

Writing 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 test is a compile-only version of the `MakefileTest` presented *previously* which checks that no warnings are issued by the compiler:

```
cat tutorials/advanced/makefiles/maketest.py
```

```
@rfm.simple_test
class MakeOnlyTest(rfm.CompileOnlyRegressionTest):
    elem_type = parameter(['float', 'double'])
    descr = 'Test demonstrating use of Makefiles'
    valid_systems = ['*']
    valid_prog_environs = ['clang', 'gnu']
    build_system = 'Make'

    @rfm.run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = [f'-DELEM_TYPE={self.elem_type}']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_not_found(r'warning', self.stdout)
```

What is worth noting here is that the standard output and standard error of the test, which are accessible through the *stdout* and *stderr* attributes, correspond now to the standard output and error of the compilation command. Therefore sanity checking can be done in exactly the same way as with a normal test.

Grouping parameter packs

New in version 3.4.2.

In the dot product example shown above, we had two independent tests that defined the same `elem_type` parameter. And the two tests cannot have a parent-child relationship, since one of them is a run-only test and the other is a compile-only one. ReFrame offers the *RegressionMixin* class that allows you to group parameters and other builtins that are meant to be reused over otherwise unrelated tests. In the example below, we create an `ElemTypeParam` mixin that holds the definition of the `elem_type` parameter which is inherited by both the concrete test classes:

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


class ElemTypeParam(rfm.RegressionMixin):
```

(continues on next page)

(continued from previous page)

```
    elem_type = parameter(['float', 'double'])


@rfm.simple_test
class MakefileTestAlt(rfm.RegressionTest, ElemTypeParam):
    descr = 'Test demonstrating use of Makefiles'
    valid_systems = ['*']
    valid_prog_environs = ['clang', 'gnu']
    executable = './dotprod'
    executable_opts = ['100000']
    build_system = 'Make'

    @rfm.run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = [f'-DELEM_TYPE={self.elem_type}']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(
            rf'Result \({self.elem_type}\):', self.stdout
        )


@rfm.simple_test
class MakeOnlyTestAlt(rfm.CompileOnlyRegressionTest, ElemTypeParam):
    descr = 'Test demonstrating use of Makefiles'
    valid_systems = ['*']
    valid_prog_environs = ['clang', 'gnu']
    build_system = 'Make'

    @rfm.run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = [f'-DELEM_TYPE={self.elem_type}']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_not_found(r'warning', self.stdout)
```

Notice how the parameters are expanded in each of the individual tests:

```
./bin/reframe -c tutorials/advanced/makefiles/maketest_mixin.py -l
```

```
[ReFrame Setup]
  version:           3.6.0-dev.0+2f8e5b3b
  command:           './bin/reframe -c tutorials/advanced/makefiles/maketest_mixin.py␣
↪-l'
  launched by:       user@tresa.local
  working directory: '/Users/user/Repositories/reframe'
  settings file:     'tutorials/config/settings.py'
  check search path: '/Users/user/Repositories/reframe/tutorials/advanced/makefiles/
↪maketest_mixin.py'
  stage directory:   '/Users/user/Repositories/reframe/stage'
  output directory:  '/Users/user/Repositories/reframe/output'

[List of matched checks]
- MakeOnlyTestAlt_double (found in '/Users/user/Repositories/reframe/tutorials/
↪advanced/makefiles/maketest_mixin.py')
```

(continues on next page)

(continued from previous page)

```
- MakeOnlyTestAlt_float (found in '/Users/user/Repositories/reframe/tutorials/
↪advanced/makefiles/maketest_mixin.py')
- MakefileTestAlt_double (found in '/Users/user/Repositories/reframe/tutorials/
↪advanced/makefiles/maketest_mixin.py')
- MakefileTestAlt_float (found in '/Users/user/Repositories/reframe/tutorials/
↪advanced/makefiles/maketest_mixin.py')
Found 4 check(s)

Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-e384bvkd.
↪log'
```

Applying a Sanity Function Iteratively

It is often the case that a common sanity pattern has to be applied many times. The following script prints 100 random integers between the limits given by the environment variables `LOWER` and `UPPER`.

```
cat tutorials/advanced/random/src/random_numbers.sh
```

```
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
```

In the corresponding regression test we want to check that all the random numbers generated lie between the two limits, which means that a common sanity check has to be applied to all the printed random numbers. Here is the corresponding regression test:

```
cat tutorials/advanced/random/randint.py
```

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


@rfm.simple_test
class DeferredIterationTest(rfm.RunOnlyRegressionTest):
    descr = 'Apply a sanity function iteratively'
    valid_systems = ['*']
    valid_prog_environs = ['*']
    executable = './random_numbers.sh'

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        numbers = sn.extractall(
            r'Random: (?P<number>\S+)', self.stdout, 'number', float
        )
        self.sanity_patterns = sn.all([
            sn.assert_eq(sn.count(numbers), 100),
```

(continues on next page)

(continued from previous page)

```
            sn.all(sn.map(lambda x: sn.assert_bounded(x, 90, 100), numbers))
        ])
```

First, we extract all the generated random numbers from the output. What we want to do is to apply iteratively the *assert_bounded()* sanity function for each number. The problem here is that we cannot simply iterate over the `numbers` list, because that would trigger prematurely the evaluation of the *extractall()*. We want to defer also the iteration. This can be achieved by using the *map()* ReFrame sanity function, which is a replacement of Python's built-in `map()` function and does exactly what we want: it applies a function on all the elements of an iterable and returns another iterable with the transformed elements. Passing the result of the `map()` function to the *all()* sanity function ensures that all the elements lie between the desired bounds.

There is still a small complication that needs to be addressed. As a direct replacement of the built-in `all()` function, ReFrame's *all()* sanity function returns `True` for empty iterables, which is not what we want. So we must make sure that all 100 numbers are generated. This is achieved by the `sn.assert_eq(sn.count(numbers), 100)` statement, which uses the *count()* sanity function for counting the generated numbers. Finally, we need to combine these two conditions to a single deferred expression that will be assigned to the test's `sanity_patterns`. We accomplish this by using the *all()* sanity function.

For more information about how exactly sanity functions work and how their execution is deferred, please refer to *Understanding the Mechanism of Sanity Functions*.

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 Test Job Script

It is often the case that we need to run some commands before or after the parallel launch of our 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 random numbers test presented *above*. The lower and upper limits for the random numbers are now set inside a helper shell script in `limits.sh` located in the test's resources, which we need to source before running our tests. Additionally, we want also to print `FINISHED` after our executable has finished. Here is the modified test file:

```
cat tutorials/advanced/random/prepostrun.py
```

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


@rfm.simple_test
class PrepostRunTest(rfm.RunOnlyRegressionTest):
    descr = 'Pre- and post-run demo test'
    valid_systems = ['*']
    valid_prog_environs = ['*']
    prerun_cmds = ['source limits.sh']
    postrun_cmds = ['echo FINISHED']
    executable = './random_numbers.sh'

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

(continues on next page)

(continued from previous page)

```
        )
        self.sanity_patterns = sn.all([
            sn.assert_eq(sn.count(numbers), 100),
            sn.all(sn.map(lambda x: sn.assert_bounded(x, 90, 100), numbers)),
            sn.assert_found(r'FINISHED', self.stdout)
        ])
```

The `prerun_cmds` and `postrun_cmds` are lists of commands to be emitted in the generated job script before and after the parallel launch of the executable. Obviously, the working directory for these commands is that of the job script itself, which is the stage directory of the test. The generated job script for this test looks like the following:

```
./bin/reframe -c tutorials/advanced/random/prepostrun.py -r
cat output/catalina/default/gnu/PrepostRunTest/rfm_PrepostRunTest_job.sh
```

```
#!/bin/bash
source limits.sh
 ./random_numbers.sh
echo FINISHED
```

Generally, ReFrame generates the job shell scripts using the following pattern:

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

The `job_scheduler_preamble` contains the backend job scheduler directives that control the job allocation. The `prepare_cmds` are commands that can be emitted before the test environment commands. These can be specified with the `prepare_cmds` partition configuration option. The `env_load_cmds` are the necessary commands for setting up the environment of the test. These include any modules or environment variables set at the system partition level or any modules or environment variables set at the test level. 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.

Adding job scheduler options per test

Sometimes a test needs to pass additional job scheduler options to the automatically generated job script. This is fairly easy to achieve with ReFrame. In the following test we want to test whether the `--mem` option of Slurm works as expected. We compiled and ran a program that consumes all the available memory of the node, but we want to restrict the available memory with the `--mem` option. Here is the test:

```
cat tutorials/advanced/jobopts/eatmemory.py
```

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

(continues on next page)

(continued from previous page)

```
@rfm.simple_test
class MemoryLimitTest(rfm.RegressionTest):
    valid_systems = ['daint:gpu', 'daint:mc']
    valid_prog_environs = ['gnu']
    sourcepath = 'eatmemory.c'
    executable_opts = ['2000M']

    @rfm.run_before('run')
    def set_memory_limit(self):
        self.job.options = ['--mem=1000']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(
```

Each ReFrame test has an associated run job descriptor which represents the scheduler job that will be used to run this test. This object has an `options` attribute, which can be used to pass arbitrary options to the scheduler. The job descriptor is initialized by the framework during the setup pipeline phase. For this reason, we cannot directly set the job options inside the test constructor and we have to use a pipeline hook that runs before running (i.e., submitting the test).

Let's run the test and inspect the generated job script:

```
./bin/reframe -c tutorials/advanced/jobopts/eatmemory.py -n MemoryLimitTest -r
cat output/daint/gpu/gnu/MemoryLimitTest/rfm_MemoryLimitTest_job.sh
```

```
#!/bin/bash
#SBATCH --job-name="rfm_MemoryLimitTest_job"
#SBATCH --ntasks=1
#SBATCH --output=rfm_MemoryLimitTest_job.out
#SBATCH --error=rfm_MemoryLimitTest_job.err
#SBATCH --time=0:10:0
#SBATCH -A csstaff
#SBATCH --constraint=gpu
#SBATCH --mem=1000
module unload PrgEnv-cray
module load PrgEnv-gnu
srun ./MemoryLimitTest 2000M
```

The job options specified inside a ReFrame test are always the last to be emitted in the job script preamble and do not affect the options that are passed implicitly through other test attributes or configuration options.

There is a small problem with this test though. What if we change the job scheduler in that partition or what if we want to port the test to a different system that does not use Slurm and and another option is needed to achieve the same result. The obvious answer is to adapt the test, but is there a more portable way? The answer is yes and this can be achieved through so-called *extra resources*. ReFrame gives you the possibility to associate scheduler options to a "resource" managed by the partition scheduler. You can then use those resources transparently from within your test.

To achieve this in our case, we first need to define a `memory` resource in the configuration:

```
            'partitions': [
                {
                    'name': 'login',
                    'descr': 'Login nodes',
```

(continues on next page)

(continued from previous page)

```
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['builtin', 'gnu', 'intel', 'pgi', 'cray'],
                },
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C gpu', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                    'resources': [
                        {
                            'name': 'memory',
                            'options': ['--mem={size}']
                        }
                    ],
                },
                {
                    'name': 'mc',
                    'descr': 'Multicore nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C mc', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                    'resources': [
                        {
                            'name': 'memory',
                            'options': ['--mem={size}']
                        }
                    ]
                }
            ]
```

Notice that we do not define the resource for all the partitions, but only for those that it makes sense. Each resource has a name and a set of scheduler options that will be passed to the scheduler when this resource will be requested by the test. The options specification can contain placeholders, whose value will also be set from the test. Let's see how we can rewrite the `MemoryLimitTest` using the `memory` resource instead of passing the `--mem` scheduler option explicitly.

```
cat tutorials/advanced/jobopts/eatmemory.py
```

```
@rfm.simple_test
class MemoryLimitWithResourcesTest(rfm.RegressionTest):
    valid_systems = ['daint:gpu', 'daint:mc']
    valid_prog_environs = ['gnu']
    sourcepath = 'eatmemory.c'
    executable_opts = ['2000M']
    extra_resources = {
        'memory': {'size': '1000'}
    }

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
```

(continues on next page)

(continued from previous page)

```
        self.sanity_patterns = sn.assert_found(
            r'(exceeded memory limit)|(Out Of Memory)', self.stderr
        )
```

The extra resources that the test needs to obtain through its scheduler are specified in the *extra_resources* attribute, which is a dictionary with the resource names as its keys and another dictionary assigning values to the resource placeholders as its values. As you can see, this syntax is completely scheduler-agnostic. If the requested resource is not defined for the current partition, it will be simply ignored.

You can now run and verify that the generated job script contains the `--mem` option:

```
./bin/reframe -c tutorials/advanced/jobopts/eatmemory.py -n␣
↪MemoryLimitWithResourcesTest -r
cat output/daint/gpu/gnu/MemoryLimitWithResourcesTest/rfm_
↪MemoryLimitWithResourcesTest_job.sh
```

Modifying the parallel launcher command

Another relatively common need is to modify the parallel launcher command. ReFrame gives the ability to do that and we will see some examples in this section.

The most common case is to pass arguments to the launcher command that you cannot normally pass as job options. The `--cpu-bind` of `srun` is such an example. Inside a ReFrame test, you can access the parallel launcher through the *launcher* of the job descriptor. This object handles all the details of how the parallel launch command will be emitted. In the following test we run a CPU affinity test using this utility and we will pin the threads using the `--cpu-bind` option:

```
cat tutorials/advanced/affinity/affinity.py
```

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


@rfm.simple_test
class AffinityTest(rfm.RegressionTest):
    valid_systems = ['daint:gpu', 'daint:mc']
    valid_prog_environs = ['*']
    sourcesdir = 'https://github.com/vkarak/affinity.git'
    build_system = 'Make'
    executable = './affinity'

    @rfm.run_before('compile')
    def set_build_system_options(self):
        self.build_system.options = ['OPENMP=1']

    @rfm.run_before('run')
    def set_cpu_binding(self):
        self.job.launcher.options = ['--cpu-bind=cores']

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(r'CPU affinity', self.stdout)
```

The approach is identical to the approach we took in the `MemoryLimitTest` test *above*, except that we now set the launcher options.

Note: The sanity checking in a real affinity checking test would be much more complex than this.

Another scenario that might often arise when testing parallel debuggers is the need to wrap the launcher command with the debugger command. For example, in order to debug a parallel program with ARM DDT, you would need to invoke the program like this: `ddt [OPTIONS] srun [OPTIONS]`. ReFrame allows you to wrap the launcher command without the test needing to know which is the actual parallel launcher command for the current partition. This can be achieved with the following pipeline hook:

```
import reframe as rfm
from reframe.core.launchers import LauncherWrapper

class DebuggerTest(rfm.RunOnlyRegressionTest):
    ...

    @rfm.run_before('run')
    def set_launcher(self):
        self.job.launcher = LauncherWrapper(self.job.launcher, 'ddt',
                                            ['--offline'])
```

The *LauncherWrapper* is a pseudo-launcher that wraps another one and allows you to prepend anything to it. In this case the resulting parallel launch command, if the current partition uses native Slurm, will be `ddt --offline srun [OPTIONS]`.

Replacing the parallel launcher

Sometimes you might need to replace completely the partition's launcher command, because the software you are testing might use its own parallel launcher. Examples are ipyparallel, the GREASY high-throughput scheduler, as well as some visualization software. The trick here is to replace the parallel launcher with the local one, which practically does not emit any launch command, and by now you should almost be able to do it all by yourself:

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


class CustomLauncherTest(rfm.RunOnlyRegressionTest):
    ...
    executable = 'custom_scheduler'
    executable_opts = [...]

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

The *getlauncher()* function takes the registered name of a launcher and returns the class that implements it. You then instantiate the launcher and assign to the *launcher* attribute of the job descriptor.

An alternative to this approach would be to define your own custom parallel launcher and register it with the framework. You could then use it as the scheduler of a system partition in the configuration, but this approach is less test-specific.

Adding more parallel launch commands

ReFrame uses a parallel launcher by default for anything defined explicitly or implicitly in the *executable* test attribute. But what if we want to generate multiple parallel launch commands? One straightforward solution is to hardcode the parallel launch command inside the *prerun_cmds* or *postrun_cmds*, but this is not so portable. The best way is to ask ReFrame to emit the parallel launch command for you. The following is a simple test for demonstration purposes that runs the `hostname` command several times using a parallel launcher. It resembles a scaling test, except that all happens inside a single ReFrame test, instead of launching multiple instances of a parameterized test.

```
cat tutorials/advanced/multilaunch/multilaunch.py
```

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


@rfm.simple_test
class MultiLaunchTest(rfm.RunOnlyRegressionTest):
    valid_systems = ['daint:gpu', 'daint:mc']
    valid_prog_environs = ['builtin']
    executable = 'hostname'
    num_tasks = 4
    num_tasks_per_node = 1

    @rfm.run_before('run')
    def pre_launch(self):
        cmd = self.job.launcher.run_command(self.job)
        self.prerun_cmds = [
            f'{cmd} -n {n} {self.executable}'
            for n in range(1, self.num_tasks)
        ]

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_eq(
            sn.count(sn.extractall(r'^nid\d+', self.stdout)), 10
        )
```

The additional parallel launch commands are inserted in either the `prerun_cmds` or `postrun_cmds` lists. To retrieve the actual parallel launch command for the current partition that the test is running on, you can use the `run_command()` method of the launcher object. Let's see how the generated job script looks like:

```
./bin/reframe -c tutorials/advanced/multilaunch/multilaunch.py -r
cat output/daint/gpu/builtin/MultiLaunchTest/rfm_MultiLaunchTest_job.sh
```

```
#!/bin/bash
#SBATCH --job-name="rfm_MultiLaunchTest_job"
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=1
#SBATCH --output=rfm_MultiLaunchTest_job.out
#SBATCH --error=rfm_MultiLaunchTest_job.err
#SBATCH --time=0:10:0
#SBATCH -A csstaff
#SBATCH --constraint=gpu
srun -n 1 hostname
srun -n 2 hostname
```

(continues on next page)

(continued from previous page)

```
srun -n 3 hostname
srun hostname
```

The first three `srun` commands are emitted through the `prerun_cmds` whereas the last one comes from the test's `executable` attribute.

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:

```
cat tutorials/advanced/flexnodes/flextest.py
```

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


@rfm.simple_test
class HostnameCheck(rfm.RunOnlyRegressionTest):
    valid_systems = ['daint:gpu', 'daint:mc']
    valid_prog_environs = ['cray']
    executable = 'hostname'
    num_tasks = 0
    num_tasks_per_node = 1

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_eq(
            sn.getattr(self, 'num_tasks'),
            sn.count(sn.findall(r'^nid\d+$', self.stdout))
        )
```

The first thing to notice in this test is that *`num_tasks`* is set to zero. This is a requirement for flexible tests. The sanity check of this test simply counts the host names printed and verifies that they are as many as expected. Notice, however, that the sanity check does not use `num_tasks` directly, but rather access the attribute through the *`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 *`getattr()`* sanity function instead of accessing it directly

---

Tip: If you want to run multiple flexible tests at once, it's better to run them using the serial execution policy, because the first test might take all the available nodes and will cause the rest to fail immediately, since there will be no available nodes for them.

Testing containerized applications

New in version 2.20.

ReFrame can be used also to test applications that run inside a container. First, we need to enable the container platform support in ReFrame's configuration and, specifically, at the partition configuration level:

```
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C gpu', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                    'resources': [
                        {
                            'name': 'memory',
                            'options': ['--mem={size}']
                        }
                    ],
                    'container_platforms': [
                        {
                            'type': 'Sarus',
                            'modules': ['sarus']
                        },
                        {
                            'type': 'Singularity',
                            'modules': ['singularity']
                        }
                    ]
                },
```

For each partition, users can define a list of container platforms supported using the `container_platforms` configuration parameter. In this case, we define the Sarus platform for which we set the `modules` parameter in order to instruct ReFrame to load the `sarus` module, whenever it needs to run with this container platform. Similarly, we add an entry for the Singularity platform.

The following parameterized test, will create two tests, one for each of the supported container platforms:

```
cat tutorials/advanced/containers/container_test.py
```

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


@rfm.simple_test
class ContainerTest(rfm.RunOnlyRegressionTest):
    platform = parameter(['Sarus', 'Singularity'])
    valid_systems = ['daint:gpu']
    valid_prog_environs = ['builtin']

    os_release_pattern = r'18.04.\d+ LTS \(Bionic Beaver\)'
    sanity_patterns = sn.assert_found(os_release_pattern, 'release.txt')

    @rfm.run_before('run')
    def set_container_variables(self):
        self.descr = f'Run commands inside a container using {self.platform}'
```

(continues on next page)

(continued from previous page)

```
        image_prefix = 'docker://' if self.platform == 'Singularity' else ''
        self.container_platform = self.platform
        self.container_platform.image = f'{image_prefix}ubuntu:18.04'
        self.container_platform.command = (
            "bash -c 'cat /etc/os-release | tee /rfm_workdir/release.txt'"
        )
```

A container-based test can be written as *RunOnlyRegressionTest* that sets the *container_platform* attribute. This attribute accepts a string that corresponds to the name of the container platform that will be used to run the container for this test. If such a platform is not configured for the current system, the test will fail.

