ReFrame Documentation

Release 4.1.0

CSCS

Feb 28, 2023

TABLE OF CONTENTS

ReFrame is a powerful framework for writing system regression tests and benchmarks, specifically targeted to HPC systems. The goal of the framework is to abstract away the complexity of the interactions with the system, separating the logic of a test from the low-level details, which pertain to the system configuration and setup. This allows users to write portable tests in a declarative way that describes only the test's functionality.

Tests in ReFrame are simple Python classes that specify the basic variables and parameters of the test. ReFrame offers an intuitive and very powerful syntax that allows users to create test libraries, test factories, as well as complete test workflows using other tests as fixtures. ReFrame will load the tests and send them down a well-defined pipeline that will execute them in parallel. 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.

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.

PUBLICATIONS

- Slides [pdf] @ 7th EasyBuild User Meeting 2022.
- 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.

WEBINARS

- "ReFrame – Efficient System and Application Performance Testing," CSCS Webinar, Aug. 29, 2022 [slides] [recording] [demo run].

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:

```
archspec==0.2.0
argcomplete==2.0.0
coverage==6.2; python_version < '3.7'
coverage==7.1.0; python_version >= '3.7'
importlib_metadata==4.0.1; python_version < '3.8'
jsonschema==3.2.0
lxml==4.9.2
pytest==7.0.1
pytest-forked==1.4.0
pytest-parallel==0.1.1
pytest-rerunfailures==10.3; python_version == '3.6'
pytest-rerunfailures==11.0; python_version >= '3.7'
PyYAML==6.0
requests==2.27.1; python_version == '3.6'
requests==2.28.1; python_version >= '3.7'
semver==2.13.0
setuptools==59.6.0; python_version == '3.6'
setuptools==67.1.0; python_version >= '3.7'
wcwidth==0.2.6
```

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/reframe-hpc/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

If you are new to ReFrame, the place to start is the first tutorial *Tutorial 1: Getting Started with ReFrame*, which will guide you step-by-step in both writing your first tests and in configuring ReFrame. The rest of the tutorials explore additional capabilities of the framework and cover several topics that you will likely come across when writing your own tests.

The *Configuring ReFrame for Your Site* page provides more details on how a configuration file is structured and the *Advanced Topics* explain some more advanced concepts as well as some implementation details. The *ReFrame Manuals* provide complete reference guides for the command line interface, the configuration parameters and the programming APIs for writing tests.

Finally, if you are not new to ReFrame and you have been using the 3.x versions, you should read the *What's New in ReFrame 4.0* page, which explains what are the key new features of ReFrame 4.0 as well as all the breaking changes.

2.2 What's New in ReFrame 4.0

ReFrame 4.0 introduces some important new features and removes all features, configuration options and interfaces that were deprecated in the 3.x versions. It also introduces a couple of new deprecations.

ReFrame 4.0 maintains backward compatibility as much as possible. Existing 3.x configurations and 3.x tests are expected to run out-of-the-box, despite any warnings issued. The framework's behavior with respect to performance logging has also changed, but configuration options are offered so that users can switch to the old behavior.

This page summarizes the key changes in ReFrame 4.0 and what users should pay attention to.

For a complete list of changes, please refer to the Release Notes.

2.2.1 New Features and Enchancements

Chaining Configuration Files

There is no need anymore to keep a huge configuration file with all your system and environment definitions and it is no more required to carry on the generic system configuration as well as any part of the builtin configuration. ReFrame 4.0 allows you to split your configuration in multiple files. This allows you to create minimal configuration files that contain only the necessary parts. For example, if you want to define a general configuration parameter, you don't need to copy the builtin configuration file and add it, but you simply add it in a single `general` section. This can also be very useful if you maintain a ReFrame installation used by others, as you can update your settings (systems, environments and other options) and any of your users' custom configuration will automatically inherit your settings if it is properly chained. To assist with system-wide installation the `RFM_CONFIG_PATH` environment variable is introduced that allows you to specify a path where ReFrame will look for configuration files to load.

Now that systems and environments definitions can be distributed over multiple configuration files, it can become easy to accidentally redefine a system or environment without a notice. For this reason, ReFrame warns you if a system or an environment is redefined in the same scope. Since in the past all configuration files where extended copies of the builtin configuration, you will get warnings that the `generic` system and the `builtin` environment are redefined, as ReFrame finds them in the builtin configuration, which is always loaded. You can safely ignore these warnings and use the definitions in your configuration file. If you want to eliminate them, though, you should remove the conflicting definitions from your configuration file.

Although ReFrame will not warn you for redefining other configuration sections, you are also advised to tidy up your configuration file and remove any parts that were copied unchanged from the builtin configuration.

For more information on how ReFrame 4.0 builds and loads its configuration, please refer to the documentation of the `--config-file` option, as well as the *Building the Final Configuration* section.

Performance Reporting and Logging

ReFrame 4.0 improves on how performance values are logged and reported. This is a breaking change, but you can easily revert to the old behavior.

ReFrame now logs performance *after* the test has finished and not during the performance stage. You can now log the result of the test by including `%(check_result)s` in your log handler format string. However, now, by default, ReFrame will log all the performance variables in a *single* record; in the past, a new record was logged for each performance variable. Also, the `%(check_perf_*)s` format placeholders are valid only in the `format_perfvars` configuration parameter and will be used to format the performance values if the `%(check_perfvalues)s` placeholder is present in the handler's `format` parameter. This change in behavior will likely break your log processing, especially if you are using the `graylog` or `httpjson` handlers or any handler that sends the full record to a log server. You can revert to the old behavior by setting the `perflog_compat` configuration parameter. This will send a separate record for each performance variable that will include all the individual `%(check_perf_*)s` attributes. For more information, check the documentation of the `format_perfvars` configuration parameter.

The behavior of the `filelog` is also substantially improved. The log file is printed by default in CSV format and a header is always printed at the beginning of each log file. If the log format changes or the performance variables logged by the test change, a new log file will be created with an adapted header. This way, every log file is consistent with the data in contains. For more information, please refer to the `filelog` handler *documentation*.

When you run a performance test, ReFrame will now print immediately after the test has finished a short summary of its performance. You can suppress this output by setting the log level at which this information is printed to `verbose` by setting the `perf_info_level` general configuration parameter.

Finally, the performance report printed at the end of the run using the `--performance-report` is revised providing more information in more compact form.

New Test Naming Scheme

ReFrame 4.0 makes default the new test naming scheme introduced in 3.10.0 and drops support of the old naming scheme. The new naming scheme does not affect normal tests, but it changes how parameterized tests and fixtures are named. Each test is now also associated with a unique hash code. For parameterized tests and fixtures this hash code is appended to the test's or fixture's base name when creating any test-specific directories and files, such as the test stage and output directories. The `-n` option can match a test either by its display name (the default), or by its unique internal name or by its unique hash code. Check the documentation of the `-n` for more information. For the details of the new naming scheme, please refer to the *Test Naming Scheme* section.

Note that any tests that used the old naming scheme to depend on parameterized tests will break with this change. Check the tutorial *Depending on Parameterized Tests* on how to create dependencies on parameterized tests in a portable way.

Custom parallel launchers

By relaxing the configuration schema, users can now define custom parallel launchers inside their Python configuration file. Check the tutorial *Adding a custom launcher to a partition* to find out how this can be achieved.

Unique run reports

ReFrame now generates a unique report for each run inside the `$HOME/.reframe/reports` directory. If you want to revert to the old behavior, where a single file was generated and was overwritten in every run, you should set the *report_file* configuration option or the *RFM_REPORT_FILE* environment variable.

New Backends

ReFrame 4.0 adds support for the Apptainer container platform and the Flux framework.

2.2.2 Dropped Features and Deprecations

ReFrame 4.0 drops support for all the deprecated features and behaviors of ReFrame 3.x versions. More specifically, the following deprecated features are dropped:

- The `@parameterized_test` decorator is dropped in favor of the *parameter* builtin.

- The *name* of the test is now read-only.

- The decorators `@final`, *@require_deps*, *@run_after* and *@run_before* are no more accesible via the `reframe` module. They are directly available in the *RegressionTest* namespace without the need of importing anything.

- The `@reframe.utility.sanity.sanity_function` decorator is dropped in favor of the *@deferrable* builtin.

- The `commands` attribute of the *ContainerPlatform* is dropped in favor of the *command* attribute.

- The `launcher` attribute of the *System* is dropped in favor of the `launcher_type` attribute.

- The `@required_version` decorator is dropped in favor of the `require_version` builtin. Also, automatically converting version strings that do not comply with the semantic versioning scheme is no more supported.

- The `DEPEND_EXACT`, `DEPEND_BY_ENV` and `DEPEND_FULLY` integer constants that were passed as the `how` argument of the *depends_on()* method are no more supported and a callable should be used instead. The `subdeps` argument is also dropped.

- The low-level `poll()` and `wait()` RegressionTest methods are dropped in favor of the *run_complete()* and *run_wait()*, respectively.

- The `schedulers` configuration section is dropped in favor of the partition-specific *sched_options*. Users should move any options set in the old section to the corresponding partition options.

- The `--ignore-check-conflicts` command line option and the corresponding `RFM_IGNORE_CHECK_CONFLICTS` environment variable are dropped.

- The `--force-local` and `--strict` command line options are removed. Please use instead `-S local=1` and `-S strict_check=1`, respectively.

- The `RFM_GRAYLOG_SERVER` environment variable is dropped in favor of the *RFM_GRAYLOG_ADDRESS*.

New Deprecations

- All occurrences of the `variables` name are deprecated in favor of `env_vars`. This includes the `variables` test attribute and the homonym systems, partitions and environments configuration parameters as well as the `variables` of the `Environment` base class.

- Although `perf_patterns` attribute is not deprecated, users are recommended to migrate to using the new `@performance_function` builtin. Please refer to *Writing A Performance Test* tutorial for a starting point.