As soon as the container platform to be used is defined, you need to specify the container image to use by setting the *image*. In the `Singularity` test variant, we add the `docker://` prefix to the image name, in order to instruct `Singularity` to pull the image from DockerHub. The default command that the container runs can be overwritten by setting the *command* attribute of the container platform.

The *image* is the only mandatory attribute for container-based checks. It is important to note that the *executable* and *executable_opts* attributes of the actual test are ignored in case of container-based tests.

ReFrame will run the container according to the given platform as follows:

```
# Sarus
sarus run --mount=type=bind,source="/path/to/test/stagedir",destination="/rfm_workdir
↪" ubuntu:18.04 bash -c 'cat /etc/os-release | tee /rfm_workdir/release.txt'

# Singularity
singularity exec -B"/path/to/test/stagedir:/rfm_workdir" docker://ubuntu:18.04 bash -
↪c 'cat /etc/os-release | tee /rfm_workdir/release.txt'
```

In the `Sarus` case, ReFrame will prepend the following command in order to pull the container image before running the container:

```
sarus pull ubuntu:18.04
```

This is the default behavior of ReFrame, which can be changed if pulling the image is not desired by setting the *pull_image* attribute to `False`. By default ReFrame will mount the stage directory of the test under `/rfm_workdir` inside the container. Once the commands are executed, the container is stopped and ReFrame goes on with the sanity and performance checks. 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')]
```

The container filesystem is ephemeral, therefore, ReFrame mounts the stage directory under `/rfm_workdir` inside the container where the user can copy artifacts as needed. These artifacts will therefore be available inside the stage directory after the container execution finishes. This is very useful if the artifacts are needed for the sanity or performance checks. If the copy is not performed by the default container command, the user can override this command by settings the *command* attribute such as to include the appropriate copy commands. In the current test, the output of the `cat /etc/os-release` is available both in the standard output as well as in the `release.txt` file, since we have used the command:

```
bash -c 'cat /etc/os-release | tee /rfm_workdir/release.txt'
```

and `/rfm_workdir` corresponds to the stage directory on the host system. Therefore, the `release.txt` file can now be used in the subsequent sanity checks:

```
    os_release_pattern = r'18.04.\d+ LTS \(Bionic Beaver\)'
    sanity_patterns = sn.assert_found(os_release_pattern, 'release.txt')
```

For a complete list of the available attributes of a specific container platform, please have a look at the *Container Platforms* section of the *Regression Tests API* guide. On how to configure ReFrame for running containerized tests, please have a look at the *Container Platform Configuration* section of the *Configuration Reference*.

Writing reusable tests

New in version 3.5.0.

So far, all the examples shown above were tight to a particular system or configuration, which makes reusing these tests in other systems not straightforward. However, the introduction of the *parameter()* and *variable()* ReFrame built-ins solves this problem, eliminating the need to specify any of the test variables in the `__init__()` method and simplifying code reuse. Hence, readers who are not familiar with these built-in functions are encouraged to read their basic use examples (see *parameter()* and *variable()*) before delving any deeper into this tutorial.

In essence, parameters and variables can be treated as simple class attributes, which allows us to leverage Python's class inheritance and write more modular tests. For simplicity, we illustrate this concept with the above `ContainerTest` example, where the goal here is to re-write this test as a library that users can simply import from and derive their tests without having to rewrite the bulk of the test. Also, for illustrative purposes, we parameterize this library test on a few different image tags (the above example just used `ubuntu:18.04`) and throw the container commands into a separate bash script just to create some source files. Thus, removing all the system and configuration specific variables, and moving as many assignments as possible into the class body, the system agnostic library test looks as follows:

```
cat tutorials/advanced/library/lib/__init__.py
```

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


class ContainerBase(rfm.RunOnlyRegressionTest, pin_prefix=True):
    '''Test that asserts the ubuntu version of the image.'''

    # Derived tests must override this parameter
    platform = parameter()
    image_prefix = variable(str, value='')

    # Parametrize the test on two different versions of ubuntu.
    dist = parameter(['18.04', '20.04'])
    dist_name = variable(dict, value={
        '18.04': 'Bionic Beaver',
        '20.04': 'Focal Fossa',
    })

    @rfm.run_after('setup')
    def set_description(self):
        self.descr = (
            f'Run commands inside a container using ubuntu {self.dist}'
        )

    @rfm.run_before('run')
    def set_container_platform(self):
```

(continues on next page)

(continued from previous page)

```
        self.container_platform = self.platform
        self.container_platform.image = (
            f'{self.image_prefix}ubuntu:{self.dist}'
        )
        self.container_platform.command = (
            "bash -c /rfm_workdir/get_os_release.sh"
        )

    @property
    def os_release_pattern(self):
        name = self.dist_name[self.dist]
        return rf'{self.dist}.\d+ LTS \({name}\)'

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.all([
            sn.assert_found(self.os_release_pattern, 'release.txt'),
            sn.assert_found(self.os_release_pattern, self.stdout)
        ])
```

Note that the class `ContainerBase` is not decorated since it does not specify the required variables `valid_systems` and `valid_prog_environs`, and it declares the `platform` parameter without any defined values assigned. Hence, the user can simply derive from this test and specialize it to use the desired container platforms. Since the parameters are defined directly in the class body, the user is also free to override or extend any of the other parameters in a derived test. In this example, we have parameterized the base test to run with the `ubuntu:18.04` and `ubuntu:20.04` images, but these values from `dist` (and also the `dist_name` variable) could be modified by the derived class if needed.

On the other hand, the rest of the test depends on the values from the test parameters, and a parameter is only assigned a specific value after the class has been instantiated. Thus, the rest of the test is expressed as hooks, without the need to write anything in the `__init__()` method. In fact, writing the test in this way permits having hooks that depend on undefined variables or parameters. This is the case with the `set_container_platform()` hook, which depends on the undefined parameter `platform`. Hence, the derived test **must** define all the required parameters and variables; otherwise ReFrame will notice that the test is not well defined and will raise an error accordingly.

Before moving onwards to the derived test, note that the `ContainerBase` class takes the additional argument `pin_prefix=True`, which locks the prefix of all derived tests to this base test. This will allow the retrieval of the sources located in the library by any derived test, regardless of what their containing directory is.

```
cat tutorials/advanced/library/lib/src/get_os_release.sh
```

```
#!/bin/bash
cat /etc/os-release | tee /rfm_workdir/release.txt
```

Now from the user's perspective, the only thing to do is to import the above base test and specify the required variables and parameters. For consistency with the above example, we set the `platform` parameter to use Sarus and Singularity, and we configure the test to run on Piz Daint with the built-in programming environment. Hence, the above `ContainerTest` is now reduced to the following:

```
cat tutorials/advanced/library/usr/container_test.py
```

```
import reframe as rfm
import tutorials.advanced.library.lib as lib


```

(continues on next page)

(continued from previous page)

```
@rfm.simple_test
class ContainerTest(lib.ContainerBase):
    platform = parameter(['Sarus', 'Singularity'])
    valid_systems = ['daint:gpu']
    valid_prog_environs = ['builtin']

    @rfm.run_after('setup')
    def set_image_prefix(self):
        if self.platform == 'Singularity':
            self.image_prefix = 'docker://'
```

In a similar fashion, any other user could reuse the above `ContainerBase` class and write the test for their own system with a few lines of code.

Happy test sharing!

2.2.3 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 download and rebuild your test for each runtime configuration. You could have a test where only the sources are fetched, and which all build tests would depend on. And, similarly, all the runtime tests would depend on their corresponding build test. This is the approach we take with the following example, that fetches, builds and runs several OSU benchmarks. We first create a basic run-only test, that fetches the benchmarks:

```
cat tutorials/deps/osu_benchmarks.py
```

```
@rfm.simple_test
class OSUDownloadTest(rfm.RunOnlyRegressionTest):
    descr = 'OSU benchmarks download sources'
    valid_systems = ['daint:login']
    valid_prog_environs = ['builtin']
    executable = 'wget'
    executable_opts = [
        'http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-5.6.
↪2.tar.gz'  # noqa: E501
    ]
    postrun_cmds = [
        'tar xzf osu-micro-benchmarks-5.6.2.tar.gz'
    ]

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_not_found('error', self.stderr)
```

This test doesn't need any specific programming environment, so we simply pick the `builtin` environment in the `login` partition. The build tests would then copy the benchmark code and build it for the different programming environments:

```
@rfm.simple_test
class OSUBuildTest(rfm.CompileOnlyRegressionTest):
    descr = 'OSU benchmarks build test'
    valid_systems = ['daint:gpu']
```

(continues on next page)

(continued from previous page)

```
    valid_prog_environs = ['gnu', 'pgi', 'intel']
    build_system = 'Autotools'

    @rfm.run_after('init')
    def inject_dependencies(self):
        self.depends_on('OSUDownloadTest', udeps.fully)

    @rfm.require_deps
    def set_sourcedir(self, OSUDownloadTest):
        self.sourcesdir = os.path.join(
            OSUDownloadTest(part='login', environ='builtin').stagedir,
            'osu-micro-benchmarks-5.6.2'
        )

    @rfm.run_before('compile')
    def set_build_system_attrs(self):
        self.build_system.max_concurrency = 8

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_not_found('error', self.stderr)
```

The only new thing that comes in with the `OSUBuildTest` test is the following:

```
    @rfm.run_after('init')
    def inject_dependencies(self):
        self.depends_on('OSUDownloadTest', udeps.fully)
```

Here we tell ReFrame that this test depends on a test named `OSUDownloadTest`. This test may or may not be defined in the same test file; all ReFrame needs is the test name. The *`depends_on()`* function will create dependencies between the individual test cases of the `OSUBuildTest` and the `OSUDownloadTest`, such that all the test cases of `OSUBuildTest` will depend on the outcome of the `OSUDownloadTest`. 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 `OSUBuildTest` is to set its `sourcesdir` to point to the source code that was fetched by the `OSUDownloadTest`. This is achieved with the following specially decorated function:

```
    @rfm.require_deps
    def set_sourcedir(self, OSUDownloadTest):
        self.sourcesdir = os.path.join(
            OSUDownloadTest(part='login', environ='builtin').stagedir,
            'osu-micro-benchmarks-5.6.2'
        )
```

The *`@require_deps`* decorator binds each argument of the decorated function to the corresponding target dependency. In order for the binding to work correctly the function arguments must be named after the target dependencies. 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, each dependency argument is actually bound to a function that accepts as argument the name of the target partition and target programming environment. If no arguments are passed, the current programming environment is implied, such that `OSUDownloadTest()` is equivalent to `OSUDownloadTest(self.current_environ.name, self.current_partition.name)`. In this

case, since both the partition and environment of the target dependency do not much those of the current test, we need to specify both.

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 sourcesdir of the test.

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'''

    valid_systems = ['daint:gpu']
    valid_prog_environs = ['gnu', 'pgi', 'intel']
    sourcesdir = None
    num_tasks = 2
    num_tasks_per_node = 1

    @rfm.run_after('init')
    def set_dependencies(self):
        self.depends_on('OSUBuildTest', udeps.by_env)

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(r'^8', self.stdout)


@rfm.simple_test
class OSULatencyTest(OSUBenchmarkTestBase):
    descr = 'OSU latency test'
    reference = {
        '*': {'latency': (0, None, None, 'us')}
    }

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

    @rfm.run_before('performance')
    def set_perf_patterns(self):
        self.perf_patterns = {
            'latency': sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
        }
```

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. As done previously, we define the dependencies with the following:

```
    @rfm.run_after('init')
    def set_dependencies(self):
        self.depends_on('OSUBuildTest', udeps.by_env)
```

Here we tell ReFrame that this test depends on a test named `OSUBuildTest` "by environment." This means that

the test cases of this test will only depend on the test cases of the `OSUBuildTest` that use the same environment; partitions may be different.

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,
            'mpi', 'pt2pt', 'osu_latency'
        )
        self.executable_opts = ['-x', '100', '-i', '1000']
```

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.simple_test
class OSUAllreduceTest(OSUBenchmarkTestBase):
    mpi_tasks = parameter(1 << i for i in range(1, 5))
    descr = 'OSU Allreduce test'
    reference = {
        '*': {'latency': (0, None, None, 'us')}
    }

    @rfm.run_after('init')
    def set_num_tasks(self):
        self.num_tasks = self.mpi_tasks

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

    @rfm.run_before('performance')
    def set_perf_patterns(self):
        self.perf_patterns = {
            'latency': sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
        }
```

The full set of OSU example tests is shown below:

```
# Copyright 2016-2021 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

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

(continues on next page)

(continued from previous page)

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

    valid_systems = ['daint:gpu']
    valid_prog_environs = ['gnu', 'pgi', 'intel']
    sourcesdir = None
    num_tasks = 2
    num_tasks_per_node = 1

    @rfm.run_after('init')
    def set_dependencies(self):
        self.depends_on('OSUBuildTest', udeps.by_env)

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(r'^8', self.stdout)


@rfm.simple_test
class OSULatencyTest(OSUBenchmarkTestBase):
    descr = 'OSU latency test'
    reference = {
        '*': {'latency': (0, None, None, 'us')}
    }

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

    @rfm.run_before('performance')
    def set_perf_patterns(self):
        self.perf_patterns = {
            'latency': sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
        }


@rfm.simple_test
class OSUBandwidthTest(OSUBenchmarkTestBase):
    descr = 'OSU bandwidth test'
    reference = {
        '*': {'bandwidth': (0, None, None, 'MB/s')}
    }

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

(continues on next page)

(continued from previous page)

```
    @rfm.run_before('performance')
    def set_perf_patterns(self):
        self.perf_patterns = {
            'bandwidth': sn.extractsingle(r'^4194304\s+(\S+)',
                                          self.stdout, 1, float)
        }


@rfm.simple_test
class OSUAllreduceTest(OSUBenchmarkTestBase):
    mpi_tasks = parameter(1 << i for i in range(1, 5))
    descr = 'OSU Allreduce test'
    reference = {
        '*': {'latency': (0, None, None, 'us')}
    }

    @rfm.run_after('init')
    def set_num_tasks(self):
        self.num_tasks = self.mpi_tasks

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

    @rfm.run_before('performance')
    def set_perf_patterns(self):
        self.perf_patterns = {
            'latency': sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
        }


@rfm.simple_test
class OSUBuildTest(rfm.CompileOnlyRegressionTest):
    descr = 'OSU benchmarks build test'
    valid_systems = ['daint:gpu']
    valid_prog_environs = ['gnu', 'pgi', 'intel']
    build_system = 'Autotools'

    @rfm.run_after('init')
    def inject_dependencies(self):
        self.depends_on('OSUDownloadTest', udeps.fully)

    @rfm.require_deps
    def set_sourcedir(self, OSUDownloadTest):
        self.sourcesdir = os.path.join(
            OSUDownloadTest(part='login', environ='builtin').stagedir,
            'osu-micro-benchmarks-5.6.2'
        )

    @rfm.run_before('compile')
    def set_build_system_attrs(self):
        self.build_system.max_concurrency = 8
```

(continues on next page)

(continued from previous page)

```
    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_not_found('error', self.stderr)


@rfm.simple_test
class OSUDownloadTest(rfm.RunOnlyRegressionTest):
    descr = 'OSU benchmarks download sources'
    valid_systems = ['daint:login']
    valid_prog_environs = ['builtin']
    executable = 'wget'
    executable_opts = [
        'http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-5.6.
↪2.tar.gz'  # noqa: E501
    ]
    postrun_cmds = [
        'tar xzf osu-micro-benchmarks-5.6.2.tar.gz'
    ]

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        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:

```
./bin/reframe -c tutorials/deps/osu_benchmarks.py -r
```

```
[ReFrame Setup]
  version:           3.6.0-dev.0+4de0fee1
  command:           './bin/reframe -c tutorials/deps/osu_benchmarks.py -r'
  launched by:       user@daint101
  working directory: '/users/user/Devel/reframe'
  settings file:     'tutorials/config/settings.py'
  check search path: '/users/user/Devel/reframe/tutorials/deps/osu_benchmarks.py'
  stage directory:   '/users/user/Devel/reframe/stage'
  output directory:  '/users/user/Devel/reframe/output'

[==========] Running 8 check(s)
[==========] Started on Wed Mar 10 20:53:56 2021

[----------] started processing OSUDownloadTest (OSU benchmarks download sources)
[ RUN      ] OSUDownloadTest on daint:login using builtin
[----------] finished processing OSUDownloadTest (OSU benchmarks download sources)

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

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

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

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

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

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

[----------] waiting for spawned checks to finish
[       OK ] ( 1/22) OSUDownloadTest on daint:login using builtin [compile: 0.007s␣
↪run: 2.033s total: 2.078s]
```

(continues on next page)

(continued from previous page)

```
[       OK ] ( 2/22) OSUBuildTest on daint:gpu using gnu [compile: 20.531s run: 0.
↪039s total: 83.089s]
[       OK ] ( 3/22) OSUBuildTest on daint:gpu using pgi [compile: 27.193s run: 55.
↪871s total: 83.082s]
[       OK ] ( 4/22) OSUAllreduceTest_16 on daint:gpu using gnu [compile: 0.007s run:␣
↪30.713s total: 33.470s]
[       OK ] ( 5/22) OSUBuildTest on daint:gpu using intel [compile: 35.256s run: 54.
↪218s total: 116.712s]
[       OK ] ( 6/22) OSULatencyTest on daint:gpu using pgi [compile: 0.011s run: 23.
↪738s total: 51.190s]
[       OK ] ( 7/22) OSUAllreduceTest_2 on daint:gpu using gnu [compile: 0.008s run:␣
↪31.879s total: 51.187s]
[       OK ] ( 8/22) OSUAllreduceTest_4 on daint:gpu using gnu [compile: 0.006s run:␣
↪37.447s total: 51.194s]
[       OK ] ( 9/22) OSUAllreduceTest_8 on daint:gpu using gnu [compile: 0.007s run:␣
↪42.914s total: 51.202s]
[       OK ] (10/22) OSUAllreduceTest_16 on daint:gpu using pgi [compile: 0.006s run:␣
↪51.172s total: 51.197s]
[       OK ] (11/22) OSULatencyTest on daint:gpu using gnu [compile: 0.007s run: 21.
↪500s total: 51.730s]
[       OK ] (12/22) OSUAllreduceTest_2 on daint:gpu using pgi [compile: 0.007s run:␣
↪35.083s total: 51.700s]
[       OK ] (13/22) OSUAllreduceTest_8 on daint:gpu using pgi [compile: 0.007s run:␣
↪46.187s total: 51.681s]
[       OK ] (14/22) OSUAllreduceTest_4 on daint:gpu using pgi [compile: 0.007s run:␣
↪41.060s total: 52.030s]
[       OK ] (15/22) OSUAllreduceTest_2 on daint:gpu using intel [compile: 0.008s␣
↪run: 27.401s total: 35.900s]
[       OK ] (16/22) OSUBandwidthTest on daint:gpu using gnu [compile: 0.008s run: 82.
↪553s total: 107.334s]
[       OK ] (17/22) OSUBandwidthTest on daint:gpu using pgi [compile: 0.009s run: 87.
↪559s total: 109.613s]
[       OK ] (18/22) OSUAllreduceTest_16 on daint:gpu using intel [compile: 0.006s␣
↪run: 99.899s total: 99.924s]
[       OK ] (19/22) OSUBandwidthTest on daint:gpu using intel [compile: 0.007s run:␣
↪116.771s total: 128.125s]
[       OK ] (20/22) OSULatencyTest on daint:gpu using intel [compile: 0.008s run:␣
↪114.236s total: 128.398s]
[       OK ] (21/22) OSUAllreduceTest_8 on daint:gpu using intel [compile: 0.008s␣
↪run: 125.541s total: 128.387s]
[       OK ] (22/22) OSUAllreduceTest_4 on daint:gpu using intel [compile: 0.007s␣
↪run: 123.079s total: 128.651s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 22/22 test case(s) from 8 check(s) (0 failure(s))
[==========] Finished on Wed Mar 10 20:58:03 2021
Log file(s) saved in: '/tmp/rfm-q0gd9y6v.log'
```

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.

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.

When selecting tests using the test filtering options, such as the `-t`, `-n` etc., ReFrame will automatically select any dependencies of these tests as well. For example, if we select only the `OSULatencyTest` for running, ReFrame will also select the `OSUBuildTest` and the `OSUDownloadTest`:

```
./bin/reframe -c tutorials/deps/osu_benchmarks.py -n OSULatencyTest -l
```

```
$ ./bin/reframe -C -c tutorials/deps/osu_benchmarks.py -n OSULatencyTest -l
[ReFrame Setup]
  version:           3.3-dev2 (rev: 8ded20cd)
  command:           './bin/reframe -C tutorials/config/settings.py -c tutorials/deps/
↪osu_benchmarks.py -n OSULatencyTest -l'
  launched by:       user@daint101
  working directory: '/users/user/Devel/reframe'
  settings file:     'tutorials/config/settings.py'
  check search path: '/users/user/Devel/reframe/tutorials/deps/osu_benchmarks.py'
  stage directory:   '/users/user/Devel/reframe/stage'
  output directory:  '/users/user/Devel/reframe/output'

[List of matched checks]
- OSUDownloadTest (found in '/users/user/Devel/reframe/tutorials/deps/osu_benchmarks.
↪py')
- OSUBuildTest (found in '/users/user/Devel/reframe/tutorials/deps/osu_benchmarks.py')
- OSULatencyTest (found in '/users/user/Devel/reframe/tutorials/deps/osu_benchmarks.py
↪')
Found 3 check(s)
Log file(s) saved in: '/tmp/rfm-4c15g820.log'
```

Finally, when ReFrame cannot resolve a dependency of a test, it will issue a warning and skip completely all the test cases that recursively depend on this one. In the following example, we restrict the run of the `OSULatencyTest` to the `daint:gpu` partition. This is problematic, since its dependencies cannot run on this partition and, particularly, the `OSUDownloadTest`. As a result, its immediate dependency `OSUBuildTest` will be skipped, which will eventually cause all combinations of the `OSULatencyTest` to be skipped.

```
./bin/reframe -c tutorials/deps/osu_benchmarks.py --system=daint:gpu -n␣
↪OSULatencyTest -l
```

```
[ReFrame Setup]
  version:           3.6.0-dev.0+4de0fee1
  command:           './bin/reframe -c tutorials/deps/osu_benchmarks.py --
↪system=daint:gpu -n OSULatencyTest -l'
  launched by:       user@daint101
  working directory: '/users/user/Devel/reframe'
  settings file:     'tutorials/config/settings.py'
  check search path: '/users/user/Devel/reframe/tutorials/deps/osu_benchmarks.py'
  stage directory:   '/users/user/Devel/reframe/stage'
  output directory:  '/users/user/Devel/reframe/output'

./bin/reframe: could not resolve dependency: ('OSUBuildTest', 'daint:gpu', 'gnu') ->
↪'OSUDownloadTest'
./bin/reframe: could not resolve dependency: ('OSUBuildTest', 'daint:gpu', 'intel') ->
↪ 'OSUDownloadTest'
./bin/reframe: could not resolve dependency: ('OSUBuildTest', 'daint:gpu', 'pgi') ->
↪'OSUDownloadTest'
./bin/reframe: skipping all dependent test cases
  - ('OSUBuildTest', 'daint:gpu', 'intel')
```

(continues on next page)

(continued from previous page)

```
  - ('OSUAllreduceTest_2', 'daint:gpu', 'intel')
  - ('OSUBuildTest', 'daint:gpu', 'pgi')
  - ('OSULatencyTest', 'daint:gpu', 'pgi')
  - ('OSUAllreduceTest_8', 'daint:gpu', 'intel')
  - ('OSUAllreduceTest_4', 'daint:gpu', 'pgi')
  - ('OSULatencyTest', 'daint:gpu', 'intel')
  - ('OSUAllreduceTest_4', 'daint:gpu', 'intel')
  - ('OSUAllreduceTest_8', 'daint:gpu', 'pgi')
  - ('OSUAllreduceTest_16', 'daint:gpu', 'pgi')
  - ('OSUAllreduceTest_16', 'daint:gpu', 'intel')
  - ('OSUBandwidthTest', 'daint:gpu', 'pgi')
  - ('OSUBuildTest', 'daint:gpu', 'gnu')
  - ('OSUBandwidthTest', 'daint:gpu', 'intel')
  - ('OSUBandwidthTest', 'daint:gpu', 'gnu')
  - ('OSUAllreduceTest_2', 'daint:gpu', 'pgi')
  - ('OSUAllreduceTest_16', 'daint:gpu', 'gnu')
  - ('OSUAllreduceTest_2', 'daint:gpu', 'gnu')
  - ('OSULatencyTest', 'daint:gpu', 'gnu')
  - ('OSUAllreduceTest_4', 'daint:gpu', 'gnu')
  - ('OSUAllreduceTest_8', 'daint:gpu', 'gnu')

[List of matched checks]

Found 0 check(s)

Log file(s) saved in: '/tmp/rfm-6cxeil6h.log'
```

Listing Dependencies

You can view the dependencies of a test by using the `-L` option:

```
./bin/reframe -c tutorials/deps/osu_benchmarks.py -n OSULatencyTest -L
```

```
< ... omitted ... >

- OSULatencyTest:
    Description:
      OSU latency test

    Environment modules:
      <none>

    Location:
      /users/user/Devel/reframe/tutorials/deps/osu_benchmarks.py

    Maintainers:
      <none>

    Node allocation:
      standard (2 task(s))

    Pipeline hooks:
      - post_setup: set_executable

    Tags:
```

(continues on next page)

(continued from previous page)

```
      <none>

    Valid environments:
      gnu, pgi, intel

    Valid systems:
      daint:gpu

    Dependencies (conceptual):
      OSUBuildTest

    Dependencies (actual):
      - ('OSULatencyTest', 'daint:gpu', 'gnu') -> ('OSUBuildTest', 'daint:login', 'gnu
↪')
      - ('OSULatencyTest', 'daint:gpu', 'intel') -> ('OSUBuildTest', 'daint:login',
↪'intel')
      - ('OSULatencyTest', 'daint:gpu', 'pgi') -> ('OSUBuildTest', 'daint:login', 'pgi
↪')

< ... omitted ... >
```

Dependencies are not only listed conceptually, e.g., "test A depends on test B," but also in a way that shows how they are actually interpreted between the different test cases of the tests. The test dependencies do not change conceptually, but their actual interpretation might change from system to system or from programming environment to programming environment. The following listing shows how the actual test cases dependencies are formed when we select only the `gnu` and `builtin` programming environment for running:

Note: If we do not select the `builtin` environment, we will end up with a dangling dependency as in the example above and ReFrame will skip all the dependent test cases.

```
./bin/reframe -c tutorials/deps/osu_benchmarks.py -n OSULatencyTest -L -p builtin -p␣
↪gnu
```

```
< ... omitted ... >

- OSULatencyTest:
    Description:
      OSU latency test

    Environment modules:
      <none>

    Location:
      /users/user/Devel/reframe/tutorials/deps/osu_benchmarks.py

    Maintainers:
      <none>

    Node allocation:
      standard (2 task(s))

    Pipeline hooks:
      - post_setup: set_executable
```

(continues on next page)

(continued from previous page)

```
    Tags:
      <none>

    Valid environments:
      gnu, pgi, intel

    Valid systems:
      daint:gpu

    Dependencies (conceptual):
      OSUBuildTest

    Dependencies (actual):
      - ('OSULatencyTest', 'daint:gpu', 'gnu') -> ('OSUBuildTest', 'daint:login', 'gnu
↪')

< ... omitted ... >
```

For more information on test dependencies, you can have a look at *How Test Dependencies Work In ReFrame*.

2.2.4 Tutorial 4: Tips and Tricks

New in version 3.4.

This tutorial focuses on some less known aspects of ReFrame's command line interface that can be helpful.

Debugging

ReFrame tests are Python classes inside Python source files, so the usual debugging techniques for Python apply, but the ReFrame frontend will filter some errors and stack traces by default in order to keep the output clean. ReFrame test files are imported, so any error that appears during import time will cause the test loading process to fail and print a stack trace pointing to the offending line. In the following, we have inserted a small typo in the `hello2.py` tutorial example:

```
./bin/reframe -c tutorials/basics/hello -R -l
```

```
./bin/reframe: name error: name 'rm' is not defined
./bin/reframe: Traceback (most recent call last):
  File "/Users/karakasv/Repositories/reframe/reframe/frontend/cli.py", line 668, in␣
↪main
    checks_found = loader.load_all()
  File "/Users/karakasv/Repositories/reframe/reframe/frontend/loader.py", line 204,␣
↪in load_all
    checks.extend(self.load_from_dir(d, self._recurse))
  File "/Users/karakasv/Repositories/reframe/reframe/frontend/loader.py", line 189,␣
↪in load_from_dir
    checks.extend(self.load_from_file(entry.path))
  File "/Users/karakasv/Repositories/reframe/reframe/frontend/loader.py", line 174,␣
↪in load_from_file
    return self.load_from_module(util.import_module_from_file(filename))
  File "/Users/karakasv/Repositories/reframe/reframe/utility/__init__.py", line 96,␣
↪in import_module_from_file
    return importlib.import_module(module_name)
  File "/usr/local/Cellar/python/3.7.7/Frameworks/Python.framework/Versions/3.7/lib/
↪python3.7/importlib/__init__.py", line 127, in import_module
```

(continues on next page)

(continued from previous page)

```
    return _bootstrap._gcd_import(name[level:], package, level)
  File "<frozen importlib._bootstrap>", line 1006, in _gcd_import
  File "<frozen importlib._bootstrap>", line 983, in _find_and_load
  File "<frozen importlib._bootstrap>", line 967, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 677, in _load_unlocked
  File "<frozen importlib._bootstrap_external>", line 728, in exec_module
  File "<frozen importlib._bootstrap>", line 219, in _call_with_frames_removed
  File "/Users/karakasv/Repositories/reframe/tutorials/basics/hello/hello2.py", line␣
↪10, in <module>
    @rm.parameterized_test(['c'], ['cpp'])
NameError: name 'rm' is not defined
```

However, if there is a Python error inside your test's constructor, ReFrame will issue a warning and keep on loading and initializing the rest of the tests.

```
./bin/reframe: skipping test due to errors: HelloMultiLangTest: use `-v' for more␣
↪information
  FILE: /Users/karakasv/Repositories/reframe/tutorials/basics/hello/hello2.py:13
./bin/reframe: skipping test due to errors: HelloMultiLangTest: use `-v' for more␣
↪information
  FILE: /Users/karakasv/Repositories/reframe/tutorials/basics/hello/hello2.py:13
[List of matched checks]
- HelloTest (found in '/Users/karakasv/Repositories/reframe/tutorials/basics/hello/
↪hello1.py')
Found 1 check(s)
```

As suggested by the warning message, passing `-v` will give you the stack trace for each of the failing tests, as well as some more information about what is going on during the loading.

```
./bin/reframe -c tutorials/basics/hello -R -lv
```

```
./bin/reframe: skipping test due to errors: HelloMultiLangTest: use `-v' for more␣
↪information
  FILE: /Users/karakasv/Repositories/reframe/tutorials/basics/hello/hello2.py:13
Traceback (most recent call last):
  File "/Users/karakasv/Repositories/reframe/reframe/core/decorators.py", line 49, in␣
↪_instantiate_all
    ret.append(_instantiate(cls, args))
  File "/Users/karakasv/Repositories/reframe/reframe/core/decorators.py", line 32, in␣
↪_instantiate
    return cls(*args)
  File "/Users/karakasv/Repositories/reframe/tutorials/basics/hello/hello2.py", line␣
↪13, in __init__
    foo
NameError: name 'foo' is not defined

./bin/reframe: skipping test due to errors: HelloMultiLangTest: use `-v' for more␣
↪information
  FILE: /Users/karakasv/Repositories/reframe/tutorials/basics/hello/hello2.py:13
Traceback (most recent call last):
  File "/Users/karakasv/Repositories/reframe/reframe/core/decorators.py", line 49, in␣
↪_instantiate_all
    ret.append(_instantiate(cls, args))
  File "/Users/karakasv/Repositories/reframe/reframe/core/decorators.py", line 32, in␣
↪_instantiate
    return cls(*args)
```

(continues on next page)

(continued from previous page)

```
  File "/Users/karakasv/Repositories/reframe/tutorials/basics/hello/hello2.py", line␣
↪13, in __init__
    foo
NameError: name 'foo' is not defined

Loaded 1 test(s)
Generated 1 test case(s)
Filtering test cases(s) by name: 1 remaining
Filtering test cases(s) by tags: 1 remaining
Filtering test cases(s) by other attributes: 1 remaining
Final number of test cases: 1
[List of matched checks]
- HelloTest (found in '/Users/karakasv/Repositories/reframe/tutorials/basics/hello/
↪hello1.py')
Found 1 check(s)
Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-ckymcl44.
↪log'
```

Tip: The `-v` option can be given multiple times to increase the verbosity level further.

Debugging deferred expressions

Although deferred expression that are used in `sanity_patterns` and `perf_patterns` behave similarly to normal Python expressions, you need to understand their implicit evaluation rules. One of the rules is that `str()` triggers the implicit evaluation, so trying to use the standard `print()` function with a deferred expression, you might get unexpected results if that expression is not yet to be evaluated. For this reason, ReFrame offers a sanity function counterpart of `print()`, which allows you to safely print deferred expressions.

Let's see that in practice, by printing the filename of the standard output for `HelloMultiLangTest` test. The *`stdout`* is a deferred expression and it will get its value later on while the test executes. Trying to use the standard print here `print()` function here would be of little help, since it would simply give us `None`, which is the value of `stdout` when the test is created.

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


@rfm.simple_test
class HelloMultiLangTest(rfm.RegressionTest):
    lang = parameter(['c', 'cpp'])
    valid_systems = ['*']
    valid_prog_environs = ['*']

    @rfm.run_after('compile')
    def set_sourcepath(self):
        self.sourcepath = f'hello.{self.lang}'

    @rfm.run_before('sanity')
    def set_sanity_patterns(self):
        self.sanity_patterns = sn.assert_found(r'Hello, World\!', sn.print(self.
↪stdout))
```

If we run the test, we can see that the correct standard output filename will be printed after sanity:

```
./bin/reframe -C tutorials/config/settings.py -c tutorials/basics/hello/hello2.py -r
```

```
[----------] waiting for spawned checks to finish
rfm_HelloMultiLangTest_cpp_job.out
[       OK ] (1/4) HelloMultiLangTest_cpp on catalina:default using gnu [compile: 0.
↪677s run: 0.700s total: 1.394s]
rfm_HelloMultiLangTest_c_job.out
[       OK ] (2/4) HelloMultiLangTest_c on catalina:default using gnu [compile: 0.
↪451s run: 1.788s total: 2.258s]
rfm_HelloMultiLangTest_c_job.out
[       OK ] (3/4) HelloMultiLangTest_c on catalina:default using clang [compile: 0.
↪329s run: 1.585s total: 1.934s]
rfm_HelloMultiLangTest_cpp_job.out
[       OK ] (4/4) HelloMultiLangTest_cpp on catalina:default using clang [compile: 0.
↪609s run: 0.373s total: 1.004s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 4 test case(s) from 2 check(s) (0 failure(s))
[==========] Finished on Wed Jan 20 17:19:01 2021
```

Debugging test loading

If you are new to ReFrame, you might wonder sometimes why your tests are not loading or why your tests are not running on the partition they were supposed to run. This can be due to ReFrame picking the wrong configuration entry or that your test is not written properly (not decorated, no `valid_systems` etc.). If you try to load a test file and list its tests by increasing twice the verbosity level, you will get enough output to help you debug such issues. Let's try loading the `tutorials/basics/hello/hello2.py` file:

```
./bin/reframe -C tutorials/config/settings.py -c tutorials/basics/hello/hello2.py -lvv
```

```
Loading user configuration
Loading configuration file: 'tutorials/config/settings.py'
Detecting system
Looking for a matching configuration entry for system 'dhcp-133-191.cscs.ch'
Configuration found: picking system 'generic'
Selecting subconfig for 'generic'
Initializing runtime
Selecting subconfig for 'generic:default'
Initializing system partition 'default'
Selecting subconfig for 'generic'
Initializing system 'generic'
Initializing modules system 'nomod'
[ReFrame Environment]
  RFM_CHECK_SEARCH_PATH=<not set>
  RFM_CHECK_SEARCH_RECURSIVE=<not set>
  RFM_CLEAN_STAGEDIR=<not set>
  RFM_COLORIZE=<not set>
  RFM_CONFIG_FILE=/Users/user/Repositories/reframe/tutorials/config/settings.py
  RFM_GRAYLOG_ADDRESS=<not set>
  RFM_IGNORE_CHECK_CONFLICTS=<not set>
  RFM_IGNORE_REQNODENOTAVAIL=<not set>
  RFM_INSTALL_PREFIX=/Users/user/Repositories/reframe
  RFM_KEEP_STAGE_FILES=<not set>
  RFM_MODULE_MAPPINGS=<not set>
```

(continues on next page)

(continued from previous page)

```
  RFM_MODULE_MAP_FILE=<not set>
  RFM_NON_DEFAULT_CRAYPE=<not set>
  RFM_OUTPUT_DIR=<not set>
  RFM_PERFLOG_DIR=<not set>
  RFM_PREFIX=<not set>
  RFM_PURGE_ENVIRONMENT=<not set>
  RFM_REPORT_FILE=<not set>
  RFM_SAVE_LOG_FILES=<not set>
  RFM_STAGE_DIR=<not set>
  RFM_SYSLOG_ADDRESS=<not set>
  RFM_SYSTEM=<not set>
  RFM_TIMESTAMP_DIRS=<not set>
  RFM_UNLOAD_MODULES=<not set>
  RFM_USER_MODULES=<not set>
  RFM_USE_LOGIN_SHELL=<not set>
  RFM_VERBOSE=<not set>
[ReFrame Setup]
  version:           3.4-dev2 (rev: 33a97c81)
  command:           './bin/reframe -C tutorials/config/settings.py -c tutorials/
↪basics/hello/hello2.py -lvv'
  launched by:       user@dhcp-133-191.cscs.ch
  working directory: '/Users/user/Repositories/reframe'
  settings file:     'tutorials/config/settings.py'
  check search path: '/Users/user/Repositories/reframe/tutorials/basics/hello/hello2.
↪py'
  stage directory:   '/Users/user/Repositories/reframe/stage'
  output directory:  '/Users/user/Repositories/reframe/output'

Looking for tests in '/Users/user/Repositories/reframe/tutorials/basics/hello/hello2.
↪py'
Validating '/Users/user/Repositories/reframe/tutorials/basics/hello/hello2.py': OK
  > Loaded 2 test(s)
Loaded 2 test(s)
Generated 2 test case(s)
Filtering test cases(s) by name: 2 remaining
Filtering test cases(s) by tags: 2 remaining
Filtering test cases(s) by other attributes: 2 remaining
Building and validating the full test DAG
Full test DAG:
  ('HelloMultiLangTest_c', 'generic:default', 'builtin') -> []
  ('HelloMultiLangTest_cpp', 'generic:default', 'builtin') -> []
Final number of test cases: 2
[List of matched checks]
- HelloMultiLangTest_c (found in '/Users/user/Repositories/reframe/tutorials/basics/
↪hello/hello2.py')
- HelloMultiLangTest_cpp (found in '/Users/user/Repositories/reframe/tutorials/basics/
↪hello/hello2.py')
Found 2 check(s)
Log file(s) saved in: '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-3956_dlu.
↪log'
```

You can see all the different phases ReFrame's frontend goes through when loading a test. The first "strange" thing to notice in this log is that ReFrame picked the generic system configuration. This happened because it couldn't find a system entry with a matching hostname pattern. However, it did not impact the test loading, because these tests are valid for any system, but it will affect the tests when running (see *Tutorial 1: Getting Started with ReFrame*) since the generic system does not define any C++ compiler.