2.3 ReFrame Tutorials

2.3.1 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/reframe-hpc/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
```

```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
```

```python
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'

    @sanity_function
    def assert_hello(self):
        return sn.assert_found(r'Hello, World\!', self.stdout)
```

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.

Finally, every regression test must always decorate a member function as the test's `@sanity_function`. This decorated function is converted into a lazily evaluated expression that asserts the sanity of the test. In this particular case, the specified sanity function checks that the executable has produced the desired phrase into the test's standard output *stdout*. 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:          4.0.0-dev.2+5ea6b7a6
```

```
  command:           './bin/reframe -c tutorials/basics/hello/hello1.py -r'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>'
  check search path: '/home/user/Repositories/reframe/tutorials/basics/hello/hello1.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-tgqpdq_b.log'


[==========] Running 1 check(s)
[==========] Started on Sat Nov 12 19:00:44 2022


[----------] start processing checks
[ RUN      ] HelloTest /2b3e4546 @generic:default+builtin
[       OK ] (1/1) HelloTest /2b3e4546 @generic:default+builtin
[----------] all spawned checks have finished


[  PASSED  ] Ran 1/1 test case(s) from 1 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:00:45 2022
Run report saved in '/home/user/.reframe/reports/run-report-319.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-tgqpdq_b.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": "2.0",
    "hostname": "host",
    "prefix_output": "/path/to/reframe/output",
    "prefix_stage": "/path/to/reframe/stage",
```

```
    "user": "user",
    "version": "3.10.0-dev.3+c22440c1",
    "workdir": "/path/to/reframe",
    "time_start": "2022-01-22T13:21:50+0100",
    "time_end": "2022-01-22T13:21:51+0100",
    "time_elapsed": 0.8124568462371826,
    "num_cases": 1,
    "num_failures": 0
  },
  "runs": [
    {
      "num_cases": 1,
      "num_failures": 0,
      "num_aborted": 0,
      "num_skipped": 0,
      "runid": 0,
      "testcases": [
        {
          "build_stderr": "rfm_HelloTest_build.err",
          "build_stdout": "rfm_HelloTest_build.out",
          "dependencies_actual": [],
          "dependencies_conceptual": [],
          "description": "HelloTest",
          "display_name": "HelloTest",
          "filename": "/path/to/reframe/tutorials/basics/hello/hello1.py",
          "environment": "builtin",
          "fail_phase": null,
          "fail_reason": null,
          "jobid": "43152",
          "job_stderr": "rfm_HelloTest_job.err",
          "job_stdout": "rfm_HelloTest_job.out",
          "maintainers": [],
          "name": "HelloTest",
          "nodelist": [
            "tresa.local"
          ],
          "outputdir": "/path/to/reframe/output/generic/default/builtin/HelloTest",
          "perfvars": null,
          "prefix": "/path/to/reframe/tutorials/basics/hello",
          "result": "success",
          "stagedir": "/path/to/reframe/stage/generic/default/builtin/HelloTest",
          "scheduler": "local",
          "system": "generic:default",
          "tags": [],
          "time_compile": 0.2716448307037353,
          "time_performance": 0.00010180473327636719,
          "time_run": 0.3764667510986328,
          "time_sanity": 0.0006909370422363281,
          "time_setup": 0.007919073104858398,
          "time_total": 0.800688028335713,
          "unique_name": "HelloTest"
        }
```

```
      ]
    }
  ],
  "restored_cases": []
}
```

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
```

```python
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 = ['*']

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

    @sanity_function
    def assert_hello(self):
        return sn.assert_found(r'Hello, World\!', self.stdout)
```

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.

```python
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.

```python
@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:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -c tutorials/basics/hello/hello2.py -r'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>'
  check search path: '/home/user/Repositories/reframe/tutorials/basics/hello/hello2.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-krmo7oc3.log'


[==========] Running 2 check(s)
[==========] Started on Sat Nov 12 19:00:45 2022


[----------] start processing checks
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @generic:default+builtin
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @generic:default+builtin
[     FAIL ] (1/2) HelloMultiLangTest %lang=cpp /71bf65a3 @generic:default+builtin
==> test failed during 'compile': test staged in '/home/user/Repositories/reframe/stage/
↪generic/default/builtin/HelloMultiLangTest_71bf65a3'
rfm_job.out
[       OK ] (2/2) HelloMultiLangTest %lang=c /7cfa870e @generic:default+builtin
[----------] all spawned checks have finished


[  FAILED  ] Ran 2/2 test case(s) from 2 check(s) (1 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:00:46 2022


================================================================================
SUMMARY OF FAILURES
--------------------------------------------------------------------------------
FAILURE INFO for HelloMultiLangTest_1
  * Expanded name: HelloMultiLangTest %lang=cpp
  * Description:
  * System partition: generic:default
  * Environment: builtin
  * Stage directory: /home/user/Repositories/reframe/stage/generic/default/builtin/
↪HelloMultiLangTest_71bf65a3
  * Node list:
```

```
  * Job type: local (id=None)
  * Dependencies (conceptual): []
  * Dependencies (actual): []
  * Maintainers: []
  * Failing phase: compile
  * Rerun with '-n /71bf65a3 -p builtin --system generic:default -r'
  * Reason: build system error: I do not know how to compile a C++ program
-------------------------------------------------------------------------------
Run report saved in '/home/user/.reframe/reports/run-report-320.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-krmo7oc3.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. For this we need to create our own configuration file.

```
vi tutorials/config/tresa.py
```

Here is what we need to type:

```python
# Copyright 2016-2022 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# and other ReFrame Project Developers. See the top-level LICENSE file for
# details.
#
# SPDX-License-Identifier: BSD-3-Clause


site_configuration = {
    'systems': [
        {
            'name': 'tresa',
            'descr': 'My Mac',
            'hostnames': ['tresa'],
            'modules_system': 'nomod',
            'partitions': [
                {
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['gnu', 'clang'],
```

```
                }
            ]
        }
    ],
    'environments': [
        {
            'name': 'gnu',
            'cc': 'gcc-12',
            'cxx': 'g++-12',
            'ftn': 'gfortran-12',
            'target_systems': ['tresa']
        },
        {
            'name': 'clang',
            'cc': 'clang',
            'cxx': 'clang++',
            'ftn': '',
            'target_systems': ['tresa']
        },
    ]
}
```

We define a system named `tresa` 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 12, 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. Note also that we set the `target_systems` for each environment definition. This restricts the definition of the environment being defined to the specified systems only. ReFrame will always pick the definition that is a closest match for the current system. Restricting the environment definitions is generally a good practice if you plan to define multiple systems in multiple configuration files, as ReFrame would otherwise complain that an environment is redefined. On the other hand, if you want to provide generic definitions of environments that are valid for multiple systems, you may skip that. This is what the builtin configuration of ReFrame does for its generic `builtin` environment.

Finally, the new system that we defined may be identified by the hostname `tresa` (see the `hostnames` systems 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: Multiple systems may defined in a configuration file, in which case they are tried in order and the first match is picked. This means that the systems whose `hostnames` patterns are more generic, they should go to the end of the list.

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/tresa.py -c tutorials/basics/hello/hello2.py -r
```

```
[ReFrame Setup]
  version:            4.0.0-dev.2+5ea6b7a6
  command:            './bin/reframe -C tutorials/config/tresa.py -c tutorials/basics/
→hello/hello2.py -r'
  launched by:        user@host
  working directory:  '/home/user/Repositories/reframe'
  settings files:     '<builtin>', 'tutorials/config/tresa.py'
  check search path:  '/home/user/Repositories/reframe/tutorials/basics/hello/hello2.py'
  stage directory:    '/home/user/Repositories/reframe/stage'
  output directory:   '/home/user/Repositories/reframe/output'
  log files:          '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-e3dlf19_.log'


[==========] Running 2 check(s)
[==========] Started on Sat Nov 12 19:00:46 2022


[----------] start processing checks
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @tresa:default+gnu
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @tresa:default+clang
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @tresa:default+gnu
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @tresa:default+clang
rfm_job.out
[       OK ] (1/4) HelloMultiLangTest %lang=c /7cfa870e @tresa:default+gnu
rfm_job.out
[       OK ] (2/4) HelloMultiLangTest %lang=c /7cfa870e @tresa:default+clang
rfm_job.out
[       OK ] (3/4) HelloMultiLangTest %lang=cpp /71bf65a3 @tresa:default+gnu
rfm_job.out
[       OK ] (4/4) HelloMultiLangTest %lang=cpp /71bf65a3 @tresa:default+clang
[----------] all spawned checks have finished


[  PASSED  ] Ran 4/4 test case(s) from 2 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:00:48 2022
Run report saved in '/home/user/.reframe/reports/run-report-321.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-e3dlf19_.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 could set the RFM_CONFIG_FILES environment variable, which takes a colon-separated list of configuration files that ReFrame will load. We will take advantage of multiple configuration files later in the tutorial.

```
export RFM_CONFIG_FILES=$(pwd)/tutorials/config/tresa.py
```

Tip: If our configuration file was named settings.py and we did not intend to use multiple configuration files in the same directory, we could also set the RFM_CONFIG_PATH environment variable.

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
```

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


#ifdef SYNC_MESSAGES
std::mutex hello_mutex;
#endif


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
```

```python
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']

    @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']

    @sanity_function
    def assert_hello(self):
        return 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 to be used. The options passed to the test's executable can be set through the `executable_opts` variable, which in this case is set to `'16'`.

Let's run the test now:

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

```
[ReFrame Setup]
  version:          4.0.0-dev.2+5ea6b7a6
  command:          './bin/reframe -c tutorials/basics/hellomp/hellomp1.py -r'
```

(continued from previous page)

```
  launched by:         user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:     '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
↪tresa.py'
  check search path: '/home/user/Repositories/reframe/tutorials/basics/hellomp/hellomp1.
↪py'
  stage directory:    '/home/user/Repositories/reframe/stage'
  output directory:   '/home/user/Repositories/reframe/output'
  log files:           '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-v56bz2uo.log'


[==========] Running 1 check(s)
[==========] Started on Sat Nov 12 19:00:48 2022


[----------] start processing checks
[ RUN      ] HelloThreadedTest /a6fa300f @tresa:default+gnu
[ RUN      ] HelloThreadedTest /a6fa300f @tresa:default+clang
[       OK ] (1/2) HelloThreadedTest /a6fa300f @tresa:default+gnu
[       OK ] (2/2) HelloThreadedTest /a6fa300f @tresa:default+clang
[----------] all spawned checks have finished


[  PASSED  ] Ran 2/2 test case(s) from 1 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:00:50 2022
Run report saved in '/home/user/.reframe/reports/run-report-322.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-v56bz2uo.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

So far, we have seen only a `grep`-like search for a string in the test's *stdout*, but ReFrame's `@sanity_function` are much more capable than this. In fact, one could practically do almost any operation in the output and process it as you would like before assessing the test's sanity. In the following, we extend the sanity checking of the above multithreaded "Hello, World!" to assert that all the threads produce a greetings line. See the highlighted lines below in the modified version of the `@sanity_function`.

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

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


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

    @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']

    @sanity_function
    def assert_num_messages(self):
        num_messages = sn.len(sn.findall(r'\[\s?\d+\] Hello, World\!',
                                         self.stdout))
        return sn.assert_eq(num_messages, 16)
```

This new `@sanity_function` counts all the pattern matches in the tests's *stdout* and checks that this count matches the expected value. The execution of the function `assert_num_messages()` is deferred to the `sanity` stage in the test's pipeline, after the executable has run and the *stdout* file has been populated. In this example, we have used the *findall()* utility function from the *sanity* module to conveniently extract the pattern matches. This module provides a broad range of utility functions that can be used to compose more complex sanity checks. However, note that the utility functions in this module are lazily evaluated expressions or *deferred expressions* which must be evaluated either implicitly or explicitly (see *Deferrable Functions Reference*).

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:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -c tutorials/basics/hellomp/hellomp2.py -r'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
```

(continues on next page)

```
→tresa.py'
  check search path: '/home/user/Repositories/reframe/tutorials/basics/hellomp/hellomp2.
→py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:            '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-a2tt4eqp.log'


[==========] Running 1 check(s)
[==========] Started on Sat Nov 12 19:00:50 2022


[----------] start processing checks
[ RUN      ] HelloThreadedExtendedTest /4733a67d @tresa:default+gnu
[ RUN      ] HelloThreadedExtendedTest /4733a67d @tresa:default+clang
[     FAIL ] (1/2) HelloThreadedExtendedTest /4733a67d @tresa:default+gnu
==> test failed during 'sanity': test staged in '/home/user/Repositories/reframe/stage/
→tresa/default/gnu/HelloThreadedExtendedTest'
[     FAIL ] (2/2) HelloThreadedExtendedTest /4733a67d @tresa:default+clang
==> test failed during 'sanity': test staged in '/home/user/Repositories/reframe/stage/
→tresa/default/clang/HelloThreadedExtendedTest'
[----------] all spawned checks have finished


[  FAILED  ] Ran 2/2 test case(s) from 1 check(s) (2 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:00:52 2022


================================================================================
SUMMARY OF FAILURES
--------------------------------------------------------------------------------
FAILURE INFO for HelloThreadedExtendedTest
  * Expanded name: HelloThreadedExtendedTest
  * Description:
  * System partition: tresa:default
  * Environment: gnu
  * Stage directory: /home/user/Repositories/reframe/stage/tresa/default/gnu/
→HelloThreadedExtendedTest
  * Node list: hostNone
  * Job type: local (id=59525)
  * Dependencies (conceptual): []
  * Dependencies (actual): []
  * Maintainers: []
  * Failing phase: sanity
  * Rerun with '-n /4733a67d -p gnu --system tresa:default -r'
  * Reason: sanity error: 13 != 16
--------------------------------------------------------------------------------
FAILURE INFO for HelloThreadedExtendedTest
  * Expanded name: HelloThreadedExtendedTest
  * Description:
  * System partition: tresa:default
  * Environment: clang
  * Stage directory: /home/user/Repositories/reframe/stage/tresa/default/clang/
→HelloThreadedExtendedTest
  * Node list: hostNone
  * Job type: local (id=59528)
```

```
  * Dependencies (conceptual): []
  * Dependencies (actual): []
  * Maintainers: []
  * Failing phase: sanity
  * Rerun with '-n /4733a67d -p clang --system tresa:default -r'
  * Reason: sanity error: 11 != 16
-------------------------------------------------------------------------------
Run report saved in '/home/user/.reframe/reports/run-report-323.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-a2tt4eqp.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
```

```python
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'
    executable_opts = ['16']

    @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']

    @sanity_function
    def assert_num_messages(self):
        num_messages = sn.len(sn.findall(r'\[\s?\d+\] Hello, World\!',
                                         self.stdout))
        return 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 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
```

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

```python
@rfm.simple_test
class StreamTest(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['gnu']
    prebuild_cmds = [
        'wget https://raw.githubusercontent.com/jeffhammond/STREAM/master/stream.c'  # noqa: E501
    ]
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    env_vars = {
        'OMP_NUM_THREADS': '4',
        'OMP_PLACES': 'cores'
    }

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

    @sanity_function
    def validate_solution(self):
        return sn.assert_found(r'Solution Validates', self.stdout)

    @performance_function('MB/s', perf_key='Copy')
    def extract_copy_perf(self):
        return sn.extractsingle(r'Copy:\s+(\S+)\s+.*', self.stdout, 1, float)

    @performance_function('MB/s', perf_key='Scale')
    def extract_scale_perf(self):
        return sn.extractsingle(r'Scale:\s+(\S+)\s+.*', self.stdout, 1, float)

    @performance_function('MB/s', perf_key='Add')
    def extract_add_perf(self):
        return sn.extractsingle(r'Add:\s+(\S+)\s+.*', self.stdout, 1, float)

    @performance_function('MB/s', perf_key='Triad')
    def extract_triad_perf(self):
        return 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 *env_vars* dictionary.

What makes a ReFrame test a performance test is the definition of at least one *performance function*. Similarly to a test's *@sanity_function*, a performance function is a member function decorated with the *@performance_function* decorator that merely extracts or computes a performance metric from the test's output and associates it with a unit. By default, every performance function defined in the test is assigned to a *performance variable* with the func-

ReFrame Documentation, Release 4.1.0

tion's name. A performance variable is a named quantity representing a performance metric that ReFrame will report on, log and can also check against a reference value. The performance variables of a test are stored in the `perf_variables` dictionary. The keys are the names of the metrics, whereas the values are *performance functions*. The `@performance_function` decorator apart from turning an ordinary method into a "performance function", it also creates an entry in the `perf_variables` dictionary. The optional `perf_key` argument can be used to assign a different name to the newly created performance variable.

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, where each of the performance functions use the `extractsingle()` utility 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 of the run for each performance test that has run. Additionally, as soon as a performance test finishes, the obtained performance for each of the metrics is immediately reported. This is especially useful if you run long suites of performance exploration tests and you do not want to wait until the end of the run to have an overview of the obtained performance.

```
[ReFrame Setup]
  version:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -c tutorials/basics/stream/stream1.py -r --
→performance-report'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
→tresa.py'
  check search path: '/home/user/Repositories/reframe/tutorials/basics/stream/stream1.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-v0ig7jt4.log'

[==========] Running 1 check(s)
[==========] Started on Sat Nov 12 19:00:53 2022

[----------] start processing checks
[ RUN      ] StreamTest /cdf4820d @tresa:default+gnu
[       OK ] (1/1) StreamTest /cdf4820d @tresa:default+gnu
P: Copy: 24031.8 MB/s (r:0, l:None, u:None)
P: Scale: 16297.9 MB/s (r:0, l:None, u:None)
P: Add: 17843.8 MB/s (r:0, l:None, u:None)
P: Triad: 18278.3 MB/s (r:0, l:None, u:None)
[----------] all spawned checks have finished

[  PASSED  ] Ran 1/1 test case(s) from 1 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:00:56 2022
```

```
====================================================================================
PERFORMANCE REPORT
------------------------------------------------------------------------------------
[StreamTest /cdf4820d @tresa:default:gnu]
  num_tasks: 1
  num_gpus_per_node: 0
  performance:
    - Copy: 24031.8 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 16297.9 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 17843.8 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 18278.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
------------------------------------------------------------------------------------
Run report saved in '/home/user/.reframe/reports/run-report-324.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-v0ig7jt4.log'
```

Setting explicitly the test's performance variables

Users are allowed to manipulate the test's `perf_variables` dictionary directly. This is useful to avoid code repetition or in cases that relying on decorated methods to populate the `perf_variables` is impractical, e.g., creating multiple performance variables in a loop.

You might have noticed that in our STREAM example above, all four performance functions are almost identical except for a small part of the regex pattern. In the following example, we define a single performance function, `extract_bw()`, that can extract any of the requested bandwidth metrics, and we populate the `perf_variables` ourselves in a pre-performance hook:

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

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


@rfm.simple_test
class StreamAltTest(rfm.RegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['gnu']
    prebuild_cmds = [
        'wget https://raw.githubusercontent.com/jeffhammond/STREAM/master/stream.c'  # noqa: E501
    ]
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    env_vars = {
        'OMP_NUM_THREADS': '4',
        'OMP_PLACES': 'cores'
    }

    @run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = ['-DSTREAM_ARRAY_SIZE=$((1 << 25))']
```

```python
        self.build_system.cflags = ['-fopenmp', '-O3', '-Wall']

    @sanity_function
    def validate_solution(self):
        return sn.assert_found(r'Solution Validates', self.stdout)

    @performance_function('MB/s')
    def extract_bw(self, kind='Copy'):
        '''Generic performance extraction function.'''

        if kind not in ('Copy', 'Scale', 'Add', 'Triad'):
            raise ValueError(f'illegal value in argument kind ({kind!r})')

        return sn.extractsingle(rf'{kind}:\s+(\S+)\s+.*',
                                self.stdout, 1, float)

    @run_before('performance')
    def set_perf_variables(self):
        '''Build the dictionary with all the performance variables.'''

        self.perf_variables = {
            'Copy': self.extract_bw(),
            'Scale': self.extract_bw('Scale'),
            'Add': self.extract_bw('Add'),
            'Triad': self.extract_bw('Triad'),
        }
```

As mentioned in the previous section the *@performance_function* decorator performs two tasks:

1. It converts a test method to *performance function*, i.e., a function that is suitable for extracting a performance metric.

2. It updates the *perf_variables* dictionary with the newly created performance function.

In this example, we are only interested in the first functionality and that's why we redefine completely the test's *perf_variables* using the `extract_bw()` performance function. If you are inheriting from a base test and you don't want to override completely its performance variables, you could call instead `update()` on *perf_variables*.

Finally, you can convert any arbitrary function or *deferred expression* into a performance function by calling the *make_performance_function()* utility as shown below:

```python
@run_before('performance')
def set_perf_vars(self):
    self.perf_variables = {
        'Copy': sn.make_performance_function(
            sn.extractsingle(r'Copy:\s+(\S+)\s+.*',
                             self.stdout, 1, float),
            'MB/s'
        )
    }
```

Note that in this case, the newly created performance function is not assigned to a test's performance variable and you will have to do this independently.

Adding reference values

On its current state, the above STREAM performance test will simply extract and report the performance variables regardless of the actual performance values. However, in some situations, it might be useful to check that the extracted performance values are within an expected range, and report a failure whenever a test performs below expectations. To this end, ReFrame tests include the *reference* variable, which enables setting 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/stream3.py
```

```python
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 https://raw.githubusercontent.com/jeffhammond/STREAM/master/stream.c'  # noqa: E501
    ]
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    env_vars = {
        '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')
        }
    }

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

    @sanity_function
    def validate_solution(self):
        return sn.assert_found(r'Solution Validates', self.stdout)

    @performance_function('MB/s')
    def extract_bw(self, kind='Copy'):
```

<div align="right">(continues on next page)</div>

```
        '''Generic performance extraction function.'''

        if kind not in ('Copy', 'Scale', 'Add', 'Triad'):
            raise ValueError(f'illegal value in argument kind ({kind!r})')

        return sn.extractsingle(rf'{kind}:\s+(\S+)\s+.*',
                                self.stdout, 1, float)

    @run_before('performance')
    def set_perf_variables(self):
        '''Build the dictionary with all the performance variables.'''

        self.perf_variables = {
            'Copy': self.extract_bw(),
            'Scale': self.extract_bw('Scale'),
            'Add': self.extract_bw('Add'),
            'Triad': self.extract_bw('Triad'),
        }
```

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. Also, the units in this reference variable are entirely optional, since they were already provided through the @performance_function decorator.

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/stream3.py -r --performance-report
```

```
FAILURE INFO for StreamWithRefTest
  * Expanded name: StreamWithRefTest
  * Description:
  * System partition: catalina:default
  * Environment: gnu
  * Stage directory: /Users/user/Repositories/reframe/stage/catalina/default/gnu/
→StreamWithRefTest
  * Node list: tresa.localNone
  * Job type: local (id=4576)
  * Dependencies (conceptual): []
  * Dependencies (actual): []
  * Maintainers: []
  * Failing phase: performance
  * Rerun with '-n /f925207b -p gnu --system catalina:default -r'
  * Reason: performance error: failed to meet reference: Add=19585.3, expected 18500␣
→(l=17575.0, u=19425.0)
```

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 at 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
```

```
job_completion_time,version,display_name,system,partition,environ,jobid,result,Copy_
↪value,Copy_unit,Copy_ref,Copy_lower,Copy_upper,Scale_value,Scale_unit,Scale_ref,Scale_
↪lower,Scale_upper,Add_value,Add_unit,Add_ref,Add_lower,Add_upper,Triad_value,Triad_
↪unit,Triad_ref,Triad_lower,Triad_upper
2022-10-18T21:41:25,4.0.0-dev.2+90fbd3ef,StreamWithRefTest,catalina,default,gnu,81351,
↪pass,24235.6,MB/s,25200,-0.05,0.05,16044.2,MB/s,16800,-0.05,0.05,17733.7,MB/s,18500,-0.
↪05,0.05,18232.0,MB/s,18800,-0.05,0.05
2022-10-18T21:41:31,4.0.0-dev.2+90fbd3ef,StreamWithRefTest,catalina,default,gnu,81377,
↪fail,23615.4,MB/s,25200,-0.05,0.05,16394.5,MB/s,16800,-0.05,0.05,17841.3,MB/s,18500,-0.
↪05,0.05,18284.1,MB/s,18800,-0.05,0.05
2022-10-18T21:46:06,4.0.0-dev.2+90fbd3ef,StreamWithRefTest,catalina,default,gnu,81480,
↪fail,23736.4,MB/s,25200,-0.05,0.05,16242.8,MB/s,16800,-0.05,0.05,17699.1,MB/s,18500,-0.
↪05,0.05,18077.3,MB/s,18800,-0.05,0.05
```

The format of this file is controlled by `handlers_perflog` logging configuration parameter and, by default, contains several information about the test. For each test, all of its performance variables are logged along with their unit, the obtained value, the reference and the lower and upper threshold. The default format is in CSV, so that it can be easily post-processed. For this reason, a header is also printed to help identify the different fields.

Since version 4.0, ReFrame is very cautious when generating this file: if a change is detected in the information that is being logged, ReFrame will not append to the file, but it will instead create a new one, saving the old file using the `.h<N>` suffix, where N is an integer that is increased every time a new file is being created due to such changes. Examples of changes in the logged information are when the log record format changes or a new performance metric is added, deleted or has its name changed. This behavior guarantees that each log file is consistent and it will not break existing parsers.

For more information on configuring performance logging in ReFrame as well as logging in general, you may refer to the *Logging Configuration* reference.

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 create a new configuration file for Piz Daint:

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

(continues on next page)

```
                    ]
                }
            ]
        }
    ],
    'environments': [
        {
            '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': 'nvidia',
            'modules': ['PrgEnv-nvidia'],
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        },
        {
            'name': 'builtin',
            'cc': 'cc',
            'cxx': 'CC',
            'ftn': 'ftn',
            'target_systems': ['daint']
        }
    ]  # end of environments
}
```

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 `tresa: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 NVIDIA. 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 add also the `modules` parameter. The `modules` parameter is a list of environment modules that needs to be loaded, in order to make available this compiler.

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

```
[ReFrame Setup]
  version:           4.0.0-dev.2
  command:           './bin/reframe -c tutorials/basics/ -R -n␣
↪HelloMultiLangTest|HelloThreadedExtended2Test|StreamWithRefTest --performance-report -r
↪'
  launched by:       user@host
  working directory: '/home/user/Devel/reframe'
  settings files:    '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path: (R) '/home/user/Devel/reframe/tutorials/basics'
  stage directory:   '/home/user/Devel/reframe/stage'
  output directory:  '/home/user/Devel/reframe/output'
  log files:         '/tmp/rfm-nyqs7jb9.log'

[==========] Running 4 check(s)
[==========] Started on Tue Nov 15 18:20:32 2022
```

(continues on next page)

```
[----------] start processing checks
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+builtin
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+gnu
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+intel
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+nvidia
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+cray
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:gpu+gnu
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:gpu+intel
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:gpu+nvidia
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:gpu+cray
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:mc+gnu
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:mc+intel
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:mc+nvidia
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @daint:mc+cray
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:login+builtin
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:login+gnu
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:login+intel
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:login+nvidia
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:login+cray
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:gpu+gnu
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:gpu+intel
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:gpu+nvidia
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:gpu+cray
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:mc+gnu
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:mc+intel
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:mc+nvidia
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @daint:mc+cray
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:login+builtin
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:login+gnu
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:login+intel
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:login+nvidia
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:login+cray
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:gpu+gnu
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:gpu+intel
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:gpu+nvidia
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:gpu+cray
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:mc+gnu
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:mc+intel
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:mc+nvidia
[ RUN      ] HelloThreadedExtended2Test /57223829 @daint:mc+cray
[ RUN      ] StreamWithRefTest /f925207b @daint:login+gnu
[ RUN      ] StreamWithRefTest /f925207b @daint:gpu+gnu
[ RUN      ] StreamWithRefTest /f925207b @daint:mc+gnu
[       OK ] ( 1/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+builtin
[       OK ] ( 2/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+gnu
[       OK ] ( 3/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+intel
[       OK ] ( 4/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+nvidia
[       OK ] ( 5/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:login+cray
[       OK ] ( 6/42) HelloMultiLangTest %lang=c /7cfa870e @daint:login+builtin
[       OK ] ( 7/42) HelloMultiLangTest %lang=c /7cfa870e @daint:login+gnu
[       OK ] ( 8/42) HelloMultiLangTest %lang=c /7cfa870e @daint:login+intel
```

```
[       OK ] ( 9/42) HelloMultiLangTest %lang=c /7cfa870e @daint:login+nvidia
[       OK ] (10/42) HelloMultiLangTest %lang=c /7cfa870e @daint:login+cray
[       OK ] (11/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:gpu+cray
[       OK ] (12/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:mc+nvidia
[       OK ] (13/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:mc+cray
[       OK ] (14/42) HelloMultiLangTest %lang=c /7cfa870e @daint:mc+cray
[       OK ] (15/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:gpu+nvidia
[       OK ] (16/42) HelloMultiLangTest %lang=c /7cfa870e @daint:gpu+intel
[       OK ] (17/42) HelloMultiLangTest %lang=c /7cfa870e @daint:gpu+nvidia
[       OK ] (18/42) HelloMultiLangTest %lang=c /7cfa870e @daint:mc+intel
[       OK ] (19/42) HelloThreadedExtended2Test /57223829 @daint:login+builtin
[       OK ] (20/42) HelloThreadedExtended2Test /57223829 @daint:login+gnu
[       OK ] (21/42) HelloThreadedExtended2Test /57223829 @daint:login+intel
[       OK ] (22/42) HelloMultiLangTest %lang=c /7cfa870e @daint:gpu+cray
[       OK ] (23/42) HelloMultiLangTest %lang=c /7cfa870e @daint:mc+gnu
[       OK ] (24/42) HelloThreadedExtended2Test /57223829 @daint:login+nvidia
[       OK ] (25/42) HelloThreadedExtended2Test /57223829 @daint:login+cray
[       OK ] (26/42) HelloMultiLangTest %lang=c /7cfa870e @daint:mc+nvidia
[       OK ] (27/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:gpu+gnu
[       OK ] (28/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:gpu+intel
[       OK ] (29/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:mc+gnu
[       OK ] (30/42) HelloMultiLangTest %lang=cpp /71bf65a3 @daint:mc+intel
[       OK ] (31/42) HelloMultiLangTest %lang=c /7cfa870e @daint:gpu+gnu
[       OK ] (32/42) StreamWithRefTest /f925207b @daint:login+gnu
P: Copy: 71061.6 MB/s (r:0, l:None, u:None)
P: Scale: 44201.5 MB/s (r:0, l:None, u:None)
P: Add: 48178.5 MB/s (r:0, l:None, u:None)
P: Triad: 48063.3 MB/s (r:0, l:None, u:None)
[       OK ] (33/42) HelloThreadedExtended2Test /57223829 @daint:mc+cray
[       OK ] (34/42) HelloThreadedExtended2Test /57223829 @daint:mc+intel
[       OK ] (35/42) HelloThreadedExtended2Test /57223829 @daint:mc+gnu
[       OK ] (36/42) HelloThreadedExtended2Test /57223829 @daint:mc+nvidia
[       OK ] (37/42) StreamWithRefTest /f925207b @daint:mc+gnu
P: Copy: 52660.1 MB/s (r:0, l:None, u:None)
P: Scale: 33117.6 MB/s (r:0, l:None, u:None)
P: Add: 34876.7 MB/s (r:0, l:None, u:None)
P: Triad: 35150.7 MB/s (r:0, l:None, u:None)
[       OK ] (38/42) HelloThreadedExtended2Test /57223829 @daint:gpu+intel
[       OK ] (39/42) HelloThreadedExtended2Test /57223829 @daint:gpu+cray
[       OK ] (40/42) HelloThreadedExtended2Test /57223829 @daint:gpu+nvidia
[       OK ] (41/42) HelloThreadedExtended2Test /57223829 @daint:gpu+gnu
[       OK ] (42/42) StreamWithRefTest /f925207b @daint:gpu+gnu
P: Copy: 49682.3 MB/s (r:0, l:None, u:None)
P: Scale: 34452.3 MB/s (r:0, l:None, u:None)
P: Add: 38030.7 MB/s (r:0, l:None, u:None)
P: Triad: 38379.0 MB/s (r:0, l:None, u:None)
[----------] all spawned checks have finished


[  PASSED  ] Ran 42/42 test case(s) from 4 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Tue Nov 15 18:22:48 2022


================================================================================
```

```
PERFORMANCE REPORT
------------------------------------------------------------------------------
[StreamWithRefTest /f925207b @daint:login:gnu]
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 71061.6 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 44201.5 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 48178.5 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 48063.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamWithRefTest /f925207b @daint:gpu:gnu]
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 49682.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 34452.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 38030.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 38379.0 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamWithRefTest /f925207b @daint:mc:gnu]
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 52660.1 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 33117.6 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 34876.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 35150.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
------------------------------------------------------------------------------
Run report saved in '/home/user/.reframe/reports/run-report-1.json'
Log file(s) saved in '/tmp/rfm-nyqs7jb9.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
```

```bash
#!/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/stream4.py
```

```python
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', 'nvidia']
    prebuild_cmds = [
        'wget https://raw.githubusercontent.com/jeffhammond/STREAM/master/stream.c'  #␣
→noqa: E501
    ]
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    env_vars = {
        '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')
```

```python
        }
    }

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

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

    @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, [])

    @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.env_vars = {
            'OMP_NUM_THREADS': num_threads,
            'OMP_PLACES': 'cores'
        }

    @sanity_function
    def validate_solution(self):
        return sn.assert_found(r'Solution Validates', self.stdout)

    @performance_function('MB/s')
    def extract_bw(self, kind='Copy'):
        if kind not in {'Copy', 'Scale', 'Add', 'Triad'}:
            raise ValueError(f'illegal value in argument kind ({kind!r})')

        return sn.extractsingle(rf'{kind}:\s+(\S+)\s+.*',
                                self.stdout, 1, float)

    @run_before('performance')
    def set_perf_variables(self):
        self.perf_variables = {
            'Copy': self.extract_bw(),
            'Scale': self.extract_bw('Scale'),
            'Add': self.extract_bw('Add'),
```

```
            'Triad': self.extract_bw('Triad'),
        }
```