After loading the configuration, ReFrame will print out its relevant environment variables and will start examining the given files in order to find and load ReFrame tests. Before attempting to load a file, it will validate it and check if it looks like a ReFrame test. If it does, it will load that file by importing it. This is where any ReFrame tests are instantiated and initialized (see `Loaded 2 test(s)`), as well as the actual test cases (combination of tests, system partitions and environments) are generated. Then the test cases are filtered based on the various filtering command line options as well as the programming environments that are defined for the currently selected system. Finally, the test case dependency graph is built and everything is ready for running (or listing).

Try passing a specific system or partition with the *--system* option or modify the test (e.g., removing the decorator that registers it) and see how the logs change.

Execution modes

ReFrame allows you to create pre-defined ways of running it, which you can invoke from the command line. These are called *execution modes* and are essentially named groups of command line options that will be passed to ReFrame whenever you request them. These are defined in the configuration file and can be requested with the *--mode* command-line option. The following configuration defines an execution mode named `maintenance` and sets up ReFrame in a certain way (selects tests to run, sets up stage and output paths etc.)

```
 'modes': [
     {
         'name': 'maintenance',
         'options': [
             '--unload-module=reframe',
             '--exec-policy=async',
             '--strict',
             '--output=/path/to/$USER/regression/maintenance',
             '--perflogdir=/path/to/$USER/regression/maintenance/logs',
             '--stage=$SCRATCH/regression/maintenance/stage',
             '--report-file=/path/to/$USER/regression/maintenance/reports/maint_
↪report_{sessionid}.json',
             '-Jreservation=maintenance',
             '--save-log-files',
             '--tag=maintenance',
             '--timestamp=%F_%H-%M-%S'
         ]
     },
]
```

The execution modes come handy in situations that you have a standardized way of running ReFrame and you don't want to create and maintain shell scripts around it. In this example, you can simply run ReFrame with

```
./bin/reframe --mode=maintenance -r
```

and it will be equivalent to passing explicitly all the above options. You can still pass any additional command line option and it will supersede or be combined (depending on the behaviour of the option) with those defined in the execution mode. In this particular example, we could change just the reservation name by running

```
./bin/reframe --mode=maintenance -J reservation=maint -r
```

There are two options that you can't use inside execution modes and these are the *-C* and *--system*. The reason is that these option select the configuration file and the configuration entry to load.

Manipulating ReFrame's environment

ReFrame runs the selected tests in the same environment as the one that it executes. It does not unload any environment modules nor sets or unsets any environment variable. Nonetheless, it gives you the opportunity to modify the environment that the tests execute. You can either purge completely all environment modules by passing the `--purge-env` option or ask ReFrame to load or unload some environment modules before starting running any tests by using the `-m` and `-u` options respectively. Of course you could manage the environment manually, but it's more convenient if you do that directly through ReFrame's command-line. If you used an environment module to load ReFrame, e.g., `reframe`, you can use the `-u` to have ReFrame unload it before running any tests, so that the tests start in a clean environment:

```
./bin/reframe -u reframe [...]
```

Environment Modules Mappings

ReFrame allows you to replace environment modules used in tests with other modules on the fly. This is quite useful if you want to test a new version of a module or another combination of modules. Assume you have a test that loads a `gromacs` module:

```
class GromacsTest(rfm.RunOnlyRegressionTest):
    ...
    modules = ['gromacs']
```

This test would use the default version of the module in the system, but you might want to test another version, before making that new one the default. You can ask ReFrame to temporarily replace the `gromacs` module with another one as follows:

```
./bin/reframe -n GromacsTest -M 'gromacs:gromacs/2020.5' -r
```

Every time ReFrame tries to load the `gromacs` module, it will replace it with `gromacs/2020.5`. You can specify multiple mappings at once or provide a file with mappings using the `--module-mappings` option. You can also replace a single module with multiple modules.

A very convenient feature of ReFrame in dealing with modules is that you do not have to care about module conflicts at all, regardless of the modules system backend. ReFrame will take care of unloading any conflicting modules, if the underlying modules system cannot do that automatically. In case of module mappings, it will also respect the module order of the replacement modules and will produce the correct series of "load" and "unload" commands needed by the modules system backend used.

Retrying and Rerunning Tests

If you are running ReFrame regularly as part of a continuous testing procedure you might not want it to generate alerts for transient failures. If a ReFrame test fails, you might want to retry a couple of times before marking it as a failure. You can achieve this with the `--max-retries`. ReFrame will then retry the failing test cases a maximum number of times before reporting them as actual failures. The failed test cases will not be retried immediately after they have failed, but rather at the end of the run session. This is done to give more chances of success in case the failures have been transient.

Another interesting feature introduced in ReFrame 3.4 is the ability to restore a previous test session. Whenever it runs, ReFrame stores a detailed JSON report of the last run under `$HOME/.reframe` (see `--report-file`). Using that file, ReFrame can restore a previous run session using the `--restore-session`. This option is useful when you combine it with the various test filtering options. For example, you might want to rerun only the failed tests or just a specific test in a dependency chain. Let's see an artificial example that uses the following test dependency graph.

Tests `T2` and `T8` are set to fail. Let's run the whole test DAG:

Fig. 1: Complex test dependency graph. Nodes in red are set to fail.

```
./bin/reframe -c unittests/resources/checks_unlisted/deps_complex.py -r
```

```
<output omitted>

[----------] waiting for spawned checks to finish
[       OK ] ( 1/10) T0 on generic:default using builtin [compile: 0.014s run: 0.297s␣
↪total: 0.337s]
[       OK ] ( 2/10) T4 on generic:default using builtin [compile: 0.010s run: 0.171s␣
↪total: 0.207s]
[       OK ] ( 3/10) T5 on generic:default using builtin [compile: 0.010s run: 0.192s␣
↪total: 0.225s]
[       OK ] ( 4/10) T1 on generic:default using builtin [compile: 0.008s run: 0.198s␣
↪total: 0.226s]
[     FAIL ] ( 5/10) T8 on generic:default using builtin [compile: n/a run: n/a␣
↪total: 0.003s]
==> test failed during 'setup': test staged in '/Users/user/Repositories/reframe/
↪stage/generic/default/builtin/T8'
[     FAIL ] ( 6/10) T9 [compile: n/a run: n/a total: n/a]
==> test failed during 'startup': test staged in '<not available>'
[       OK ] ( 7/10) T6 on generic:default using builtin [compile: 0.007s run: 0.224s␣
↪total: 0.262s]
[       OK ] ( 8/10) T3 on generic:default using builtin [compile: 0.007s run: 0.211s␣
↪total: 0.235s]
[     FAIL ] ( 9/10) T2 on generic:default using builtin [compile: 0.011s run: 0.318s␣
↪total: 0.389s]
==> test failed during 'sanity': test staged in '/Users/user/Repositories/reframe/
↪stage/generic/default/builtin/T2'
[     FAIL ] (10/10) T7 [compile: n/a run: n/a total: n/a]
==> test failed during 'startup': test staged in '<not available>'
[----------] all spawned checks have finished

[  FAILED  ] Ran 10 test case(s) from 10 check(s) (4 failure(s))
[==========] Finished on Thu Jan 21 13:58:43 2021

<output omitted>
```

You can restore the run session and run only the failed test cases as follows:

```
./bin/reframe --restore-session --failed -r
```

Of course, as expected, the run will fail again, since these tests were designed to fail.

Instead of running the failed test cases of a previous run, you might simply want to rerun a specific test. This has little meaning if you don't use dependencies, because it would be equivalent to running it separately using the `-n` option. However, if a test was part of a dependency chain, using `--restore-session` will not rerun its dependencies, but it will rather restore them. This is useful in cases where the test that we want to rerun depends on time-consuming tests. There is a little tweak, though, for this to work: you need to have run with `--keep-stage-files` in order to keep the stage directory even for tests that have passed. This is due to two reasons: (a) if a test needs resources from its parents, it will look into their stage directories and (b) ReFrame stores the state of a finished test case inside its stage directory and it will need that state information in order to restore a test case.

Let's try to rerun the `T6` test from the previous test dependency chain:

```
./bin/reframe -c unittests/resources/checks_unlisted/deps_complex.py --keep-stage-
↪files -r
```

```
./bin/reframe --restore-session --keep-stage-files -n T6 -r
```

Notice how only the `T6` test was rerun and none of its dependencies, since they were simply restored:

```
[==========] Running 1 check(s)
[==========] Started on Thu Jan 21 14:27:18 2021

[----------] started processing T6 (T6)
[ RUN      ] T6 on generic:default using builtin
[----------] finished processing T6 (T6)

[----------] waiting for spawned checks to finish
[       OK ] (1/1) T6 on generic:default using builtin [compile: 0.012s run: 0.428s␣
↪total: 0.464s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 1 test case(s) from 1 check(s) (0 failure(s))
[==========] Finished on Thu Jan 21 14:27:19 2021
```

If we tried to run `T6` without restoring the session, we would have to rerun also the whole dependency chain, i.e., also `T5`, `T1`, `T4` and `T0`.

```
./bin/reframe -c unittests/resources/checks_unlisted/deps_complex.py -n T6 -r
```

```
[----------] waiting for spawned checks to finish
[       OK ] (1/5) T0 on generic:default using builtin [compile: 0.012s run: 0.424s␣
↪total: 0.464s]
[       OK ] (2/5) T4 on generic:default using builtin [compile: 0.011s run: 0.348s␣
↪total: 0.381s]
[       OK ] (3/5) T5 on generic:default using builtin [compile: 0.007s run: 0.225s␣
↪total: 0.248s]
[       OK ] (4/5) T1 on generic:default using builtin [compile: 0.009s run: 0.235s␣
↪total: 0.267s]
[       OK ] (5/5) T6 on generic:default using builtin [compile: 0.010s run: 0.265s␣
↪total: 0.297s]
[----------] all spawned checks have finished


[  PASSED  ] Ran 5 test case(s) from 5 check(s) (0 failure(s))
[==========] Finished on Thu Jan 21 14:32:09 2021
```

Integrating into a CI pipeline

New in version 3.4.1.

Instead of running your tests, you can ask ReFrame to generate a child pipeline specification for the Gitlab CI. This will spawn a CI job for each ReFrame test respecting test dependencies. You could run your tests in a single job of your Gitlab pipeline, but you would not take advantage of the parallelism across different CI jobs. Having a separate CI job per test makes it also easier to spot the failing tests.

As soon as you have set up a runner for your repository, it is fairly straightforward to use ReFrame to automatically generate the necessary CI steps. The following is an example of `.gitlab-ci.yml` file that does exactly that:

```
stages:
  - generate
  - test

generate-pipeline:
  stage: generate
  script:
    - reframe --ci-generate=${CI_PROJECT_DIR}/pipeline.yml -c ${CI_PROJECT_DIR}/path/
↪to/tests
  artifacts:
    paths:
      - ${CI_PROJECT_DIR}/pipeline.yml

test-jobs:
  stage: test
  trigger:
    include:
      - artifact: pipeline.yml
        job: generate-pipeline
    strategy: depend
```

It defines two stages. The first one, called `generate`, will call ReFrame to generate the pipeline specification for the desired tests. All the usual test selection options can be used to select specific tests. ReFrame will process them as usual, but instead of running the selected tests, it will generate the correct steps for running each test individually as a Gitlab job. We then pass the generated CI pipeline file to second phase as an artifact and we are done!

The following figure shows one part of the automatically generated pipeline for the test graph depicted *above*.

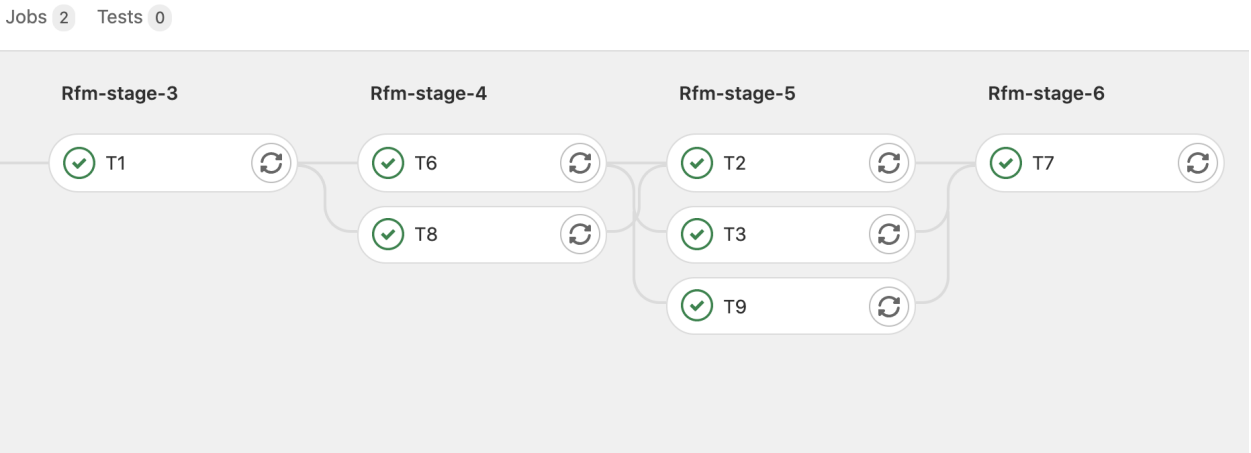

Fig. 2: Snapshot of a Gitlab pipeline generated automatically by ReFrame.

Note: The ReFrame executable must be available in the Gitlab runner that will run the CI jobs.

2.2.5 Online Tutorials

- Tutorial at 6th EasyBuild User Meeting 2021 [YouTube]

2.3 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 site.

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 a 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.

2.3.1 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.

2.3.2 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 by looking into the configuration file of the tutorials, which permits ReFrame to run both on the Piz Daint supercomputer and a local computer. For the complete listing and description of all configuration options, you should refer to the *Configuration Reference*.

```
site_configuration = {
    'systems': [
        {
            'name': 'catalina',
            'descr': 'My Mac',
            'hostnames': ['tresa'],
            'modules_system': 'nomod',
            'partitions': [
                {
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['gnu', 'clang'],
                }
            ]
        },
        {
            'name': 'daint',
            'descr': 'Piz Daint Supercomputer',
            'hostnames': ['daint'],
            'modules_system': 'tmod32',
            'partitions': [
                {
                    'name': 'login',
                    'descr': 'Login nodes',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['builtin', 'gnu', 'intel', 'pgi', 'cray'],
                },
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C gpu', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                    'resources': [
                        {
                            'name': 'memory',
                            'options': ['--mem={size}']
                        }
                    ],
                    'container_platforms': [
                        {
                            'type': 'Sarus',
                            'modules': ['sarus']
                        },
                        {
                            'type': 'Singularity',
                            'modules': ['singularity']
                        }
                    ]
                },
                {
                    'name': 'mc',
                    'descr': 'Multicore nodes',
```

(continues on next page)

(continued from previous page)

```
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C mc', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                    'resources': [
                        {
                            'name': 'memory',
                            'options': ['--mem={size}']
                        }
                    ]
                }
            ]
        },
        {
            'name': 'generic',
            'descr': 'Generic example system',
            'hostnames': ['.*'],
            'partitions': [
                {
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['builtin']
                }
            ]
        },
    ],
    'environments': [
        {
            'name': 'gnu',
            'cc': 'gcc-9',
            'cxx': 'g++-9',
            'ftn': 'gfortran-9'
        },
        {
            'name': 'gnu',
            'modules': ['PrgEnv-gnu'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'cray',
            'modules': ['PrgEnv-cray'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'intel',
            'modules': ['PrgEnv-intel'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
```

(continues on next page)

(continued from previous page)

```
            'target_systems': ['daint']
        },
        {
            'name': 'pgi',
            'modules': ['PrgEnv-pgi'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'clang',
            'cc': 'clang',
            'cxx': 'clang++',
            'ftn': ''
        },
        {
            'name': 'builtin',
            'cc': 'cc',
            'cxx': '',
            'ftn': ''
        },
        {
            'name': 'builtin',
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        }
    ],
    'logging': [
        {
            'level': 'debug',
            'handlers': [
                {
                    'type': 'stream',
                    'name': 'stdout',
                    'level': 'info',
                    'format': '%(message)s'
                },
                {
                    'type': 'file',
                    'level': 'debug',
                    'format': '[%(asctime)s] %(levelname)s: %(check_info)s:
↪%(message)s',   # noqa: E501
                    '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|'
```

(continues on next page)

(continued from previous page)

```
                        'ref=%(check_perf_ref)s '
                        '(l=%(check_perf_lower_thres)s, '
                        'u=%(check_perf_upper_thres)s)|'
                        '%(check_perf_unit)s'
                    ),
                    'append': True
                }
            ]
        }
    ],
}
```

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 three systems, a Mac laptop, Piz Daint and a generic fallback system:

```
    'systems': [
        {
            'name': 'catalina',
            'descr': 'My Mac',
            'hostnames': ['tresa'],
            'modules_system': 'nomod',
            'partitions': [
                {
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['gnu', 'clang'],
                }
            ]
        },
        {
            'name': 'daint',
            'descr': 'Piz Daint Supercomputer',
            'hostnames': ['daint'],
            'modules_system': 'tmod32',
            'partitions': [
                {
                    'name': 'login',
                    'descr': 'Login nodes',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['builtin', 'gnu', 'intel', 'pgi', 'cray'],
                },
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C gpu', '-A csstaff'],
```

(continues on next page)

(continued from previous page)

```
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                    'resources': [
                        {
                            'name': 'memory',
                            'options': ['--mem={size}']
                        }
                    ],
                    'container_platforms': [
                        {
                            'type': 'Sarus',
                            'modules': ['sarus']
                        },
                        {
                            'type': 'Singularity',
                            'modules': ['singularity']
                        }
                    ]
                },
                {
                    'name': 'mc',
                    'descr': 'Multicore nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C mc', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                    'resources': [
                        {
                            'name': 'memory',
                            'options': ['--mem={size}']
                        }
                    ]
                }
            ]
        },
        {
            'name': 'generic',
            'descr': 'Generic example system',
            'hostnames': ['.*'],
            'partitions': [
                {
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['builtin']
```

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`: This refers to the modules management backend which should be used for loading envi-

ronment modules on this system. Multiple backends are supported, as well as the special `nomod` backend which implements the different modules system operations as no-ops. For the complete list of the supported modules systems, see here.

- `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',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C gpu', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'pgi', 'cray'],
                    'max_jobs': 100,
                    'resources': [
                        {
                            'name': 'memory',
                            'options': ['--mem={size}']
                        }
                    ],
                    'container_platforms': [
                        {
                            'type': 'Sarus',
                            'modules': ['sarus']
                        },
                        {
                            'type': 'Singularity',
```

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.
- `resources`: This is a set of optional additional scheduler resources that the tests can access transparently. For more information, please have a look here.

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. For each environment referenced inside a partition, a definition of it must be present in this section. In our example, we define environments for all the basic compilers as well as a default built-in one, which is used with the generic system configuration. In certain contexts, it is useful to see a ReFrame environment as a wrapper of a programming toolchain (MPI + compiler combination):

```
                }
            ]
        },
    ],
    'environments': [
        {
            'name': 'gnu',
            'cc': 'gcc-9',
            'cxx': 'g++-9',
            'ftn': 'gfortran-9'
        },
        {
            'name': 'gnu',
            'modules': ['PrgEnv-gnu'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'cray',
            'modules': ['PrgEnv-cray'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'intel',
            'modules': ['PrgEnv-intel'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'pgi',
            'modules': ['PrgEnv-pgi'],
```

(continues on next page)

(continued from previous page)

```
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'clang',
            'cc': 'clang',
            'cxx': 'clang++',
            'ftn': ''
        },
        {
            'name': 'builtin',
            'cc': 'cc',
            'cxx': '',
            'ftn': ''
        },
        {
            'name': 'builtin',
            'cc': 'cc',
            'cxx': 'CC',
```

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. A programming environment in ReFrame is essentially a collection of environment modules, environment variables and compiler definitions.

An important feature in ReFrame's configuration, is that you can define section objects differently for different systems or system partitions by using the `target_systems` property. Notice, for example, how the `gnu` environment is defined differently for the system `daint` compared to the generic definition. The `target_systems` property is a list of systems or system/partition combinations where this definition of the environment is in effect. This means that `gnu` will be defined this way only for regression tests running on `daint`. For all the other systems, it will be defined using the first definition.

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:

```
            'ftn': 'ftn',
            'target_systems': ['daint']
        }
    ],
    'logging': [
        {
            'level': 'debug',
            'handlers': [
                {
                    'type': 'stream',
                    'name': 'stdout',
                    'level': 'info',
                    'format': '%(message)s'
                },
                {
```

(continues on next page)

(continued from previous page)

```
                    'type': 'file',
                    'level': 'debug',
                    'format': '[%(asctime)s] %(levelname)s: %(check_info)s:
↪%(message)s',    # noqa: E501
                    '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)|'
                        '%(check_perf_unit)s'
                    ),
                    '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. For a complete list of available log handler types, see here.
- `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, a 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. Performance log messages are generated *only* for performance tests, i.e., tests defining the `perf_patterns` attribute. The performance log handlers are stored in the `handlers_perflog` property. The `filelog` handler used in this example will create a file per test and per system/partition combination (`./<system>/<partition>/<testname>.log`) and will append to it the obtained performance data every time a performance test is run. Notice how the message to be logged is structured in the `format` property, 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.