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 *Test API Reference* 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/stream4.py -r --performance-report
```

```
[ReFrame Setup]
  version:           4.0.0-dev.2
  command:           './bin/reframe -c tutorials/basics/stream/stream4.py -r --
→performance-report'
  launched by:       user@host
  working directory: '/home/user/Devel/reframe'
  settings files:    '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path: '/home/user/Devel/reframe/tutorials/basics/stream/stream4.py'
  stage directory:   '/home/user/Devel/reframe/stage'
  output directory:  '/home/user/Devel/reframe/output'
  log files:         '/tmp/rfm-yf6xjn_4.log'


[==========] Running 1 check(s)
[==========] Started on Tue Nov 15 18:22:48 2022

[----------] start processing checks
```

```
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:login+gnu
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:login+intel
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:login+nvidia
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:login+cray
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:gpu+gnu
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:gpu+intel
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:gpu+nvidia
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:gpu+cray
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:mc+gnu
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:mc+intel
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:mc+nvidia
[ RUN      ] StreamMultiSysTest /eec1c676 @daint:mc+cray
[       OK ] ( 1/12) StreamMultiSysTest /eec1c676 @daint:login+gnu
P: Copy: 97772.6 MB/s (r:0, l:None, u:None)
P: Scale: 69418.6 MB/s (r:0, l:None, u:None)
P: Add: 71941.0 MB/s (r:0, l:None, u:None)
P: Triad: 73679.7 MB/s (r:0, l:None, u:None)
[       OK ] ( 2/12) StreamMultiSysTest /eec1c676 @daint:login+intel
P: Copy: 85123.0 MB/s (r:0, l:None, u:None)
P: Scale: 79701.7 MB/s (r:0, l:None, u:None)
P: Add: 81632.7 MB/s (r:0, l:None, u:None)
P: Triad: 44391.5 MB/s (r:0, l:None, u:None)
[       OK ] ( 3/12) StreamMultiSysTest /eec1c676 @daint:login+nvidia
P: Copy: 76641.4 MB/s (r:0, l:None, u:None)
P: Scale: 59041.9 MB/s (r:0, l:None, u:None)
P: Add: 64792.5 MB/s (r:0, l:None, u:None)
P: Triad: 69441.4 MB/s (r:0, l:None, u:None)
[       OK ] ( 4/12) StreamMultiSysTest /eec1c676 @daint:login+cray
P: Copy: 35658.5 MB/s (r:0, l:None, u:None)
P: Scale: 27732.2 MB/s (r:0, l:None, u:None)
P: Add: 39037.7 MB/s (r:0, l:None, u:None)
P: Triad: 45310.3 MB/s (r:0, l:None, u:None)
[       OK ] ( 5/12) StreamMultiSysTest /eec1c676 @daint:gpu+gnu
P: Copy: 42666.3 MB/s (r:0, l:None, u:None)
P: Scale: 38491.0 MB/s (r:0, l:None, u:None)
P: Add: 43686.4 MB/s (r:0, l:None, u:None)
P: Triad: 43466.6 MB/s (r:0, l:None, u:None)
[       OK ] ( 6/12) StreamMultiSysTest /eec1c676 @daint:gpu+intel
P: Copy: 51726.7 MB/s (r:0, l:None, u:None)
P: Scale: 54185.6 MB/s (r:0, l:None, u:None)
P: Add: 57608.3 MB/s (r:0, l:None, u:None)
P: Triad: 57390.7 MB/s (r:0, l:None, u:None)
[       OK ] ( 7/12) StreamMultiSysTest /eec1c676 @daint:gpu+nvidia
P: Copy: 51810.8 MB/s (r:0, l:None, u:None)
P: Scale: 39653.4 MB/s (r:0, l:None, u:None)
P: Add: 44008.0 MB/s (r:0, l:None, u:None)
P: Triad: 44384.4 MB/s (r:0, l:None, u:None)
[       OK ] ( 8/12) StreamMultiSysTest /eec1c676 @daint:gpu+cray
P: Copy: 51101.8 MB/s (r:0, l:None, u:None)
P: Scale: 38568.1 MB/s (r:0, l:None, u:None)
P: Add: 43193.6 MB/s (r:0, l:None, u:None)
P: Triad: 43142.9 MB/s (r:0, l:None, u:None)
```

```
[       OK ] ( 9/12) StreamMultiSysTest /eec1c676 @daint:mc+gnu
P: Copy: 48292.9 MB/s (r:0, l:None, u:None)
P: Scale: 38499.5 MB/s (r:0, l:None, u:None)
P: Add: 43555.7 MB/s (r:0, l:None, u:None)
P: Triad: 43871.4 MB/s (r:0, l:None, u:None)
[       OK ] (10/12) StreamMultiSysTest /eec1c676 @daint:mc+cray
P: Copy: 46538.3 MB/s (r:0, l:None, u:None)
P: Scale: 40133.3 MB/s (r:0, l:None, u:None)
P: Add: 43363.9 MB/s (r:0, l:None, u:None)
P: Triad: 43450.3 MB/s (r:0, l:None, u:None)
[       OK ] (11/12) StreamMultiSysTest /eec1c676 @daint:mc+nvidia
P: Copy: 46648.2 MB/s (r:0, l:None, u:None)
P: Scale: 40384.5 MB/s (r:0, l:None, u:None)
P: Add: 44001.1 MB/s (r:0, l:None, u:None)
P: Triad: 44489.7 MB/s (r:0, l:None, u:None)
[       OK ] (12/12) StreamMultiSysTest /eec1c676 @daint:mc+intel
P: Copy: 51335.9 MB/s (r:0, l:None, u:None)
P: Scale: 49490.3 MB/s (r:0, l:None, u:None)
P: Add: 56859.9 MB/s (r:0, l:None, u:None)
P: Triad: 56544.5 MB/s (r:0, l:None, u:None)
[----------] all spawned checks have finished

[  PASSED  ] Ran 12/12 test case(s) from 1 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Tue Nov 15 18:24:00 2022


==============================================================================
PERFORMANCE REPORT
------------------------------------------------------------------------------
[StreamMultiSysTest /eec1c676 @daint:login:gnu]
  num_cpus_per_task: 10
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 97772.6 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 69418.6 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 71941.0 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 73679.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:login:intel]
  num_cpus_per_task: 10
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 85123.0 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 79701.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 81632.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 44391.5 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:login:nvidia]
  num_cpus_per_task: 10
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 76641.4 MB/s (r: 0 MB/s l: -inf% u: +inf%)
```

```
    - Scale: 59041.9 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 64792.5 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 69441.4 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:login:cray]
  num_cpus_per_task: 10
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 35658.5 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 27732.2 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 39037.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 45310.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:gpu:gnu]
  num_cpus_per_task: 12
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 42666.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 38491.0 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 43686.4 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 43466.6 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:gpu:intel]
  num_cpus_per_task: 12
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 51726.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 54185.6 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 57608.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 57390.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:gpu:nvidia]
  num_cpus_per_task: 12
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 51810.8 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 39653.4 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 44008.0 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 44384.4 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:gpu:cray]
  num_cpus_per_task: 12
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 51101.8 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 38568.1 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 43193.6 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 43142.9 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:mc:gnu]
  num_cpus_per_task: 36
  num_gpus_per_node: 0
  num_tasks: 1
```

```
  performance:
    - Copy: 48292.9 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 38499.5 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 43555.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 43871.4 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:mc:intel]
  num_cpus_per_task: 36
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 51335.9 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 49490.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 56859.9 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 56544.5 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:mc:nvidia]
  num_cpus_per_task: 36
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 46648.2 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 40384.5 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 44001.1 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 44489.7 MB/s (r: 0 MB/s l: -inf% u: +inf%)
[StreamMultiSysTest /eec1c676 @daint:mc:cray]
  num_cpus_per_task: 36
  num_gpus_per_node: 0
  num_tasks: 1
  performance:
    - Copy: 46538.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Scale: 40133.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Add: 43363.9 MB/s (r: 0 MB/s l: -inf% u: +inf%)
    - Triad: 43450.3 MB/s (r: 0 MB/s l: -inf% u: +inf%)
--------------------------------------------------------------------------------
Run report saved in '/home/user/.reframe/reports/run-report-2.json'
Log file(s) saved in '/tmp/rfm-yf6xjn_4.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!

2.3.2 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 either the configuration files presented in *Tutorial 1: Getting Started with ReFrame*, which you can find in `tutorials/config/tresa.py` and `tutorials/config/daint.py` or build on top of them. We also assume that the reader is already familiar with the concepts presented in the basic tutorial. Finally, to avoid specifying the tutorial configuration every time, make sure to export it here:

```
export RFM_CONFIG_FILES=$(pwd)/tutorials/config/tresa.py:$(pwd)/tutorials/config/daint.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
```

```python
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', 'nvidia']
    prebuild_cmds = [
        'wget https://raw.githubusercontent.com/jeffhammond/STREAM/master/stream.c'  # ␣
→noqa: E501
    ]
    build_system = 'SingleSource'
    sourcepath = 'stream.c'
    env_vars = {
        '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'],
        'nvidia':  ['-mp', '-O3']
    })

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

(continues on next page)

```python
    })

    @run_after('init')
    def setup_build(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})'
        )

    @run_before('compile')
    def set_compiler_flags(self):
        self.build_system.cppflags = [f'-DSTREAM_ARRAY_SIZE={self.array_size}',
                                      f'-DNTIMES={self.ntimes}']
        environ = self.current_environ.name
        self.build_system.cflags = self.flags.get(environ, [])

    @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.env_vars = {
            'OMP_NUM_THREADS': num_threads,
            'OMP_PLACES': 'cores'
        }

    @sanity_function
    def validate_solution(self):
        return sn.assert_found(r'Solution Validates', self.stdout)

    @performance_function('MB/s', perf_key='Triad')
    def extract_triad_bw(self):
        return 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:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -c tutorials/advanced/parameterized/stream.py -l'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
```

```
  settings files:     '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
→tresa.py'
  check search path: '/home/user/Repositories/reframe/tutorials/advanced/parameterized/
→stream.py'
  stage directory:    '/home/user/Repositories/reframe/stage'
  output directory:   '/home/user/Repositories/reframe/output'
  log files:          '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-lou71c2g.log'

[List of matched checks]
- StreamMultiSysTest %num_bytes=536870912 /cf10843f
- StreamMultiSysTest %num_bytes=268435456 /97fb363f
- StreamMultiSysTest %num_bytes=134217728 /7b4d01d3
- StreamMultiSysTest %num_bytes=67108864 /530b0154
- StreamMultiSysTest %num_bytes=33554432 /7199fc93
- StreamMultiSysTest %num_bytes=16777216 /9d1b9ea8
- StreamMultiSysTest %num_bytes=8388608 /3f29039f
- StreamMultiSysTest %num_bytes=4194304 /e30054cd
- StreamMultiSysTest %num_bytes=2097152 /45efaec5
- StreamMultiSysTest %num_bytes=1048576 /92327981
- StreamMultiSysTest %num_bytes=524288 /eb104cd0
Found 11 check(s)

Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-lou71c2g.log'
```

ReFrame generates 11 tests from the single parameterized test. When listing parameterized tests, ReFrame adds the list of parameters after the base test name using the notation %<param>=<value>. Each generated test gets also a unique name. For more details on how the test names are generated for various types of tests, please refer to *Test Naming Scheme*.

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):

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


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

    @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
```

```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;
```

```cpp
    }

    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());
    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
```

```python
@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'

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

    @sanity_function
```

```
    def validate_test(self):
        return 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
```

```bash
#!/bin/bash

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

trap _onerror ERR

make -j 1 CC="cc" CXX="CC" FC="ftn" NVCC="nvcc" 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
```

```python
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}))'
    ]

    @sanity_function
    def assert_solution(self):
        return sn.assert_bounded(
            sn.extractsingle(
                r'Random: (?P<number>\S+)', self.stdout, 'number', float
            ),
            self.lower, self.upper
        )
```

There is nothing special for this test compared to those presented so far except that it derives from the *RunOnlyRegressionTest* class. Note that setting the *executable* in this type of test is always required. 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
```

```python
@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'

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

(continues on next page)

```
    @sanity_function
    def validate_compilation(self):
        return 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:

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


class ElemTypeParam(rfm.RegressionMixin):
    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'

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

    @sanity_function
    def validate_test(self):
        return 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']
```

```python
    build_system = 'Make'

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

    @sanity_function
    def validate_build(self):
        return 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:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -c tutorials/advanced/makefiles/maketest_mixin.py -l'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
→tresa.py'
  check search path: '/home/user/Repositories/reframe/tutorials/advanced/makefiles/
→maketest_mixin.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-z_z51hkz.log'

[List of matched checks]
- MakeOnlyTestAlt %elem_type=double /8b62380e
- MakeOnlyTestAlt %elem_type=float /da39ec20
- MakefileTestAlt %elem_type=double /89aac4a2
- MakefileTestAlt %elem_type=float /a998ce67
Found 4 check(s)

Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-z_z51hkz.log'
```

Applying a Sanity Function Iteratively

It is often the case that a common sanity function 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
```

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

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

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

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'

    @sanity_function
    def validate_test(self):
        numbers = sn.extractall(
            r'Random: (?P<number>\S+)', self.stdout, 'number', float
        )
        return sn.all([
            sn.assert_eq(sn.count(numbers), 100),
            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 returned by the test's `@sanity_function`. 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 Deferrable 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
```

```python
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'

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

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

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

```bash
#!/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
```

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

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

    @sanity_function
    def validate_test(self):
        return sn.assert_found(
            r'(exceeded memory limit)|(Out Of Memory)', self.stderr
        )
```

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
```

```bash
#!/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 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 of both of our `daint` partitions:

```
            {
                'name': 'gpu',
                'descr': 'Hybrid nodes',
                'scheduler': 'slurm',
                'launcher': 'srun',
                'access': ['-C gpu', '-A csstaff'],
                'environs': ['gnu', 'intel', 'nvidia', '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
```

```python
@rfm.simple_test
class MemoryLimitWithResourcesTest(rfm.RegressionTest):
```

(continues on next page)

```python
    valid_systems = ['daint:gpu', 'daint:mc']
    valid_prog_environs = ['gnu']
    sourcepath = 'eatmemory.c'
    executable_opts = ['2000M']
    extra_resources = {
        'memory': {'size': '1000'}
    }

    @sanity_function
    def validate_test(self):
        return 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
```

```python
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'

    @run_before('compile')
```

```python
    def set_build_system_options(self):
        self.build_system.options = ['OPENMP=1']


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


    @sanity_function
    def validate_test(self):
        return 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:

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


class DebuggerTest(rfm.RunOnlyRegressionTest):
    ...

    @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:

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


class CustomLauncherTest(rfm.RunOnlyRegressionTest):
    ...
```

```
    executable = 'custom_scheduler'
    executable_opts = [...]

    @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.

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 hard-code 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
```

```python
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

    @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)
        ]

    @sanity_function
    def validate_test(self):
        return 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
```

```bash
#!/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
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.

Adding a custom launcher to a partition

New in version 4.0.0.

An alternative to the approaches above would be to define your own custom parallel launcher and register it with the framework. You could then use it as the launcher of a system partition in the configuration and use it in multiple tests.

Each launcher needs to implement the *command()* method and can optionally change the default *run_command()* method.

As an example of how easy it is to define a new parallel launcher backend, here is the actual implementation of the `mpirun` launcher:

```python
from reframe.core.backends import register_launcher
from reframe.core.launchers import JobLauncher


@register_launcher('mpirun')
class MpirunLauncher(JobLauncher):
    def command(self, job):
        return ['mpirun', '-np', str(job.num_tasks)]
```

The *command()* returns a list of command tokens that will be combined with any user-supplied options by the *run_command()* method to generate the actual launcher command line. Notice you can use the `job` argument to get job-specific information that will allow you to construct the correct launcher invocation.

If you use a Python-based configuration file, you can define your custom launcher directly inside your config as follows:

```python
from reframe.core.backends import register_launcher
from reframe.core.launchers import JobLaucher


@register_launcher('slrun')
class MySmartLauncher(JobLauncher):
```

```python
    def command(self, job):
        return ['slrun', ...]

site_configuration = {
    'systems': [
        {
            'name': 'my_system',
            'partitions': [
                {
                    'name': 'my_partition',
                    'launcher': 'slrun'
                    ...
                }
            ],
            ...
        },
        ...
    ],
    ...
}
```

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
```

```python
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

    @sanity_function
    def validate_test(self):
        return sn.assert_eq(
```

```
            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 as default, which is a requirement for flexible tests. However, with flexible tests, this value is updated right after the job completes to the actual number of tasks that were used. Consequently, this allows the sanity function of the test to assert that the number host names printed matches *num_tasks*.

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', 'nvidia', 'cray'],
        'max_jobs': 100,
        'container_platforms': [
            {
                'type': 'Sarus',
                'modules': ['sarus']
            },
            {
                'type': 'Singularity',
                'modules': ['singularity']
            }
        ]
    },
```

For each partition, users can define a list of all supported container platforms 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. Optionally, users are allowed to set the `default` attribute to `True` in order to mark a specific container platform as the default of that partition (see below on how this information is being used). If no default container platform is specified explicitly, then always the first in the list will be considered as successful.