For a complete reference of logging configuration parameters, please refer to the *Configuration Reference*.

General configuration options

General configuration options of the framework go under the `general` section of the configuration file. This section is optional and, in fact, we do not define it for our tutorial configuration file. However, there are several 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.

2.3.3 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': 'cray',
        'modules': ['cray']
    },
    {
```

(continues on next page)

(continued from previous page)

```
        'name': 'gnu',
        'modules': ['gnu']
    },
    {
        'name': 'gnu',
        'modules': ['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 `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`.

2.3.4 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 tutorials/config/settings.py --system=daint --show-
↪config=systems/0/partitions
```

```
[
  {
    "name": "login",
    "descr": "Login nodes",
    "scheduler": "local",
    "launcher": "local",
    "environs": [
      "gnu",
      "intel",
      "pgi",
      "cray"
    ],
    "max_jobs": 10
  },
  {
    "name": "gpu",
    "descr": "Hybrid nodes",
    "scheduler": "slurm",
    "launcher": "srun",
    "access": [
      "-C gpu",
```

(continues on next page)

(continued from previous page)

```
      "-A csstaff"
    ],
    "environs": [
      "gnu",
      "intel",
      "pgi",
      "cray"
    ],
    "max_jobs": 100
  },
  {
    "name": "mc",
    "descr": "Multicore nodes",
    "scheduler": "slurm",
    "launcher": "srun",
    "access": [
      "-C mc",
      "-A csstaff"
    ],
    "environs": [
      "gnu",
      "intel",
      "pgi",
      "cray"
    ],
    "max_jobs": 100
  }
]
```

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

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

```
[
  {
    "name": "login",
    "descr": "Login nodes",
    "scheduler": "local",
    "launcher": "local",
    "environs": [
      "gnu",
      "intel",
      "pgi",
      "cray"
    ],
    "max_jobs": 10
  }
]
```

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 tutorials/config/settings.py --system=daint --show-
↪config=environments/@gnu
```

```
{
  "name": "gnu",
  "modules": [
    "PrgEnv-gnu"
  ],
  "cc": "cc",
  "cxx": "CC",
  "ftn": "ftn",
  "target_systems": [
    "daint"
  ]
}
```

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 tutorials/config/settings.py --system=daint --show-
↪config=environments/@gnu/cxx
```

```
"CC"
```

If you explicitly query a configuration value which is not defined in the configuration file, ReFrame will print its default value.

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. 3: 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. 4: 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.

Fig. 5: The serial execution policy.

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 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. 6: 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 on `T0`. This can be expressed inside `T1` using the `depends_on()` method:

```
@rfm.simple_test
class T0(rfm.RegressionTest):
    ...
    valid_systems = ['P0', 'P1']
    valid_prog_environs = ['E0', 'E1']


@rfm.simple_test
class T1(rfm.RegressionTest):
    ...
    valid_systems = ['P0', 'P1']
    valid_prog_environs = ['E0', 'E1']

    def __init__(self):
        self.depends_on('T0')
```

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

Fig. 7: Simple test dependency presented conceptually.

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. If not specified differently, test cases on different partitions or programming environments are independent. This is the default behavior of the `depends_on()` function. The following image shows the actual test case dependencies of the two tests above:

Fig. 8: Test case dependencies partitioned by case (default).

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:

```
import reframe.utility.udeps as udeps


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

This will create a fully connected graph between the test cases of the two tests as it is shown in the following figure:

Fig. 9: Fully dependent test cases.

There are more options that the test case subgraph can be split than the two extremes we presented so far. The following figures show the different splittings.

Split by partition

The test cases are split in fully connected components per partition. Test cases from different partitions are independent.

Fig. 10: Test case dependencies partitioned by partition.

Split by environment

The test cases are split in fully connected components per environment. Test cases from different environments are independent.

Fig. 11: Test case dependencies partitioned by environment.

Split by exclusive partition

The test cases are split in fully connected components that do not contain the same partition. Test cases from the same partition are independent.

Split by exclusive environment

The test cases are split in fully connected components that do not contain the same environment. Test cases from the same environment are independent.

Fig. 12: Test case dependencies partitioned by exclusive partition.

Fig. 13: Test case dependencies partitioned by exclusive environment.

Split by exclusive case

The test cases are split in fully connected components that do not contain the same environment and the same partition. Test cases from the same environment and the same partition are independent.

Fig. 14: Test case dependencies partitioned by exclusive case.

Custom splits

Users may define custom dependency patterns by supplying their own `how` function. The `how` argument accepts a `callable` which takes as arguments the source and destination of a possible edge in the test case subgraph. If the callable returns `True`, then ReFrame will place an edge (i.e., a dependency) otherwise not. The following code will create dependencies only if the source partition is `P0` and the destination environment is `E1`:

```
def myway(src, dst):
    psrc, esrc = src
    pdst, edst = dst
    return psrc == 'P0' and edst == 'E1'


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

This corresponds to the following test case dependency subgraph:

Notice how all the rest test cases are completely independent.

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, P0, E0)` to `(T1, P0, E1)` introduces a dependency from `T0` to `T1` forming a cycle at the test level. If you end up requiring such type of dependency in your tests, you might have to reconsider how you organize your tests.

Note: Technically, the framework could easily support such types of dependencies, but ReFrame's output would have to change substantially.

Fig. 15: Custom test case dependencies.

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:

```
    @rfm.require_deps
    def set_executable(self, OSUBuildTest):
        self.executable = os.path.join(
            OSUBuildTest().stagedir,
            'mpi', 'pt2pt', 'osu_latency'
        )
```

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(
        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. 16: 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>
>>> 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(1): print(i)
...
2
2
>>> for i in dsize(1): 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 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'.
```

Warning: Changed in version 3.4: The old configuration syntax in no longer supported and it will not be automatically converted by the *-C* option.

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`.

Configuration 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 and ReFrame 3.4 disables that by default. 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 is no longer valid and it will raise a deprecation warning for ReFrame versions >= 3.0 and a reframe syntax error for versions >=3.4. 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']
```

(continues on next page)

(continued from previous page)

```
    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`.

Warning: Changed in version 3.4: Overriding a pipeline stage method is no longer allowed and a reframe syntax error is raised.

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`, it will again result in a reframe syntax error, which is 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.

Warning: Changed in version 3.4: The `prebuild_cmd`, `postbuild_cmd`, `pre_run` and `post_run` attributes have been removed.

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. A common characteristic of all test filtering options is that if a test is selected, then all its dependencies will be selected, too, regardless if they match the filtering criteria or not. This happens recursively so that if test `T1` depends on `T2` and `T2` depends on `T3`, then selecting `T1` would also select `T2` and `T3`.

-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.

--failed
Select only the failed test cases for a previous run. This option can only be used in combination with the *--restore-session*. To rerun the failed cases from the last run, you can use reframe --restore-session --failed -r.

New in version 3.4.

--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.

--ci-generate=FILE
Do not run the tests, but generate a Gitlab child pipeline specification in FILE. You can set up your Gitlab CI to use the generated file to run every test as a separate Gitlab job respecting test dependencies. For more information, have a look in *Integrating into a CI pipeline*.

New in version 3.4.1.

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

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

--list-tags
List the unique tags of the selected tests. The tags are printed in alphabetical order.

New in version 3.6.0.

-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.

--dont-restage
Do not restage a test if its stage directory exists. Normally, if the stage directory of a test exists, ReFrame will remove it and recreate it. This option disables this behavior.

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

New in version 3.1.

--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.

--report-file=FILE
The file where ReFrame will store its report. The `FILE` argument may contain the special placeholder `{sessionid}`, in which case ReFrame will generate a new report each time it is run by appending a counter to the report file.

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

New in version 3.1.

--report-junit=FILE
Instruct ReFrame to generate a JUnit XML report in `FILE`. The generated report adheres to the XSD schema here and it takes into account only the first run, ignoring retries of failed tests.

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

New in version 3.6.0.

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.

--maxfail=NUM
The maximum number of failing test cases before the execution is aborted. After `NUM` failed test cases the rest of the test cases will be aborted. The counter of the failed test cases is reset to 0 in every retry.

--disable-hook=HOOK
Disable the pipeline hook named `HOOK` from all the tests that will run. This feature is useful when you have implemented test workarounds as pipeline hooks, in which case you can quickly disable them from the command line. This option may be specified multiple times in order to disable multiple hooks at the same time.

New in version 3.2.

--restore-session [REPORT]
Restore a testing session that has run previously. `REPORT` is a run report file generated by ReFrame. If `REPORT` is not given, ReFrame will pick the last report file found in the default location of report files (see the *--report-file* option). If passed alone, this option will simply rerun all the test cases that have run previously based on the report file data. It is more useful to combine this option with any of the *test filtering* options, in which case only the selected test cases will be executed. The difference in test selection process when using this option is that the dependencies of the selected tests will not be selected for execution, as they would normally, but they will be restored. For example, if test `T1` depends on `T2` and `T2` depends on `T3`, then running `reframe -n T1 -r` would cause both `T2` and `T3` to run. However, by doing `reframe -n T1 --restore-session -r`, only `T1` would run and its immediate dependence `T2` will be restored. This is useful when you have deep test dependencies or some of the tests in the dependency chain are very time consuming.

Note: In order for a test case to be restored, its stage directory must be present. This is not a problem when rerunning a failed case, since the stage directories of its dependencies are automatically kept, but if you want to rerun a successful test case, you should make sure to have run with the *--keep-stage-files* option.

New in version 3.4.

Options controlling job submission

-J, --job-option=OPTION
Pass `OPTION` directly to the job scheduler backend. The syntax of `OPTION` is `-J key=value`. If `OPTION` starts with `-` it will be passed verbatim to the backend job scheduler. If `OPTION` starts with `#` it will be emitted verbatim in the job script. Otherwise, ReFrame will pass `--key value` or `-k value` (if `key` is a single character) to the backend scheduler. Any job options specified with this command-line option will be emitted after any job options specified in the `access` system partition configuration parameter.

Especially for the Slurm backends, constraint options, such as `-J constraint=value`, `-J C=value`, `-J --constraint=value` or `-J -C=value`, are going to be combined with any constraint options specified in the `access` system partition configuration parameter. For example, if `-C x` is specified in the `access` and `-J C=y` is passed to the command-line, ReFrame will pass `-C x&y` as a constraint to the scheduler.

Notice, however, that if constraint options are specified through multiple `-J` options, only the last one will be considered. If you wish to completely overwrite any constraint options passed in `access`, you should consider passing explicitly the Slurm directive with `-J '#SBATCH --constraint=new'`.

Changed in version 3.0: This option has become more flexible.

Changed in version 3.1: Use `&` to combine constraints.

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.
- `STATE`: Flexible tests will be assigned as many tasks as needed in order to span over the nodes that are currently in state `STATE`. 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 nodes in the given state.

  If this option is not specified, the default allocation policy for flexible tests is 'idle'.
- 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.

Changed in version 3.1: It is now possible to pass an arbitrary node state as a flexible node allocation parameter.

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.

`--module-path`=PATH
Manipulate the `MODULEPATH` environment variable before acting on any tests. If `PATH` starts with the `-` character, it will be removed from the `MODULEPATH`, whereas if it starts with the `+` character, it will be added

to it. In all other cases, `PATH` will completely override MODULEPATH. This option may be specified multiple times, in which case all the paths specified will be added or removed in order.

New in version 3.3.

--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.

Changed in version 3.3: If the mapping replaces a module collection, all new names must refer to module collections, too.

See also:

Module collections with Environment Modules and Lmod.

--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_CLEAN_STAGEDIR
Clean stage directory of tests before populating it.

New in version 3.1.

Associated command line option	*--dont-restage*
Associated configuration parameter	`clean_stagedir` 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_ADDRESS
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

New in version 3.1.

RFM_GRAYLOG_SERVER
Deprecated since version 3.1: Please *RFM_GRAYLOG_ADDRESS* instead.

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_TRAP_JOB_ERRORS
Ignore job exit code

Associated configuration parameter	`trap_job_errors` 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_REPORT_FILE
The file where ReFrame will store its report.

New in version 3.1.

Associated command line option	*--report-file*
Associated configuration parameter	`report_file` general configuration parameter

RFM_REPORT_JUNIT
The file where ReFrame will generate a JUnit XML report.

New in version 3.6.0.

Associated command line option	*--report-junit*
Associated configuration parameter	`report_junit` general configuration parameter

RFM_RESOLVE_MODULE_CONFLICTS
Resolve module conflicts automatically.

New in version 3.6.0.

Associated command line option	n/a
Associated configuration parameter	`resolve_module_conflicts` 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_SYSLOG_ADDRESS
The address of the Syslog server to send performance logs. The address is specified in `host:port` format. If no port is specified, the address refers to a UNIX socket.

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

New in version 3.1.

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.
- `spack`: Spack's built-in mechanism for managing modules.
- `nomod`: This is to denote that no modules system is used by this system.

New in version 3.4: The `spack` backend is added.

`.systems[].`**`modules`**

Required No

Default `[]`

A list of *environment module objects* 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 module objects* to be loaded before running a regression test on this partition.

`.systems[].partitions[].`**`time_limit`**

Required No

Default `null`

The time limit for the jobs submitted on this partition. When the value is `null`, no time limit is applied.

`.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 `8`

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[].prepare_cmds`

Required No

Default `[]`

List of shell commands to be emitted before any environment loading commands are emitted.

New in version 3.5.0.

`.systems[].partitions[].resources`

Required No

Default `[]`

A list of job scheduler resource specification objects.

`.systems[].partitions[].processor`

Required No

Default `{}`

Processor information for this partition stored in a *processor info object*.

New in version 3.5.0.

`.systems[].partitions[].devices`

Required No

Default `[]`

A list with *device info objects* for this partition.

New in version 3.5.0.

`.systems[].partitions[].extras`

Required No

Default `{}`

User defined attributes of the system partition that will be accessible from the ReFrame tests. By default it is an empty dictionary.

New in version 3.5.0.

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 `[]`

A list of *environment module objects* 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 module objects* 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',
        '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 `"undefined"`

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.
- `debug2`: Further debug messages.
- `undefined`: This is the lowest level; do not filter any message.

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.

New in version 3.3: The `debug2` and `undefined` levels are added.

Changed in version 3.3: The default level is now `undefined`.

`.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_partition)s`: The system partition where this test is currently executing.
- `%(check_system)s`: The system where this test is currently executing.
- `%(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.
- `%(check_ATTR)s`: This will log the value of the attribute `ATTR` of the currently executing regression test. Dictionaries will be logged in JSON format and all other iterables, except strings, will be logged as comma-separated lists. If `ATTR` is not an attribute of the test, `%(check_ATTR)s` will be logged as `null`. This allows users to log arbitrary attributes of their tests. For the complete list of test attributes, please refer to *Regression Tests API*.
- `%(check_job_ATTR)s`: This will log the value of the attribute `ATTR` of the *`job`* associated to the currently executing regression test.
- `%(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.

New in version 3.3: Allow arbitrary test attributes to be logged.

New in version 3.4.2: Allow arbitrary job attributes to be logged.

`.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 No

The name of the file where this handler will write log records. If not specified, ReFrame will create a log file prefixed with `rfm-` in the system's temporary directory.

Changed in version 3.3: The `name` parameter is no more required and the default log file resides in the system's temporary directory.

`.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.

`.schedulers[].resubmit_on_errors`

Required No

Default `[]`

This option is relevant to the Slurm backends only.

If any of the listed errors occur, ReFrame will try to resubmit the job after some seconds. As an example, you could have ReFrame trying to resubmit a job in case that the maximum submission limit per user is reached by setting this field to `["QOSMaxSubmitJobPerUserLimit"]`. You can ignore multiple errors at the same time if you add more error strings in the list.

New in version 3.4.1.

Warning: Job submission is a synchronous operation in ReFrame. If this option is set, ReFrame's execution will block until the error conditions specified in this list are resolved. No other test would be able to proceed.

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[].clean_stagedir`

Required No

Default `true`

Clean stage directory of tests before populating it.

New in version 3.1.

`.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[].trap_job_errors`

Required No

Default `false`

Trap command errors in the generated job scripts and let them exit immediately.

`.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[].report_file`

Required No

Default `"${HOME}/.reframe/reports/run-report.json"`

The file where ReFrame will store its report.

New in version 3.1.

Changed in version 3.2: Default value has changed to avoid generating a report file per session.

`.general[].report_junit`

Required No

Default `null`

The file where ReFrame will store its report in JUnit format. The report adheres to the XSD schema here.

New in version 3.6.0.

`.general[].resolve_module_conflicts`

Required No

Default `true`

ReFrame by default resolves any module conflicts and emits the right sequence of `module unload` and `module load` commands, in order to load the requested modules. This option disables this behavior if set to `false`.

You should avoid using this option for modules system that cannot handle module conflicts automatically, such as early Tmod verions.

Disabling the automatic module conflict resolution, however, can be useful when modules in a remote system partition are not present on the host where ReFrame runs. In order to resolve any module conflicts and generate the right load sequence of modules, ReFrame loads temporarily the requested modules and tracks any conflicts along the way. By disabling this option, ReFrame will simply emit the requested `module load` commands without attempting to load any module.

New in version 3.6.0.

`.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 module objects* 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 module objects* 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.

Module Objects

New in version 3.3.

A *module object* in ReFrame's configuration represents an environment module. It can either be a simple string or a JSON object with the following attributes:

`.name`

Required Yes

The name of the module.

`.collection`

Required No

Default `false`

A boolean value indicating whether this module refers to a module collection. Module collections are treated differently from simple modules when loading.

`path`

Required No

Default `null`

If the module is not present in the default `MODULEPATH`, the module's location can be specified here. ReFrame will make sure to set and restore the `MODULEPATH` accordingly for loading the module.

New in version 3.5.0.

See also:

Module collections with Environment Modules and Lmod.

Processor Info

New in version 3.5.0.

A *processor info object* in ReFrame's configuration is used to hold information about the processor of a system partition and is made available to the tests through the *`processor`* attribute of the *`current_partition`*.

`.arch`

Required No

Default `None`

The microarchitecture of the processor.

`.num_cpus`

Required No

Default `None`

Number of logical CPUs.

`.num_cpus_per_core`

Required No

Default `None`

Number of logical CPUs per core.

`.num_cpus_per_socket`

Required No

Default `None`

Number of logical CPUs per socket.

`.num_sockets`

Required No

Default `None`

Number of sockets.

`.topology`

Required No

Default `None`

Processor topology. An example follows:

```
'topology': {
   'numa_nodes': ['0x000000ff'],
   'sockets': ['0x000000ff'],
   'cores': ['0x00000003', '0x0000000c',
             '0x00000030', '0x000000c0'],
   'caches': [
      {
            'type': 'L3',
            'size': 6291456,
            'linesize': 64,
            'associativity': 0,
            'num_cpus': 8,
            'cpusets': ['0x000000ff']
      },
      {
            'type': 'L2',
            'size': 262144,
            'linesize': 64,
            'associativity': 4,
            'num_cpus': 2,
            'cpusets': ['0x00000003', '0x0000000c',
                        '0x00000030', '0x000000c0']
      },
      {
            'type': 'L1',
            'size': 32768,
            'linesize': 64,
            'associativity': 0,
            'num_cpus': 2,
            'cpusets': ['0x00000003', '0x0000000c',
                        '0x00000030', '0x000000c0']
      }
   ]
}
```

Device Info

New in version 3.5.0.

A *device info object* in ReFrame's configuration is used to hold information about a specific type of devices in a system partition and is made available to the tests through the *devices* attribute of the *current_partition*.

.type

Required No

Default `None`

The type of the device, for example `"gpu"`.

.arch

Required No

Default `None`

The microarchitecture of the device.

.num_devices

Required No

Default `None`

Number of devices of this type inside the system partition.

2.7.3 ReFrame Programming APIs

Regression Tests API

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**(**args*, ***kwargs*)

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.

run_wait()

Wait for this test to finish.

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 run job is created 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` or `None` if a run job has not yet been created.

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` or `None` if a run job has not yet been created.

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.

Deprecated since version 3.3: Please use a callable as the `how` argument.

reframe.core.pipeline.**DEPEND_EXACT = 1**
Constant to be passed as the `how` argument of the *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.

Deprecated since version 3.3: Please use a callable as the `how` argument.

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.

Deprecated since version 3.3: Please use a callable as the `how` argument.

class reframe.core.pipeline.**RegressionMixin**(**args*, ***kwargs*)
Bases: `object`

Base mixin class for regression tests.

Multiple inheritance from more than one *RegressionTest* class is not allowed in ReFrame. Hence, mixin classes provide the flexibility to bundle reusable test add-ons, leveraging the metaclass magic implemented in `RegressionTestMeta`. Using this metaclass allows mixin classes to use powerful ReFrame features, such as hooks, parameters or variables.

New in version 3.4.2.

class reframe.core.pipeline.**RegressionTest**(**args*, ***kwargs*)
Bases: *reframe.core.pipeline.RegressionMixin*, `reframe.utility.jsonext.JSONSerializable`

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.

Warning: Changed in version 3.4.2: Multiple inheritance with a shared common ancestor is not allowed.

Note: Changed in version 2.19: Base constructor takes no arguments.

build_locally = True
New in version 3.3.

Always build the source code for this test locally. If set to `False`, ReFrame will spawn a build job on the partition where the test will run. Setting this to `False` is useful when cross-compilation is not supported on the system where ReFrame is run. Normally, ReFrame will mark the test as a failure if the spawned job exits with a non-zero exit code. However, certain scheduler backends, such as the `squeue` do not set it. In such cases, it is the user's responsibility to check whether the build phase failed by adding an appropriate sanity check.

Type boolean : :default: `True`

build_system = None
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`.

build_time_limit = None
New in version 3.5.1.

The time limit for the build job of the regression test.

It is specified similarly to the *time_limit* attribute.

Type `str` or `float` or `int`

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.
>
> Changed in version 3.4: Overriding this method directly in no longer allowed. 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.
>
> Changed in version 3.4: Overriding this method directly in no longer allowed. 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.

Changed in version 3.4: Overriding this method directly in no longer allowed. 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.

Changed in version 3.4: Overriding this method directly in no longer allowed. 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.

Changed in version 3.4: Overriding this method directly in no longer allowed. See here for more details.

container_platform = None
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=None*, **args*, ***kwargs*)
Add a dependency to another test.

Parameters

- **target** – The name of the test that this one will depend on.
- **how** – A callable that defines how the test cases of this test depend on the the test cases of the target test. This callable should accept two arguments:
  - The source test case (i.e., a test case of this test) represented as a two-element tuple containing the names of the partition and the environment of the current test case.
  - Test destination test case (i.e., a test case of the target test) represented as a two-element tuple containing the names of the partition and the environment of the current target test case.

  It should return `True` if a dependency between the source and destination test cases exists, `False` otherwise.

  This function will be called multiple times by the framework when the test DAG is constructed, in order to determine the connectivity of the two tests.

  In the following example, this test depends on `T1` when their partitions match, otherwise their test cases are independent.

```
def by_part(src, dst):
    p0, _ = src
    p1, _  = dst
    return p0 == p1

self.depends_on('T0', how=by_part)
```

  The framework offers already a set of predefined relations between the test cases of inter-dependent tests. See the *reframe.utility.udeps* for more details.

  The default `how` function is *reframe.utility.udeps.by_case()*, where test cases on different partitions and environments are independent.

See also:

- *How Test Dependencies Work In ReFrame*
- *Test Case Dependencies Management*

New in version 2.21.

Changed in version 3.3: Dependencies between test cases from different partitions are now allowed. The `how` argument now accepts a callable.

Deprecated since version 3.3: Passing an integer to the `how` argument as well as using the `subdeps` argument is deprecated.

descr
A detailed description of the test.

Type `str`

Default `self.name`

exclusive_access = False
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': [
    {
        'name': 'gpu',
        'options': ['--gres=gpu:{num_gpus_per_node}']
    },
    {
        'name': 'datawarp',
        'options': [
            '#DW jobdw capacity={capacity}',
            '#DW stage_in source={stagein_src}'
        ]
    }
]
```

A regression test may then 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*, *part=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 test'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 []

Changed in version 3.3: This field accepts now also file glob patterns.

local = False
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 = None
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 = None
Number of CPUs per task required by this test.

Ignored if None.

Type integral or `None`

Default `None`

num_gpus_per_node = 0
Number of GPUs per node required by this test. This attribute is translated internally to the `_rfm_gpu` resource. For more information on test resources, have a look at the *extra_resources* attribute.

Type integral

Default `0`

num_tasks = 1
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 = None
Number of tasks per core required by this test.

Ignored if `None`.

Type integral or `None`

Default `None`

num_tasks_per_node = None
Number of tasks per node required by this test.

Ignored if `None`.

Type integral or `None`

Default `None`

num_tasks_per_socket = None
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 = None
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()
See *run_complete()*.

Deprecated since version 3.2.

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 []

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.
>
> Changed in version 3.4: Overriding this method directly in no longer allowed. See here for more details.

run_complete()
Check if the run phase has completed.

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:** You may not override this method directly unless you are in special test. See here for more details.
>
> Changed in version 3.4: Overriding this method directly in no longer allowed. See here for more details.

run_wait()
Wait for the run phase of this test to finish.

Raises *reframe.core.exceptions.ReframeError* – In case of errors.

> **Warning:** You may not override this method directly unless you are in special test. See here for more details.
>
> Changed in version 3.4: Overriding this method directly in no longer allowed. See here for more details.

sanity_patterns
Refer to the *ReFrame Tutorials* for concrete usage examples.

If not set a sanity error will be raised during sanity checking.

Type A deferrable expression (i.e., the result of a *sanity function*)

Default `required`

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 `required`.

If a test doesn't care about its output, this must be stated explicitly as follows:

```
self.sanity_patterns = sn.assert_true(1)
```

Changed in version 3.6: The default value has changed from `None` to `required`.

---

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.
>
> Changed in version 3.4: Overriding this method directly in no longer allowed. See here for more details.

skip(*msg=None*)
Skip test.

Parameters **msg** – A message explaining why the test was skipped.

New in version 3.5.1.

skip_if(*cond*, *msg=None*)
Skip test if condition is true.

Parameters

- **cond** – The condition to check for skipping the test.
- **msg** – A message explaining why the test was skipped.

New in version 3.5.1.

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 = 'src'
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` or `None` if a run job has not yet been created.

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` or `None` if a run job has not yet been created.

strict_check = True

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()

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 = None

Time limit for this test.

Time limit is specified as a string in the form `<days>d<hours>h<minutes>m<seconds>s` or as number of seconds. If set to `None`, the `time_limit` of the current system partition will be used.

Type `str` or `float` or `int`

Default `None`

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.

Changed in version 3.2: - The old syntax using a `(h, m, s)` tuple is dropped. - Support of *timedelta* objects is dropped. - Number values are now accepted.

Changed in version 3.5.1: The default value is now `None` and it can be set globally per partition via the configuration.

use_multithreading = None

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 * is in the list then all programming environments are supported by this test.

Type List[str]

Default required

Note: Changed in version 2.12: Programming environments can now be specified using wildcards.

Changed in version 2.17: Support for wildcards is dropped.

Changed in version 3.3: Default value changed from [] to None.

Changed in version 3.6: Default value changed from None to required.

---

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 None

Changed in version 3.3: Default value changed from [] to None.

Changed in version 3.6: Default value changed from None to required.

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()
See *run_wait()*.

Deprecated since version 3.2.

class reframe.core.pipeline.**RunOnlyRegressionTest**(**args*, ***kwargs*)
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()*.

setup(*partition*, *environ*, ***job_opts*)
The setup stage of the regression test pipeline.

Similar to the *RegressionTest.setup()*, except that no build job is created for this test.

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.

Deprecated since version 3.6.0: Please use the *parameter()* built-in instead.

@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.

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 `'>=3.5.0'` will allow the following test to be loaded only by ReFrame 3.5.0 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('3.5.1', '>=3.5.6')`, in which case the test will be selected if *any* of the versions is satisfied, even if the versions specifications are conflicting.

New in version 2.13.

Changed in version 3.5.0: Passing ReFrame version numbers that do not comply with the semantic versioning specification is deprecated. Examples of non-compliant version numbers are `3.5` and `3.5-dev0`. These should be written as `3.5.0` and `3.5.0-dev.0`.

@reframe.core.decorators.**simple_test**
Class decorator for registering 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

New in version 2.20.

Pipeline hooks is an easy way to perform operations while the test traverses the execution pipeline. You can attach arbitrary functions to run before or after any pipeline stage, which are called *pipeline hooks*. Multiple hooks can be attached before or after the same pipeline stage, in which case the order of execution will match the order in which the functions are defined in the class body of the test. A single hook can also be applied to multiple stages and it will be executed multiple times. All pipeline hooks of a test class are inherited by its subclasses. Subclasses may override a pipeline hook of their parents by redefining the hook function and re-attaching it at the same pipeline stage. There are seven pipeline stages where you can attach test methods: `init`, `setup`, `compile`, `run`, `sanity`, `performance` and `cleanup`. The `init` stage is not a real pipeline stage, but it refers to the test initialization.

Hooks attached to any stage will run exactly before or after this stage executes. So although a "post-init" and a "pre-setup" hook will both run *after* a test has been initialized and *before* the test goes through the first pipeline stage, they will execute in different times: the post-init hook will execute *right after* the test is initialized. The framework will then continue with other activities and it will execute the pre-setup hook *just before* it schedules the test for executing its setup stage.

`@reframe.core.decorators.`**`run_after`**`(`*`stage`*`)`
Decorator for attaching a test method to a pipeline stage.

This is analogous to the *run_before*, except that `'init'` can also be used as the `stage` argument. In this case, the hook will execute right after the test is initialized (i.e. after the `__init__()` method is called), before entering the test's pipeline. In essence, a post-init hook is equivalent to defining additional `__init__()` functions in the test. All the other properties of pipeline hooks apply equally here. The following code

```
@rfm.run_after('init')
def foo(self):
    self.x = 1
```

is equivalent to

```
def __init__(self):
    self.x = 1
```

Changed in version 3.5.2: Add the ability to define post-init hooks in tests.

`@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'`.

`@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.

Additionally, this decorator will attach the function to run *after* the test's setup phase, but *before* any other "post_setup" pipeline hook.

This decorator is also directly available under the `reframe` module.

New in version 2.21.

Builtins

New in version 3.4.2.

ReFrame provides built-in functions that facilitate the creation of extensible tests (i.e. a test library). These *builtins* are intended to be used directly in the class body of the test, allowing the ReFrame internals to *pre-process* their input before the actual test creation takes place. This provides the ReFrame internals with further control over the user's input, making the process of writing regression tests less error-prone thanks to a better error checking. In essence, these builtins exert control over the test creation, and they allow adding and/or modifying certain attributes of the regression test.

`RegressionTest.`**`parameter`**(*values=None*, *inherit_params=False*, *filter_params=None*)

Inserts or modifies a regression test parameter. If a parameter with a matching name is already present in the parameter space of a parent class, the existing parameter values will be combined with those provided by this method following the inheritance behavior set by the arguments `inherit_params` and `filter_params`. Instead, if no parameter with a matching name exists in any of the parent parameter spaces, a new regression test parameter is created. A regression test can be parameterized as follows:

```
class Foo(rfm.RegressionTest):
    variant = parameter(['A', 'B'])
    # print(variant) # Error: a parameter may only be accessed from the class␣
↪instance.

    @rfm.run_after('init')
    def do_something(self):
        if self.variant == 'A':
            do_this()
        else:
            do_other()
```

One of the most powerful features of these built-in functions is that they store their input information at the class level. However, a parameter may only be accessed from the class instance and accessing it directly from the class body is disallowed. With this approach, extending or specializing an existing parameterized regression test becomes straightforward, since the test attribute additions and modifications made through built-in functions in the parent class are automatically inherited by the child test. For instance, continuing with the example above, one could override the `do_something()` hook in the `Foo` regression test as follows:

```
class Bar(Foo):
    @rfm.run_after('init')
    def do_something(self):
        if self.variant == 'A':
            override_this()
        else:
            override_other()
```

Parameters

- **`values`** – A list containing the parameter values. If no values are passed when creating a new parameter, the parameter is considered as *declared* but not *defined* (i.e. an abstract

parameter). Instead, for an existing parameter, this depends on the parameter's inheritance behaviour and on whether any values where provided in any of the parent parameter spaces.

- **inherit_params** – If `False`, no parameter values that may have been defined in any of the parent parameter spaces will be inherited.
- **filter_params** – Function to filter/modify the inherited parameter values that may have been provided in any of the parent parameter spaces. This function must accept a single argument, which will be passed as an iterable containing the inherited parameter values. This only has an effect if used with `inherit_params=True`.

`RegressionTest.`**variable**(**types*, *value=None*)

Inserts a new regression test variable. Declaring a test variable through the *variable()* built-in allows for a more robust test implementation than if the variables were just defined as regular test attributes (e.g. `self.a = 10`). Using variables declared through the *variable()* built-in guarantees that these regression test variables will not be redeclared by any child class, while also ensuring that any values that may be assigned to such variables comply with its original declaration. In essence, declaring test variables with the *variable()* built-in removes any potential test errors that might be caused by accidentally overriding a class attribute. See the example below.

```
class Foo(rfm.RegressionTest):
    my_var = variable(int, value=8)
    not_a_var = my_var - 4

    @rfm.run_after('init')
    def access_vars(self):
        print(self.my_var) # prints 8.
        # self.my_var = 'override' # Error: my_var must be an int!
        self.not_a_var = 'override' # However, this would work. Dangerous!
        self.my_var = 10 # tests may also assign values the standard way
```

Here, the argument `value` in the *variable()* built-in sets the default value for the variable. This value may be accessed directly from the class body, as long as it was assigned before either in the same class body or in the class body of a parent class. This behavior extends the standard Python data model, where a regular class attribute from a parent class is never available in the class body of a child class. Hence, using the *variable()* built-in enables us to directly use or modify any variables that may have been declared upstream the class inheritance chain, without altering their original value at the parent class level.

```
class Bar(Foo):
    print(my_var) # prints 8
    # print(not_a_var) # This is standard Python and raises a NameError

    # Since my_var is available, we can also update its value:
    my_var = 4

    # Bar inherits the full declaration of my_var with the original type-checking.
    # my_var = 'override' # Wrong type error again!

    @rfm.run_after('init')
    def access_vars(self):
        print(self.my_var) # prints 4
        print(self.not_a_var) # prints 4


print(Foo.my_var) # prints 8
print(Bar.my_var) # prints 4
```

Here, `Bar` inherits the variables from `Foo` and can see that `my_var` has already been declared in the parent

class. Therefore, the value of `my_var` is updated ensuring that the new value complies to the original variable declaration. However, the value of `my_var` at `Foo` remains unchanged.

These examples above assumed that a default value can be provided to the variables in the bases tests, but that might not always be the case. For example, when writing a test library, one might want to leave some variables undefined and force the user to set these when using the test. As shown in the example below, imposing such requirement is as simple as not passing any `value` to the *`variable()`* built-in, which marks the given variable as *required*.

```
# Test as written in the library
class EchoBaseTest(rfm.RunOnlyRegressionTest):
  what = variable(str)

  valid_systems = ['*']
  valid_prog_environs = ['PrgEnv-gnu']

  @rfm.run_before('run')
  def set_exec_and_sanity(self):
      self.executable = f'echo {self.what}'
      self.sanity_patterns = sn.assert_found(fr'{self.what}')


# Test as written by the user
@rfm.simple_test
class HelloTest(EchoBaseTest):
  what = 'Hello'


# A parameterized test with type-checking
@rfm.simple_test
class FoodTest(EchoBaseTest):
  param = parameter(['Bacon', 'Eggs'])

  @rfm.run_after('init')
  def set_vars_with_params(self):
    self.what = self.param
```

Similarly to a variable with a value already assigned to it, the value of a required variable may be set either directly in the class body, on the `__init__()` method, or in any other hook before it is referenced. Otherwise an error will be raised indicating that a required variable has not been set. Conversely, a variable with a default value already assigned to it can be made required by assigning it the `required` keyword.