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

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

```python
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']

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

    @sanity_function
    def assert_release(self):
        os_release_pattern = r'18.04.\d+ LTS \(Bionic Beaver\)'
        return sn.assert_found(os_release_pattern, '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. It is not necessary to specify this attribute, in which case, the default container platform of the current partition will be used. You can still differentiate your test based on the actual container platform that is being used by checking the self.container_platform.name variable.

As soon as the container platform to be used is determined, you need to specify the container image to use by setting the *image*. If the image is not specified, then the container logic is skipped and the test executes as if the *container_platform* was never set.

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 if the containerized code path is taken, i.e., when *image* is not None.

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:

```
    @sanity_function
    def assert_release(self):
        os_release_pattern = r'18.04.\d+ LTS \(Bionic Beaver\)'
        return 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 *Test API Reference* guide. On how to configure ReFrame for running containerized tests, please have a look at the *Container Platform Configuration* section of the *Configuration Reference*.

Changed in version 3.12.0: There is no need any more to explicitly set the `container_platform` in the test. This is automatically initialized from the default platform of the current partition.

Combining containerized and native application tests

New in version 3.12.0.

It is very easy in ReFrame to have a single run-only test to either test the native or the containerized version of an application. This is possible, since the framework will only take the "containerized" code path only if the *image* attribute of the `container_platform` is not `None`. Otherwise, the *bare metal* version of the tested application will be run. The following test uses exactly this trick to test a series of GROMACS images as well as the native one provided on the Piz Daint supercomputer. It also extends the GROMACS benchmark tests that are provided with ReFrame's test library (see *ReFrame Test Library (experimental)*). For simplicity, we are assuming a single system here (the hybrid partition of Piz Daint) and we set fixed values for the `num_cpus_per_task` as well as the `-ntomp` option of GROMACS (NB: in a real-world test we would use the auto-detected processor topology information to set these values; see *Auto-detecting processor information* for more information). We also redefine and restrict the benchmark's parameters `benchmark_info` and `nb_impl` to the values that are of interest for the demonstration of this test. Finally, we also reset the executable to use `gmx` instead of the `gmx_mpi` that is used from the library test.

```python
import reframe as rfm
from hpctestlib.sciapps.gromacs.benchmarks import gromacs_check


def _hecbiosim_bench(params):
    for p in params:
        if p[0] == 'HECBioSim/hEGFRDimerSmallerPL':
            return [p]


@rfm.simple_test
class gromacs_containerized_test(gromacs_check):
    # Restrict library test parameters to only those relevant for this example
    benchmark_info = parameter(inherit_params=True,
                               filter_params=_hecbiosim_bench,
                               fmt=lambda x: x[0])
    nb_impl = parameter(['gpu'])

    # New parameter for testing the various images
    gromacs_image = parameter([
        None,
        'nvcr.io/hpc/gromacs:2020',
        'nvcr.io/hpc/gromacs:2020.2',
        'nvcr.io/hpc/gromacs:2021',
        'nvcr.io/hpc/gromacs:2021.3',
        'nvcr.io/hpc/gromacs:2022.1'
    ])
    valid_systems = ['daint:gpu']
    valid_prog_environs = ['gnu']
    use_multithreading = False
    executable = 'gmx mdrun'
    executable_opts += ['-dlb yes', '-ntomp 12', '-npme -1', '-v']
    num_tasks = 1
    num_tasks_per_node = 1
    num_cpus_per_task = 12

    @run_after('init')
    def setup_container_run(self):
        exec_cmd = ' '.join([self.executable, *self.executable_opts])
        self.container_platform.image = self.gromacs_image
        self.container_platform.command = exec_cmd
        if self.gromacs_image is None:
            self.modules = ['daint-gpu', 'GROMACS']
```

All this test does in addition to the library test it inherits from is to set the *image* and the *command* attributes of the *container_platform*. The former is set from the `gromacs_image` test parameter whereas the latter from the test's *executable* and *executable_opts* attributes. Remember that these attributes are ignored if the framework takes the path of launching a container. Finally, if the image is `None` we handle the case of the native run, in which case we load the modules required to run GROMACS natively on the target system.

In the following, we run the GPU version of a single benchmark with a series of images from NVIDIA and natively:

```
$ ./bin/reframe -C tutorials/config/daint.py -c tutorials/advanced/containers/gromacs_
↪test.py -r
```

```
[==========] Running 6 check(s)
[==========] Started on Fri Jun 17 16:20:16 2022

[----------] start processing checks
[ RUN      ] gromacs_containerized_test %benchmark_info=HECBioSim/hEGFRDimerSmallerPL
↪%nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2022.1 @daint:gpu+gnu
[ RUN      ] gromacs_containerized_test %benchmark_info=HECBioSim/hEGFRDimerSmallerPL
↪%nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2021.3 @daint:gpu+gnu
[ RUN      ] gromacs_containerized_test %benchmark_info=HECBioSim/hEGFRDimerSmallerPL
↪%nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2021 @daint:gpu+gnu
[ RUN      ] gromacs_containerized_test %benchmark_info=HECBioSim/hEGFRDimerSmallerPL
↪%nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2020.2 @daint:gpu+gnu
[ RUN      ] gromacs_containerized_test %benchmark_info=HECBioSim/hEGFRDimerSmallerPL
↪%nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2020 @daint:gpu+gnu
[ RUN      ] gromacs_containerized_test %benchmark_info=HECBioSim/hEGFRDimerSmallerPL
↪%nb_impl=gpu %gromacs_image=None @daint:gpu+gnu
[       OK ] (1/6) gromacs_containerized_test %benchmark_info=HECBioSim/
↪hEGFRDimerSmallerPL %nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2020.2␣
↪@daint:gpu+gnu
[       OK ] (2/6) gromacs_containerized_test %benchmark_info=HECBioSim/
↪hEGFRDimerSmallerPL %nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2020 @daint:gpu+gnu
[       OK ] (3/6) gromacs_containerized_test %benchmark_info=HECBioSim/
↪hEGFRDimerSmallerPL %nb_impl=gpu %gromacs_image=None @daint:gpu+gnu
[       OK ] (4/6) gromacs_containerized_test %benchmark_info=HECBioSim/
↪hEGFRDimerSmallerPL %nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2022.1␣
↪@daint:gpu+gnu
[       OK ] (5/6) gromacs_containerized_test %benchmark_info=HECBioSim/
↪hEGFRDimerSmallerPL %nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2021 @daint:gpu+gnu
[       OK ] (6/6) gromacs_containerized_test %benchmark_info=HECBioSim/
↪hEGFRDimerSmallerPL %nb_impl=gpu %gromacs_image=nvcr.io/hpc/gromacs:2021.3␣
↪@daint:gpu+gnu
[----------] all spawned checks have finished

[  PASSED  ] Ran 6/6 test case(s) from 6 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Fri Jun 17 16:23:47 2022
```

We can also inspect the generated job scripts for the native and a containerized run:

```
cat output/daint/gpu/gnu/gromacs_containerized_test_0/rfm_gromacs_containerized_test_0_
↪job.sh
```

```bash
#!/bin/bash
#SBATCH --job-name="rfm_gromacs_containerized_test_0_job"
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=12
#SBATCH --output=rfm_gromacs_containerized_test_0_job.out
#SBATCH --error=rfm_gromacs_containerized_test_0_job.err
#SBATCH -A csstaff
#SBATCH --constraint=gpu
#SBATCH --hint=nomultithread
module unload PrgEnv-cray
module load PrgEnv-gnu
```

(continues on next page)

```
module load daint-gpu
module load GROMACS
curl -LJO https://github.com/victorusu/GROMACS_Benchmark_Suite/raw/1.0.0/HECBioSim/
↪hEGFRDimerSmallerPL/benchmark.tpr
srun gmx mdrun -dlb yes -ntomp 12 -npme -1 -v -nb gpu -s benchmark.tpr
```

And the containerized run:

```
cat output/daint/gpu/gnu/gromacs_containerized_test_1/rfm_gromacs_containerized_test_1_
↪job.sh
```

```
#!/bin/bash
#SBATCH --job-name="rfm_gromacs_containerized_test_1_job"
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=12
#SBATCH --output=rfm_gromacs_containerized_test_1_job.out
#SBATCH --error=rfm_gromacs_containerized_test_1_job.err
#SBATCH -A csstaff
#SBATCH --constraint=gpu
#SBATCH --hint=nomultithread
module unload PrgEnv-cray
module load PrgEnv-gnu
module load sarus
curl -LJO https://github.com/victorusu/GROMACS_Benchmark_Suite/raw/1.0.0/HECBioSim/
↪hEGFRDimerSmallerPL/benchmark.tpr
sarus pull nvcr.io/hpc/gromacs:2020
srun sarus run --mount=type=bind,source="/users/user/Devel/reframe/stage/daint/gpu/gnu/
↪gromacs_containerized_test_43",destination="/rfm_workdir" -w /rfm_workdir nvcr.io/hpc/
↪gromacs:2020 gmx mdrun -dlb yes -ntomp 12 -npme -1 -v -nb gpu -s benchmark.tpr
```

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
```

```python
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',
    })

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

    @run_before('run')
    def set_container_platform(self):
        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 \({name}\)'

    @sanity_function
    def assert_release(self):
        return 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 ahead with 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
```

```bash
#!/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
```

```python
import tutorials.advanced.library.lib as lib


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

    @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!

2.3.3 Tutorial 3: Using Dependencies in ReFrame Tests

New in version 2.21.