```
class MyRequiredTest(HelloTest):
  what = required
```

Running the above test will cause the `set_exec_and_sanity()` hook from `EchoBaseTest` to throw an error indicating that the variable `what` has not been set.

Parameters

- **`types`** – the supported types for the variable.
- **`value`** – the default value assigned to the variable. If no value is provided, the variable is set as `required`.
- **`field`** – the field validator to be used for this variable. If no field argument is provided, it defaults to `reframe.core.fields.TypedField`. Note that the field validator provided by this argument must derive from `reframe.core.fields.Field`.

Environments and Systems

class reframe.core.environments.**Environment**(*name*, *modules=None*, *variables=None*)
Bases: reframe.utility.jsonext.JSONSerializable

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 modules_detailed
A view of the modules associated with this environment in a detailed format.

Each module is represented as a dictionary with the following attributes:

- name: the name of the module.
- collection: True if the module name refers to a module collection.

Type List[Dict[str, object]]

New in version 3.3.

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.**DeviceType**(*device_info*)
Bases: reframe.utility.jsonext.JSONSerializable

A representation of a device inside ReFrame.

New in version 3.5.0.

Warning: Users may not create *DeviceType* objects directly.

property arch
The architecture of the device.

Type str or None

property device_type
The type of the device.

Type str or None

property info
All the available information from the configuration.

Type dict

property num_devices
Number of devices of this type.

It will return 1 if it wasn't set in the configuration.

Type integral

class reframe.core.systems.**ProcessorType**(*processor_info*)
Bases: reframe.utility.jsonext.JSONSerializable

A representation of a processor inside ReFrame.

New in version 3.5.0.

Warning: Users may not create *ProcessorType* objects directly.

property arch
The microarchitecture of the processor.

Type str or None

property info
All the available information from the configuration.

Type dict

property num_cores
Total number of cores.

Type integral or None

property num_cores_per_numa_node
Number of cores per NUMA node.

Type integral or None

property num_cores_per_socket
Number of cores per socket.

Type integral or None

property num_cpus
Number of logical CPUs.

Type integral or None

property num_cpus_per_core
Number of logical CPUs per core.

Type integral or None

property num_cpus_per_socket
Number of logical CPUs per socket.

Type integral or None

property num_numa_nodes
Number of NUMA nodes.

Type integral or None

property num_sockets
Number of sockets.

Type integral or None

property topology
Processor topology.

Type Dict[str, obj] or None

class reframe.core.systems.**System**(*name*, *descr*, *hostnames*, *modules_system*, *preload_env*, *prefix*, *outputdir*, *resourcesdir*, *stagedir*, *partitions*)
Bases: reframe.utility.jsonext.JSONSerializable

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*, *sched_type*, *launcher_type*, *descr*, *access*, *container_environs*, *resources*, *local_env*, *environs*, *max_jobs*, *prepare_cmds*, *processor*, *devices*, *extras*)
Bases: reframe.utility.jsonext.JSONSerializable

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

property devices
A list of devices in the current partition.

New in version 3.5.0.

Type List[reframe.core.systems.DeviceType]

environment(*name*)
Return the partition environment named name.

property environs
The programming environments associated with this system partition.

Type List[ProgEnvironment]

property extras
User defined attributes of the system.

By default, it is an empty dictionary.

New in version 3.5.0.

Type object

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
See *launcher_type*.

Deprecated since version 3.2: Please use *launcher_type* instead.

property launcher_type
The type of the backend launcher of this partition.

New in version 3.2.

Type 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 prepare_cmds
Commands to be emitted before loading the modules.

Type List[str]

property processor
Processor information for the current partition.

New in version 3.5.0.

Type *reframe.core.systems.ProcessorType*

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 backend scheduler of this partition.

Type reframe.core.schedulers.JobScheduler.

Note: Changed in version 2.8: Prior versions returned a string representing the scheduler and job launcher combination.

Changed in version 3.2: The property now stores a `JobScheduler` instance.

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_exclusive_access=None*, *sched_options=None*)

Bases: `reframe.utility.jsonext.JSONSerializable`

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.

property exitcode

The exit code of this job.

This may or may not be set depending on the scheduler backend.

New in version 2.21.

Type `int` or `None`

property jobid

The ID of this job.

New in version 2.21.

Changed in version 3.2: Job ID type is now a string.

Type `str` 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*

property nodelist

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.

New in version 2.17.

Type `List[str]` or `None`

options
Options to be passed to the backend job scheduler.

Type `List[str]`

Default `[]`

property state
The state of this job.

The value of this field is scheduler-specific.

New in version 2.21.

Type :class`str` or `None`

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.

abstract command(*job*)
The launcher command to be emitted for a specific job.

Launcher backends provide concrete implementations of this method.

Parameters **job** – A job descriptor.

Returns the basic launcher command as a list of tokens.

options
List of options to be passed to the job launcher invocation.

Type `List[str]`

Default `[]`

run_command(*job*)
The full launcher command to be emitted for a specific job.

This includes any user options.

Parameters **job** – a job descriptor.

Returns the launcher command as a string.

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.

command(*job*)

The launcher command to be emitted for a specific job.

Launcher backends provide concrete implementations of this method.

Parameters **job** – A job descriptor.

Returns the basic launcher command as a list of tokens.

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.

available_modules(*substr=None*)
Return a list of available modules that contain `substr` in their name.

Return type List[str]

conflicted_modules(*name*, *collection=False*, *path=None*)
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.

Parameters

- **name** – The name of the module.
- **collection** – The module is a "module collection" (TMod4/LMod only).
- **path** – The path where the module resides if not in the default `MODULEPATH`.

Returns A list of conflicting module names.

Changed in version 3.3: The `collection` argument is added.

Changed in version 3.5.0: The `path` argument is added.

emit_load_commands(*name*, *collection=False*, *path=None*)
Return the appropriate shell commands for loading a module.

Module mappings are not taken into account by this function.

Parameters

- **name** – The name of the module to load.
- **collection** – The module is a "module collection" (TMod4/LMod only)
- **path** – The path where the module resides if not in the default `MODULEPATH`.

Returns A list of shell commands.

Changed in version 3.3: The `collection` argument was added and module mappings are no more taken into account by this function.

Changed in version 3.5.0: The `path` argument is added.

emit_unload_commands(*name*, *collection=False*, *path=None*)
Return the appropriate shell commands for unloading a module.

Module mappings are not taken into account by this function.

Parameters

- **name** – The name of the module to unload.

- **collection** – The module is a "module collection" (TMod4/LMod only)
- **path** – The path where the module resides if not in the default `MODULEPATH`.

Returns A list of shell commands.

Changed in version 3.3: The `collection` argument was added and module mappings are no more taken into account by this function.

Changed in version 3.5.0: The `path` argument is added.

execute(*cmd*, **args*)
Execute an arbitrary module command.

Parameters

- **cmd** – The command to execute, e.g., `load`, `restore` etc.
- **args** – The arguments to pass to the command.

Returns The command output.

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*, *collection=False*, *path=None*, *force=False*)
Load the module `name`.

Parameters

- **collection** – The module is a "module collection" (TMod4/Lmod only)
- **path** – The path where the module resides if not in the default `MODULEPATH`.
- **force** – If 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 A list of two-element tuples, where each tuple contains the module that was loaded and the list of modules that had to be unloaded first due to conflicts. This list will be normally of size one, but it can be longer if there is mapping that maps module `name` to multiple other modules.

Changed in version 3.3: - The `collection` argument is added. - This function now returns a list of tuples.

Changed in version 3.5.0: - The `path` argument is added. - The `force` argument is now the last argument.

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*, *collection=False*, *path=None*)
Unload module `name`.

Parameters

- **name** – The name of the module to unload. If module `name` is resolved to multiple real modules, all the referred to modules will be unloaded in reverse order.
- **collection** – The module is a "module collection" (TMod4 only)
- **path** – The path where the module resides if not in the default `MODULEPATH`.

Changed in version 3.3: The `collection` argument was added.

Changed in version 3.5.0: The `path` argument is added.

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.

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 `[]`

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.

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.**EasyBuild**

Bases: *reframe.core.buildsystems.BuildSystem*

A build system for building test code using EasyBuild.

ReFrame will use EasyBuild to build and install the code in the test's stage directory by default. ReFrame uses environment variables to configure EasyBuild for running, so Users can pass additional options to the `eb` command and modify the default behaviour.

New in version 3.5.0.

easyconfigs

The list of easyconfig files to build and install. This field is required.

Type List[str]

Default []

emit_package

Instruct EasyBuild to emit a package for the built software. This will essentially pass the `--package` option to `eb`.

Type bool

Default []

options

Options to pass to the `eb` command.

Type List[str]

Default []

package_opts

Options controlling the package creation from EasyBuild. For each key/value pair of this dictionary, ReFrame will pass `--package-{key}={val}` to the EasyBuild invocation.

Type Dict[str, str]

Default {}

prefix

Default prefix for the EasyBuild installation.

Relative paths will be appended to the stage directory of the test. ReFrame will set the following environment variables before running EasyBuild.

```
export EASYBUILD_BUILDPATH={prefix}/build
export EASYBUILD_INSTALLPATH={prefix}
export EASYBUILD_PREFIX={prefix}
export EASYBUILD_SOURCEPATH={prefix}
```

Users can change these defaults by passing specific options to the `eb` command.

Type str

Default easybuild

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`.

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`*.

`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.

command
The command to be executed within the container.

If no command is given, then the default command of the corresponding container image is going to be executed.

New in version 3.5.0: Changed the attribute name from *commands* to *command* and its type to a string.

Type str or None

Default None

commands
The commands to be executed within the container.

Deprecated since version 3.5.0: Please use the *command* field instead.

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). The stage directory of the ReFrame test is always mounted under /rfm_workdir inside the container, independelty of this field.

Type list[tuple[str, str]]

Default []

options
Additional options to be passed to the container runtime when executed.

Type list[str]

Default []

pull_image
Pull the container image before running.

This does not have any effect for the *Singularity* container platform.

New in version 3.5.

Type bool

Default True

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.

Deprecated since version 3.5: Please use the *options* field to set the working directory.

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*.

Mapping of Test Attributes to Job Scheduler Backends

Test attribute	Slurm option	Torque option	PBS option
num_tasks	--ntasks	[1] -l nodes={num_tasks//num_tasks_per_node}:ppn={num_tasks_per_node}	-l select={num_tasks//num_tasks_per_node}:mpiprocs={num_tasks_per_node}
num_tasks_per_node	--ntasks-per-node	see num_tasks	see num_tasks
num_tasks_per_core	n/a	n/a	n/a
num_tasks_per_socket	n/a	n/a	n/a
num_cpus_per_task	--cpus-per-task	see num_tasks	see num_tasks
time_limit	--time=hh:mm:ss	-l walltime=hh:mm:ss	-l walltime=hh:mm::ss
exclusive_access	--exclusive	n/a	n/a
use_smt	--hint=[no]multithread	n/a	n/a

If any of the attributes is set to `None` it will not be emitted at all in the job script. In cases that the attribute is required, it will be set to `1`.

[1] The `--nodes` option may also be emitted if the `use_nodes_option` scheduler configuration parameter is set.

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`.
- **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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. The re.MULTILINE flag is set for the pattern search.
- **filename** – The name of the file to examine or a file descriptor as in `open()`. Any `OSError` raised while processing the file will be propagated as a *reframe.core.exceptions.SanityError*.

- **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.
- **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_found_s**(*patt*, *string*, *msg=None*)
Assert that regex pattern `patt` is found in the string `string`.

Parameters

- **patt** – as in *assert_found()*.
- **string** – The string to examine.
- **msg** – as in *assert_found()*. You may use `{0}` ... `{N}` as placeholders for the function arguments.

Returns `True` on success.

Raises ***reframe.core.exceptions.SanityError*** – if assertion fails.

New in version 3.4.1.

`reframe.utility.sanity.`**assert_ge**(*a*, *b*, *msg=None*)
Assert that `a >= b`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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_found_s**(*patt*, *string*, *msg=None*)
Assert that regex pattern `patt` is not found in `string`.

This is the inverse of *assert_found_s()*.

Returns `True` on success.

Raises ***reframe.core.exceptions.SanityError*** – if assertion fails.

New in version 3.4.1.

reframe.utility.sanity.**assert_not_in**(*item*, *container*, *msg=None*)
Assert that `item` is not in `container`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0}` ... `{N}` as placeholders for the function arguments.

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.

- **msg** – The error message to use if the assertion fails. You may use `{0} ... {N}` as placeholders for the function arguments.

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`.

Parameters **msg** – The error message to use if the assertion fails. You may use `{0} ... {N}` as placeholders for the function arguments.

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. The re.MULTILINE flag is set for the pattern search.
- **filename** – The name of the file to examine or a file descriptor as in `open()`.
- **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 or iterable of callables taking a single argument and returning a new value. If not an iterable, it will be used to convert the extracted values for all the capturing groups specified in `tag`. Otherwise, each conversion function will be used to convert the value extracted from the corresponding capturing group in `tag`. If more conversion functions are supplied than the corresponding capturing groups in `tag`, the last conversion function will be used for the additional capturing groups.

Returns A list of tuples of converted values extracted from the capturing groups specified in `tag`, if `tag` is an iterable. Otherwise, a list of the converted values extracted from the single capturing group specified in `tag`.

Raises ***reframe.core.exceptions.SanityError*** – In case of errors.

Changed in version 3.1: Multiple regex capturing groups are now supporetd via `tag` and multiple conversion functions can be used in `conv`.

`reframe.utility.sanity.`**`extractall_s`**(*patt*, *string*, *tag=0*, *conv=None*)

Extract all values from the capturing group `tag` of a matching regex `patt` in `string`.

arg patt as in *`extractall()`*.

‘ :arg string: The string to examine.

arg tag as in *`extractall()`*.

arg conv as in *`extractall()`*.

returns same as *`extractall()`*.

New in version 3.4.1.

`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.`**`extractiter_s`**(*patt*, *string*, *tag=0*, *conv=None*)

Get an iterator over the values extracted from the capturing group `tag` of a matching regex `patt` in `string`.

This function is equivalent to *`extractall_s()`* except that it returns a generator object, instead of a list, which you can use to iterate over the extracted values.

New in version 3.4.1.

`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.**extractsingle_s**(*patt*, *string*, *tag=0*, *conv=None*, *item=0*)
Extract a single value from the capturing group `tag` of a matching regex `patt` in `string`.

This function is equivalent to `extractall_s(patt, string, tag, conv)[item]`, except that it raises a `SanityError` if `item` is out of bounds.

Parameters

- **patt** – as in *extractall_s()*.
- **string** – as in *extractall_s()*.
- **tag** – as in *extractall_s()*.
- **conv** – as in *extractall_s()*.
- **item** – the specific element to extract.

Returns The extracted value.

Raises ***reframe.core.exceptions.SanityError*** – In case of errors.

New in version 3.4.1.

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. The re.MULTILINE flag is set for the pattern search.
- **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.**findall_s**(*patt*, *string*)
Get all matches of regex `patt` in `string`.

Parameters

- **patt** – as in *findall()*
- **string** – The string to examine.

Returns same as `finall()`.

New in version 3.4.1.

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.**finditer_s**(*patt*, *string*)
Get an iterator over the matches of the regex `patt` in `string`.

This function is equivalent to *findall_s()* except that it returns a generator object instead of a list, which you can use to iterate over the raw matches.

New in version 3.4.1.

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.**path_exists**(*path*)
Replacement for the `os.path.exists()` function.

New in version 3.4.

reframe.utility.sanity.**path_isdir**(*path*)
Replacement for the `os.path.isdir()` function.

New in version 3.4.

reframe.utility.sanity.**path_isfile**(*path*)
Replacement for the os.path.isfile() function.

New in version 3.4.

reframe.utility.sanity.**path_islink**(*path*)
Replacement for the os.path.islink() function.

New in version 3.4.

reframe.utility.sanity.**print**(*obj*, *, *sep=' '*, *end='\n'*, *file=None*, *flush=False*)
Replacement for the built-in print() function.

The only difference is that this function takes a *single* object argument and it returns that, 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.

Changed in version 3.4: This function accepts now a single object argument in contrast to the built-in print() function, which accepts multiple.

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.

Utility Functions

New in version 3.3.

This is a collection of utility functions and classes that are used by the framework but can also be useful when writing regression tests. Functions or classes marked as draft should be used with caution, since they might change or be replaced without a deprecation warning.

General Utilities

class reframe.utility.**MappingView**(*mapping*)
Bases: collections.abc.Mapping

A read-only view of a mapping.

See collections.abc.Mapping for a list of supported of operations.

get(*key*, *default=None*)
Return the value mapped to `key` or `default`, if `key` does not exist.

Parameters

- **key** – The key to look up.
- **default** – The default value to return if the key is not present.

Returns The value associated to the requested key.

items()
Return a set-like object providing a view on the underlying mapping's items.

keys()
Return a set-like object providing a view on the underlying mapping's keys.

values()
Return a set-like object providing a view on the underlying mapping's values.

class reframe.utility.**OrderedSet**(**args*)
Bases: collections.abc.MutableSet

An ordered set.

This container behaves like a normal set but remembers the insertion order of its elements. It can also interoperate with standard Python sets.

Operations between ordered sets respect the order of the elements of the operands. For example, if `x` and `y` are both ordered sets, then `x | y` will be a new ordered set with the (unique) elements of `x` and `y` in the order they appear in `x` and `y`. The same holds for all the other set operations.

add(*elem*)
See same method in set.

clear()
See same method in set.

difference(**others*)
See same method in set.

discard(*elem*)
See same method in set.

intersection(**others*)
See same method in set.

isdisjoint(*other*)
See same method in set.

issubset(*other*)
See same method in set.

issuperset(*other*)
See same method in set.

pop()
See same method in set.

remove(*elem*)
See same method in set.

symmetric_difference(*other*)
See same method in set.

union(**others*)
See same method in set.

class reframe.utility.**ScopedDict**(*mapping={}*, *scope_sep=':'*, *global_scope='*'*)
Bases: collections.UserDict

This is a special dictionary that imposes scopes on its keys.

When a key is not found, it will be searched up in the scope hierarchy. If not found even at the global scope, a KeyError will be raised.

A scoped dictionary is initialized using a two-level normal dictionary that defines the different scopes and the keys inside them. Scopes can be nested by concatenating them using the : separator by default: scope:subscope. Below is an example of a scoped dictionary that also demonstrates key lookup:

```
d = ScopedDict({
    'a': {'k1': 1, 'k2': 2},
    'a:b': {'k1': 3, 'k3': 4},
    '*': {'k1': 7, 'k3': 9, 'k4': 10}
})

assert d['a:k1'] == 1    # resolved in the scope 'a'
assert d['a:k3'] == 9    # resolved in the global scope
assert d['a:b:k1'] == 3  # resolved in the scope 'a:b'
assert d['a:b:k2'] == 2  # resolved in the scope 'a'
assert d['a:b:k4'] == 10 # resolved in the global scope
d['a:k5']    # KeyError
d['*:k2']    # KeyError
```

If no scope is specified in the key lookup, the global scope is assumed. For example, d['k1'] will return 7. The syntaxes d[':k1'] and d['*:k1'] are all equivalent. If you try to retrieve a whole scope, e.g., d['a:b'], KeyError will be raised. For retrieving scopes, you should use the *scope()* function.

Key deletion follows the same resolution mechanism as key retrieval, except that you are allowed to delete whole scopes. For example, del d['*'] will delete the global scope, such that subsequent access of d['a:k3'] will raise a KeyError. If a key specification matches both a key and scope, the key will be deleted and not the scope.

Parameters

- **mapping** – A two-level mapping of the form

```
{
     scope1: {k1: v1, k2: v2},
     scope2: {k1: v1, k3: v3}
}
```

  Both the scope keys and the actual dictionary keys must be strings, otherwise a TypeError will be raised.

- **scope_sep** – A character that separates the scopes.
- **global_scope** – A key that represents the global scope.

property global_scope_mark
The key representing the global scope of this dictionary.

scope(*name*)
Retrieve a whole scope.

Parameters **scope** – The name of the scope to retrieve.

Returns A dictionary with the keys that are within the requested scope.

property scope_separator
The scope separator of this dictionary.

update(*other*)
Update this dictionary from the values of a two-level mapping as described above.

Parameters **other** – A two-level mapping defining scopes and keys.

class reframe.utility.**SequenceView**(*container*)
Bases: collections.abc.Sequence

A read-only view of a sequence.

See collections.abc.Sequence for a list of supported of operations.

Parameters **container** – The container to create a view on.

Raises **TypeError** – If the container does not fulfill the collections.abc.Sequence interface.

count(*value*)
Count occurrences of value in the container.

Parameters **value** – The value to search for.

Returns The number of occurrences.

index(*value*, *start=0*, *stop=None*)
Return the first index of value.

Parameters

- **value** – The value to search for.
- **start** – The position where the search starts.
- **stop** – The position where the search stops. The element at this position is not looked at. If None, this equals to the sequence's length.

Returns The index of the first element found that equals value.

Raises **ValueError** – if the value is not present.

reframe.utility.**allx**(*iterable*)
Same as the built-in all(), except that it returns False if iterable is empty.

reframe.utility.**attr_validator**(*validate_fn*)
Validate object attributes recursively.

This returns a function which you can call with the object to check. It will return True if the validate_fn() returns True for all object attributes recursively. If the object to be validated is an iterable, its elements will be validated individually.

Parameters **validate_fn** – A callable that validates an object. It takes a single argument, which is the object to validate.

Returns A validation function that will perform the actual validation. It accepts a single argument, which is the object to validate. It returns a two-element tuple, containing the result of the validation as a boolean and a formatted string indicating the faulty attribute.

Note: Objects defining `__slots__` are passed directly to the `validate_fn` function.

New in version 3.3.

`reframe.utility.`**`decamelize`**(*s*, *delim='_'*)

Decamelize a string.

For example, `MyBaseClass` will be converted to `my_base_class`. The delimiter may be changed by setting the `delim` argument.

Parameters

- **`s`** – A string in camel notation.
- **`delim`** – The delimiter that will be used to separate words.

Returns The converted string.

`reframe.utility.`**`find_modules`**(*substr*, *environ_mapping=None*)

Return all modules in the current system that contain `substr` in their name.

This function is a generator and will yield tuples of partition, environment and module combinations for each partition of the current system and for each environment of a partition.

The `environ_mapping` argument allows you to map module name patterns to ReFrame environments. This is useful for flat module name schemes, in order to avoid incompatible combinations of modules and environments.

You can use this function to parametrize regression tests over the available environment modules. The following example will generate tests for all the available `netcdf` packages in the system:

```
@rfm.simple_test
class MyTest(rfm.RegressionTest):
    module_info = parameter(find_modules('netcdf'))

    @rfm.run_after('init')
    def apply_module_info(self):
        s, e, m = self.module_info
        self.valid_systems = [s]
        self.valid_prog_environs = [e]
        self.modules = [m]
        ...
```

The following example shows the use of `environ_mapping` with flat module name schemes. In this example, the toolchain for which the package was built is encoded in the module's name. Using the `environ_mapping` argument we can map module name patterns to ReFrame environments, so that invalid combinations are pruned:

```
my_find_modules = functools.partial(find_modules, environ_mapping={
    r'.*CrayGNU.*': 'PrgEnv-gnu',
    r'.*CrayIntel.*': 'PrgEnv-intel',
    r'.*CrayCCE.*': 'PrgEnv-cray'
})

@rfm.simple_test
class MyTest(rfm.RegressionTest):
```

(continues on next page)

(continued from previous page)

```
    module_info = parameter(my_find_modules('GROMACS'))

    @rfm.run_after('init')
    def apply_module_info(self):
        s, e, m = self.module_info
        self.valid_systems = [s]
        self.valid_prog_environs = [e]
        self.modules = [m]
        ...
```

Parameters

- **substr** – A substring that the returned module names must contain.
- **environ_mapping** – A dictionary mapping regular expressions to environment names.

Returns An iterator that iterates over tuples of the module, partition and environment name combinations that were found.

reframe.utility.**import_module_from_file**(*filename*, *force=False*)

Import module from file.

Parameters

- **filename** – The path to the filename of a Python module.
- **force** – Force reload of module in case it is already loaded.

Returns The loaded Python module.

reframe.utility.**is_copyable**(*obj*)

Check if an object can be copied with `copy.deepcopy()`, without performing the copy.

This is a superset of *`is_picklable()`*. It returns `True` also in the following cases:

- The object defines a `__copy__()` method.
- The object defines a `__deepcopy__()` method.
- The object is a function.
- The object is a builtin type.

New in version 3.3.

reframe.utility.**is_picklable**(*obj*)

Check if an object can be pickled.

New in version 3.3.

reframe.utility.**longest**(**iterables*)

Return the longest sequence.

This function raises a `TypeError` if any of the iterables is not `Sized`.

Parameters **iterables** – The iterables to check.

Returns The longest iterable.

reframe.utility.**nodelist_abbrev**(*nodes*)

Create an abbreviated string representation of the node list.

For example, the node list

```
['nid001', 'nid002', 'nid010', 'nid011', 'nid012', 'nid510', 'nid511']
```

will be abbreviated as follows:

```
nid00[1-2],nid0[10-12],nid51[0-1]
```

New in version 3.5.3.

Parameters **nodes** – The node list to abbreviate.

Returns The abbreviated list representation.

reframe.utility.**ppretty**(*value*, *htchar=' '*, *lfchar='\n'*, *indent=4*, *basic_offset=0*, *repr=<built-in function repr>*)

Format value in a pretty way.

If value is a container, this function will recursively format the container's elements.

Parameters

- **value** – The value to be formatted.
- **htchar** – Horizontal-tab character.
- **lfchar** – Linefeed character.
- **indent** – Number of `htchar` characters for every indentation level.
- **basic_offset** – Basic offset for the representation, any additional indentation space is added to the `basic_offset`.
- **repr** – A *repr()*-like function that will be used for printing values. This function is allowed to accept all the arguments of *ppretty()* except the `repr` argument.

Returns A formatted string of `value`.

reframe.utility.**repr**(*obj*, *htchar=' '*, *lfchar='\n'*, *indent=4*, *basic_offset=0*)

A `repr()` replacement function for debugging purposes printing all object attributes recursively.

This function does not follow the standard `repr()` convention, but it prints each object as a set of key/value pairs along with its memory location. It also keeps track of the already visited objects, and abbreviates their representation.

Parameters **obj** – The object to be dumped. For the rest of the arguments, see *ppretty()*.

Returns The formatted object dump.

reframe.utility.**shortest**(**iterables*)

Return the shortest sequence.

This function raises a `TypeError` if any of the iterables is not `Sized`.

Parameters **iterables** – The iterables to check.

Returns The shortest iterable.

System Utilities

class reframe.utility.osext.**change_dir**(*dir_name*)
Bases: object

Context manager to temporarily change the current working directory.

Parameters **dir_name** – The directory to temporarily change to.

reframe.utility.osext.**concat_files**(*dst*, **files*, *sep='\n'*, *overwrite=False*)
Concatenate files into dst.

Parameters

- **dst** – The name of the output file.
- **files** – The files to concatenate.
- **sep** – The separator to use during concatenation.
- **overwrite** – Overwrite the output file if it already exists.

Raises

- **TypeError** – In case files it not an iterable object.
- **ValueError** – In case output already exists and ovewrite is False.

reframe.utility.osext.**copytree**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=<function copy2>*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)
Compatibility version of shutil.copytree() for Python < 3.8.

This function will automatically delegate to shutil.copytree() for Python versions >= 3.8.

reframe.utility.osext.**copytree_virtual**(*src*, *dst*, *file_links=None*, *symlinks=False*, *copy_function=<function copy2>*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)
Copy src to dst, but create symlinks for the files listed in file_links.

If file_links is empty or None, this is equivalent to *copytree()*. The rest of the arguments are passed as-is to *copytree()*. Paths in file_links must be relative to src. If you try to pass '.' in file_links, an OSError will be raised.

reframe.utility.osext.**cray_cdt_version**()
Return the Cray Development Toolkit (CDT) version or None if the version cannot be retrieved.

reframe.utility.osext.**cray_cle_info**(*filename='/etc/opt/cray/release/cle-release'*)
Return the Cray Linux Environment (CLE) release information.

Parameters **filename** – The file that contains the CLE release information

Returns A named tuple with the following attributes that correspond to the release information: release, build, date, arch, network, patchset.

reframe.utility.osext.**expandvars**(*s*)
Expand environment variables in s and perform any command substitution.

This function is the same as os.path.expandvars(), except that it also recognizes the syntax of shell command substitution: $(cmd) or `cmd`.

reframe.utility.osext.**follow_link**(*path*)
Return the final target of a symlink chain.

If `path` is not a symlink, it will be returned as is.

`reframe.utility.osext.`**`force_remove_file`**(*filename*)
Remove filename ignoring `FileNotFoundError`.

`reframe.utility.osext.`**`git_clone`**(*url*, *targetdir=None*)
Clone a git repository from a URL.

Parameters

- **`url`** – The URL to clone from.
- **`targetdir`** – The directory where the repository will be cloned to. If `None`, a new directory will be created with the repository name as if `git clone {url}` was issued.

`reframe.utility.osext.`**`git_repo_exists`**(*url*, *timeout=5*)
Check if URL refers to a valid Git repository.

Parameters

- **`url`** – The URL to check.
- **`timeout`** – Timeout in seconds.

Returns `True` if URL is a Git repository, `False` otherwise or if timeout is reached.

`reframe.utility.osext.`**`git_repo_hash`**(*commit='HEAD'*, *short=True*, *wd=None*)
Return the SHA1 hash of a Git commit.

Parameters

- **`commit`** – The commit to look at.
- **`short`** – Return a short hash. This always corresponds to the first 8 characters of the long hash. We don't rely on Git for the short hash, since depending on the version it might return either 7 or 8 characters.
- **`wd`** – Change to this directory before retrieving the hash. If `None`, ReFrame's install prefix will be used.

Returns The Git commit hash or `None` if the hash could not be retrieved.

`reframe.utility.osext.`**`inpath`**(*entry*, *pathvar*)
Check if entry is in path.

Parameters

- **`entry`** – The entry to look for.
- **`pathvar`** – A path variable in the form *'entry1:entry2:entry3'*.

Returns `True` if the entry exists in the path variable, `False` otherwise.

`reframe.utility.osext.`**`is_interactive`**()
Check if the current Python session is interactive.

`reframe.utility.osext.`**`is_url`**(*s*)
Check if string is a URL.

`reframe.utility.osext.`**`mkstemp_path`**(**args*, ***kwargs*)
Create a temporary file and return its path.

This is a wrapper to `tempfile.mkstemp()` except that it closes the temporary file as soon as it creates it and returns the path.

`args` and `kwargs` passed through to `tempfile.mkstemp()`.

reframe.utility.osext.**osgroup**()
Return the group name of the current OS user.

If the group name cannot be retrieved, None will be returned.

reframe.utility.osext.**osuser**()
Return the name of the current OS user.

If the user name cannot be retrieved, None will be returned.

reframe.utility.osext.**reframe_version**()
Return ReFrame version.

If ReFrame's installation contains the repository metadata and the current version is a pre-release version, the repository's hash will be appended to the actual version.

reframe.utility.osext.**rmtree**(**args*, *max_retries=3*, ***kwargs*)
Persistent version of shutil.rmtree().

If shutil.rmtree() fails with ENOTEMPTY or EBUSY, ignore the error and retry up to max_retries times to delete the directory.

This version of *rmtree()* is mostly provided to work around a race condition between when sacct reports a job as completed and when the Slurm epilog runs. See gh #291 for more information. Furthermore, it offers a work around for NFS file systems where stale file handles may be present during the *rmtree()* call, causing it to throw a busy device/resource error. See gh #712 for more information.

args and kwargs are passed through to shutil.rmtree().

If onerror is specified in kwargs and it is not None, this function is completely equivalent to shutil.rmtree().

Parameters

- **args** – Arguments to be passed through to shutil.rmtree().
- **max_reties** – Maximum number of retries if the target directory cannot be deleted.
- **kwargs** – Keyword arguments to be passed through to shutil.rmtree().

reframe.utility.osext.**run_command**(*cmd*, *check=False*, *timeout=None*, *shell=False*, *log=True*)
Run command synchronously.

This function will block until the command executes or the timeout is reached. It essentially calls *run_command_async()* and waits for the command's completion.

Parameters

- **cmd** – The command to execute as a string or a sequence. See *run_command_async()* for more details.
- **check** – Raise an error if the command exits with a non-zero exit code.
- **timeout** – Timeout in seconds.
- **shell** – Spawn a new shell to execute the command.
- **log** – Log the execution of the command through ReFrame's logging facility.

Returns A subprocess.CompletedProcess object with information about the command's outcome.

Raises

- ***reframe.core.exceptions.SpawnedProcessError*** – If check is True and the command fails.

- ***reframe.core.exceptions.SpawnedProcessTimeout*** – If the command times out.

reframe.utility.osext.**run_command_async**(*cmd*, *stdout=- 1*, *stderr=- 1*, *shell=False*, *log=True*, ***popen_args*)

Run command asynchronously.

A wrapper to `subprocess.Popen` with the following tweaks:

- It always passes `universal_newlines=True` to `Popen`.
- If `shell=False` and `cmd` is a string, it will lexically split `cmd` using `shlex.split(cmd)`.

Parameters

- **cmd** – The command to run either as a string or a sequence of arguments.
- **stdout** – Same as the corresponding argument of `Popen`. Default is `subprocess.PIPE`.
- **stderr** – Same as the corresponding argument of `Popen`. Default is `subprocess.PIPE`.
- **shell** – Same as the corresponding argument of `Popen`.
- **log** – Log the execution of the command through ReFrame's logging facility.
- **popen_args** – Any additional arguments to be passed to `Popen`.

Returns A new `Popen` object.

reframe.utility.osext.**samefile**(*path1*, *path2*)

Check if paths refer to the same file.

If paths exist, this is equivalent to `os.path.samefile()`. If only one of the paths exists and is a symbolic link, it will be followed and its final target will be compared to the other path. If both paths do not exist, a simple string comparison will be performed (after the paths have been normalized).

reframe.utility.osext.**subdirs**(*dirname*, *recurse=False*)

Get the list of subdirectories of `dirname` including `dirname`.

If `recurse` is `True`, this function will retrieve all subdirectories in pre-order.

Parameters

- **dirname** – The directory to start searching.
- **recurse** – If `True`, then recursively search for subdirectories.

Returns The list of subdirectories found.

reframe.utility.osext.**unique_abs_paths**(*paths*, *prune_children=True*)

Get the unique absolute paths from a given list of `paths`.

Parameters

- **paths** – An iterable of paths.
- **prune_children** – Discard paths that are children of other paths in the list.

Raises **TypeError** – In case `paths` it not an iterable object.

Type Checking Utilities

Dynamic recursive type checking of collections.

This module defines types for collections, such as lists, dictionaries etc., that you can use with the `isinstance()` builtin function to recursively type check all the elements of the collection. Suppose you have a list of integers, suchs as `[1, 2, 3]`, the following checks should be true:

```
l = [1, 2, 3]
assert isinstance(l, List[int]) == True
assert isinstance(l, List[float]) == False
```

Aggregate types can be combined in an arbitrary depth, so that you can type check any complex data strcture:

```
d = {'a': [1, 2], 'b': [3, 4]}
assert isisntance(d, Dict) == True
assert isisntance(d, Dict[str, List[int]]) == True
```

This module offers the following aggregate types:

`List[T]`
A list with elements of type `T`.

`Set[T]`
A set with elements of type `T`.

`Dict[K,V]`
A dictionary with keys of type `K` and values of type `V`.

`Tuple[T]`
A tuple with elements of type `T`.

`Tuple[T1,T2,...,Tn]`
A tuple with `n` elements, whose types are exactly `T1`, `T2`, ..., `Tn` in that order.

`Str[patt]`
A string type whose members are all the strings matching the regular expression `patt`.

Implementation details

Internally, this module leverages metaclasses and the `__isinstancecheck__()` method to customize the behaviour of the `isinstance()` builtin.

By implementing also the `__getitem__()` accessor method, this module follows the look-and-feel of the type hints proposed in PEP484. This method returns a new type that is a subtype of the base container type. Using the facilities of `abc.ABCMeta`, builtin types, such as `list`, `str` etc. are registered as subtypes of the base container types offered by this module. The type hierarchy of the types defined in this module is the following (example shown for `List`, but it is analogous for the rest of the types):

```
      List
    /   |
   /    |
  /     |
list  List[T]
```

In the above example `T` may refer to any type, so that `List[List[int]]` is an instance of `List`, but not an instance of `List[int]`.

Test Case Dependencies Management

Managing the test case "micro-dependencies" between two tests.

This module defines a set of basic functions that can be used with the `how` argument of the *reframe.core.pipeline.RegressionTest.depends_on()* function to control how the individual dependencies between the test cases of two tests are formed.

All functions take two arguments, the source and destination vertices of an edge in the test case dependency subgraph that connects two tests. In the relation *"T0 depends on T1"*, the source are the test cases of "T0" and the destination are the test cases of "T1." The source and destination arguments are two-element tuples containing the names of the partition and the environment of the corresponding test cases. These functions return `True` if there is an edge connecting the two test cases or `False` otherwise.

A `how` function will be called by the framework multiple times when the test DAG is built. More specifically, for each test dependency relation, it will be called once for each test case combination of the two tests.

The `how` functions essentially split the test case subgraph of two dependent tests into fully connected components based on the values of their supported partitions and environments.

The *How Test Dependencies Work In ReFrame* page contains more information about test dependencies and shows visually the test case subgraph connectivity that the different `how` functions described here achieve.

New in version 3.3.

`reframe.utility.udeps.`**`by_case`**(*src*, *dst*)
The test cases of two dependent tests will be split by partition and by environment.

Test cases from different partitions and different environments are independent.

`reframe.utility.udeps.`**`by_env`**(*src*, *dst*)
The test cases of two dependent tests will be split by environment.

Test cases from different environments are independent.

`reframe.utility.udeps.`**`by_part`**(*src*, *dst*)
The test cases of two dependent tests will be split by partition.

Test cases from different partitions are independent.

`reframe.utility.udeps.`**`by_xcase`**(*src*, *dst*)
The test cases of two dependent tests will be split by the exclusive disjunction (XOR) of their partitions and environments.

Test cases from the same environment and the same partition are independent.

`reframe.utility.udeps.`**`by_xenv`**(*src*, *dst*)
The test cases of two dependent tests will be split by the exclusive disjunction (XOR) of their environments.

Test cases from the same environment are independent.

`reframe.utility.udeps.`**`by_xpart`**(*src*, *dst*)
The test cases of two dependent tests will be split by the exclusive disjunction (XOR) of their partitions.

Test cases from the same partition are independent.

`reframe.utility.udeps.`**`fully`**(*src*, *dst*)
The test cases of two dependent tests will be fully connected.

ReFrame Errors

When writing ReFrame tests, you don't need to check for any exceptions raised. The runtime will take care of finalizing your test and continuing execution.

Dealing with ReFrame errors is only useful if you are extending ReFrame's functionality, either by modifying its core or by creating new regression test base classes for fulfilling your specific needs.

Warning: This API is considered a developer's API, so it can change from version to version without a deprecation warning.

exception reframe.core.exceptions.**AbortTaskError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised by the runtime inside a regression task to denote that it has been aborted due to an external reason (e.g., keyboard interrupt, fatal error in other places etc.)

exception reframe.core.exceptions.**BuildError**(*stdout*, *stderr*, *prefix=None*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a build fails.

exception reframe.core.exceptions.**BuildSystemError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a build system is not configured properly.

exception reframe.core.exceptions.**ConfigError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a configuration error occurs.

exception reframe.core.exceptions.**ContainerError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a container platform is not configured properly.

exception reframe.core.exceptions.**DependencyError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a dependency problem is encountered.

exception reframe.core.exceptions.**EnvironError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when an error related to an environment occurs.

exception reframe.core.exceptions.**FailureLimitError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when the limit of test failures has been reached.

exception reframe.core.exceptions.**ForceExitError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when ReFrame execution must be forcefully ended, e.g., after a SIGTERM was received.

exception reframe.core.exceptions.**JobBlockedError**(*msg=None*, *jobid=None*)
Bases: *reframe.core.exceptions.JobError*

Raised by job schedulers when a job is blocked indefinitely.

exception reframe.core.exceptions.**JobError**(*msg=None*, *jobid=None*)
Bases: *reframe.core.exceptions.ReframeError*

Raised for job related errors.

property jobid
The job ID of the job that encountered the error.

exception reframe.core.exceptions.**JobNotStartedError**(*msg=None*, *jobid=None*)
Bases: *reframe.core.exceptions.JobError*

Raised when trying an operation on a unstarted job.

exception reframe.core.exceptions.**JobSchedulerError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a job scheduler encounters an error condition.

exception reframe.core.exceptions.**LoggingError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when an error related to logging has occurred.

exception reframe.core.exceptions.**NameConflictError**(**args*)
Bases: *reframe.core.exceptions.RegressionTestLoadError*

Raised when there is a name clash in the test suite.

exception reframe.core.exceptions.**PerformanceError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised to denote an error in performance checking, e.g., when a performance reference is not met.

exception reframe.core.exceptions.**PipelineError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a condition prevents the regression test pipeline to continue and the error may not be described by another more specific exception.

exception reframe.core.exceptions.**ReframeBaseError**(**args*)
Bases: BaseException

Base exception for any ReFrame error.

This exception base class offers a specialized `__str__()` method that concatenates the messages of a chain of exceptions by inspecting their `__cause__` field. For example, the following piece of code will print `error message 2: error message 1`:

```
from reframe.core.exceptions import *


def foo():
    raise ReframeError('error message 1')

def bar():
    try:
        foo()
    except ReframeError as e:
        raise ReframeError('error message 2') from e

if __name__ == '__main__':
    try:
        bar()
```

(continues on next page)

(continued from previous page)

```
    except Exception as e:
        print(e)
```

exception reframe.core.exceptions.**ReframeError**(**args*)
Bases: *reframe.core.exceptions.ReframeBaseError*, Exception

Base exception for soft errors.

Soft errors may be treated by simply printing the exception's message and trying to continue execution if possible.

exception reframe.core.exceptions.**ReframeFatalError**(**args*)
Bases: *reframe.core.exceptions.ReframeBaseError*

A fatal framework error.

Execution must be aborted.

exception reframe.core.exceptions.**ReframeSyntaxError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when the syntax of regression tests is incorrect.

exception reframe.core.exceptions.**RegressionTestLoadError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when the regression test cannot be loaded.

exception reframe.core.exceptions.**SanityError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised to denote an error in sanity checking.

exception reframe.core.exceptions.**SkipTestError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a test needs to be skipped.

exception reframe.core.exceptions.**SpawnedProcessError**(*args*, *stdout*, *stderr*, *exitcode*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a spawned OS command has failed.

property command
The command that the spawned process tried to execute.

property exitcode
The exit code of the process.

property stderr
The standard error of the process as a string.

property stdout
The standard output of the process as a string.

exception reframe.core.exceptions.**SpawnedProcessTimeout**(*args*, *stdout*, *stderr*, *timeout*)
Bases: *reframe.core.exceptions.SpawnedProcessError*

Raised when a spawned OS command has timed out.

property timeout
The timeout of the process.

exception reframe.core.exceptions.**StatisticsError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised to denote an error in dealing with statistics.

exception reframe.core.exceptions.**TaskDependencyError**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised inside a regression task by the runtime when one of its dependencies has failed.

exception reframe.core.exceptions.**TaskExit**(**args*)
Bases: *reframe.core.exceptions.ReframeError*

Raised when a regression task must exit the pipeline prematurely.

reframe.core.exceptions.**is_exit_request**(*exc_type*, *exc_value*, *tb*)
Check if the error is a request to exit.

reframe.core.exceptions.**is_severe**(*exc_type*, *exc_value*, *tb*)
Check if exception is a severe one.

reframe.core.exceptions.**user_frame**(*exc_type*, *exc_value*, *tb*)
Return a user frame from the exception's traceback.

As user frame is considered the first frame that is outside from `reframe` module.

Returns A frame object or `None` if no user frame was found.

reframe.core.exceptions.**what**(*exc_type*, *exc_value*, *tb*)
A short description of the error.

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

M

N

O

P

R

S

T

U

V

W

Z