A ReFrame test may define dependencies to other tests. An example scenario is to test different runtime configurations of a benchmark that you need to compile, or run a scaling analysis of a code. In such cases, you don't want to 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'
    ]

    @sanity_function
    def validate_download(self):
        return sn.assert_true(os.path.exists('osu-micro-benchmarks-5.6.2'))
```

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']
    valid_prog_environs = ['gnu', 'nvidia', 'intel']
    build_system = 'Autotools'

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

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

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

    @sanity_function
    def validate_build(self):
        return sn.assert_not_found('error', self.stderr)
```

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

```
    @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:

```python
@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:

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

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

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

    @sanity_function
    def validate_test(self):
        return sn.assert_found(r'^8', self.stdout)


@rfm.simple_test
```

(continues on next page)

```python
class OSULatencyTest(OSUBenchmarkTestBase):
    descr = 'OSU latency test'

    @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']

    @performance_function('us')
    def latency(self):
        return 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:

```python
    @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:

```python
    @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.

```python
@rfm.simple_test
class OSUAllreduceTest(OSUBenchmarkTestBase):
    mpi_tasks = parameter(1 << i for i in range(1, 5))
    descr = 'OSU Allreduce test'

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

    @require_deps
```

```python
    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']

    @performance_function('us')
    def latency(self):
        return sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
```

The full set of OSU example tests is shown below:

```python
# Copyright 2016-2022 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


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

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

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

    @sanity_function
    def validate_test(self):
        return sn.assert_found(r'^8', self.stdout)


@rfm.simple_test
class OSULatencyTest(OSUBenchmarkTestBase):
    descr = 'OSU latency test'

    @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']

    @performance_function('us')
    def latency(self):
        return sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
# rfmdocend: osupingpong


@rfm.simple_test
class OSUBandwidthTest(OSUBenchmarkTestBase):
    descr = 'OSU bandwidth test'

    @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']

    @performance_function('MB/s')
    def bandwidth(self):
        return 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'

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

    @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']

    @performance_function('us')
    def latency(self):
        return 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', 'nvidia', 'intel']
    build_system = 'Autotools'

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

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

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

    @sanity_function
    def validate_build(self):
        return 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'
    ]

    @sanity_function
    def validate_download(self):
        return sn.assert_true(os.path.exists('osu-micro-benchmarks-5.6.2'))
```

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:            4.0.0-dev.2
```

```
  command:             './bin/reframe -c tutorials/deps/osu_benchmarks.py -r'
  launched by:         user@host
  working directory:   '/home/user/Devel/reframe'
  settings files:      '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path:   '/home/user/Devel/reframe/tutorials/deps/osu_benchmarks.py'
  stage directory:     '/home/user/Devel/reframe/stage'
  output directory:    '/home/user/Devel/reframe/output'
  log files:           '/tmp/rfm-r1a7v0w3.log'


[==========] Running 8 check(s)
[==========] Started on Tue Nov 15 18:24:00 2022


[----------] start processing checks
[ RUN      ] OSUDownloadTest /7de668df @daint:login+builtin
[       OK ] ( 1/22) OSUDownloadTest /7de668df @daint:login+builtin
[ RUN      ] OSUBuildTest /19b4fb56 @daint:gpu+gnu
[ RUN      ] OSUBuildTest /19b4fb56 @daint:gpu+intel
[ RUN      ] OSUBuildTest /19b4fb56 @daint:gpu+nvidia
[       OK ] ( 2/22) OSUBuildTest /19b4fb56 @daint:gpu+gnu
[ RUN      ] OSUAllreduceTest %mpi_tasks=16 /7f033d39 @daint:gpu+gnu
[ RUN      ] OSUAllreduceTest %mpi_tasks=8 /005fca19 @daint:gpu+gnu
[ RUN      ] OSUAllreduceTest %mpi_tasks=4 /84b85d90 @daint:gpu+gnu
[ RUN      ] OSUAllreduceTest %mpi_tasks=2 /9d550c4f @daint:gpu+gnu
[ RUN      ] OSUBandwidthTest /764cdb0b @daint:gpu+gnu
[ RUN      ] OSULatencyTest /14f35a43 @daint:gpu+gnu
[       OK ] ( 3/22) OSUBuildTest /19b4fb56 @daint:gpu+intel
[ RUN      ] OSUAllreduceTest %mpi_tasks=16 /7f033d39 @daint:gpu+intel
[ RUN      ] OSUAllreduceTest %mpi_tasks=8 /005fca19 @daint:gpu+intel
[ RUN      ] OSUAllreduceTest %mpi_tasks=4 /84b85d90 @daint:gpu+intel
[       OK ] ( 4/22) OSUBuildTest /19b4fb56 @daint:gpu+nvidia
[ RUN      ] OSUAllreduceTest %mpi_tasks=16 /7f033d39 @daint:gpu+nvidia
[ RUN      ] OSUAllreduceTest %mpi_tasks=8 /005fca19 @daint:gpu+nvidia
[ RUN      ] OSUAllreduceTest %mpi_tasks=4 /84b85d90 @daint:gpu+nvidia
[ RUN      ] OSUAllreduceTest %mpi_tasks=2 /9d550c4f @daint:gpu+intel
[ RUN      ] OSUAllreduceTest %mpi_tasks=2 /9d550c4f @daint:gpu+nvidia
[ RUN      ] OSUBandwidthTest /764cdb0b @daint:gpu+intel
[ RUN      ] OSUBandwidthTest /764cdb0b @daint:gpu+nvidia
[ RUN      ] OSULatencyTest /14f35a43 @daint:gpu+intel
[ RUN      ] OSULatencyTest /14f35a43 @daint:gpu+nvidia
[       OK ] ( 5/22) OSUAllreduceTest %mpi_tasks=4 /84b85d90 @daint:gpu+gnu
P: latency: 5.31 us (r:0, l:None, u:None)
[       OK ] ( 6/22) OSUAllreduceTest %mpi_tasks=2 /9d550c4f @daint:gpu+intel
P: latency: 10.07 us (r:0, l:None, u:None)
[       OK ] ( 7/22) OSUAllreduceTest %mpi_tasks=2 /9d550c4f @daint:gpu+gnu
P: latency: 1.67 us (r:0, l:None, u:None)
[       OK ] ( 8/22) OSUAllreduceTest %mpi_tasks=4 /84b85d90 @daint:gpu+intel
P: latency: 24.97 us (r:0, l:None, u:None)
[       OK ] ( 9/22) OSUAllreduceTest %mpi_tasks=2 /9d550c4f @daint:gpu+nvidia
P: latency: 8.92 us (r:0, l:None, u:None)
[       OK ] (10/22) OSUAllreduceTest %mpi_tasks=16 /7f033d39 @daint:gpu+intel
P: latency: 14.78 us (r:0, l:None, u:None)
[       OK ] (11/22) OSULatencyTest /14f35a43 @daint:gpu+nvidia
```

```
P: latency: 2.19 us (r:0, l:None, u:None)
[       OK ] (12/22) OSULatencyTest /14f35a43 @daint:gpu+gnu
P: latency: 1.76 us (r:0, l:None, u:None)
[       OK ] (13/22) OSUAllreduceTest %mpi_tasks=16 /7f033d39 @daint:gpu+gnu
P: latency: 19.54 us (r:0, l:None, u:None)
[       OK ] (14/22) OSULatencyTest /14f35a43 @daint:gpu+intel
P: latency: 4.4 us (r:0, l:None, u:None)
[       OK ] (15/22) OSUAllreduceTest %mpi_tasks=4 /84b85d90 @daint:gpu+nvidia
P: latency: 6.88 us (r:0, l:None, u:None)
[       OK ] (16/22) OSUAllreduceTest %mpi_tasks=8 /005fca19 @daint:gpu+intel
P: latency: 21.37 us (r:0, l:None, u:None)
[       OK ] (17/22) OSUAllreduceTest %mpi_tasks=8 /005fca19 @daint:gpu+gnu
P: latency: 10.15 us (r:0, l:None, u:None)
[       OK ] (18/22) OSUAllreduceTest %mpi_tasks=8 /005fca19 @daint:gpu+nvidia
P: latency: 52.87 us (r:0, l:None, u:None)
[       OK ] (19/22) OSUAllreduceTest %mpi_tasks=16 /7f033d39 @daint:gpu+nvidia
P: latency: 64.77 us (r:0, l:None, u:None)
[       OK ] (20/22) OSUBandwidthTest /764cdb0b @daint:gpu+intel
P: bandwidth: 9118.51 MB/s (r:0, l:None, u:None)
[       OK ] (21/22) OSUBandwidthTest /764cdb0b @daint:gpu+nvidia
P: bandwidth: 8476.18 MB/s (r:0, l:None, u:None)
[       OK ] (22/22) OSUBandwidthTest /764cdb0b @daint:gpu+gnu
P: bandwidth: 8326.06 MB/s (r:0, l:None, u:None)
[----------] all spawned checks have finished

[ PASSED  ] Ran 22/22 test case(s) from 8 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Tue Nov 15 18:27:13 2022
Run report saved in '/home/user/.reframe/reports/run-report-3.json'
Log file(s) saved in '/tmp/rfm-r1a7v0w3.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 immediately.

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
```

```
[ReFrame Setup]
  version:           4.0.0-dev.2
  command:           './bin/reframe -c tutorials/deps/osu_benchmarks.py -n␣
→OSULatencyTest -l'
  launched by:       user@host
  working directory: '/home/user/Devel/reframe'
```

```
  settings files:    '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path: '/home/user/Devel/reframe/tutorials/deps/osu_benchmarks.py'
  stage directory:   '/home/user/Devel/reframe/stage'
  output directory:  '/home/user/Devel/reframe/output'
  log files:         '/tmp/rfm-7id7z75s.log'

[List of matched checks]
- OSULatencyTest /14f35a43
    ^OSUBuildTest /19b4fb56
      ^OSUDownloadTest /7de668df
Found 3 check(s)

Log file(s) saved in '/tmp/rfm-7id7z75s.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:           4.0.0-dev.2
  command:           './bin/reframe -c tutorials/deps/osu_benchmarks.py -n␣
→OSULatencyTest --system=daint:gpu -l'
  launched by:       user@host
  working directory: '/home/user/Devel/reframe'
  settings files:    '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path: '/home/user/Devel/reframe/tutorials/deps/osu_benchmarks.py'
  stage directory:   '/home/user/Devel/reframe/stage'
  output directory:  '/home/user/Devel/reframe/output'
  log files:         '/tmp/rfm-12gjxnvc.log'

WARNING: could not resolve dependency: ('OSUBuildTest', 'daint:gpu', 'gnu') ->
→'OSUDownloadTest'
WARNING: could not resolve dependency: ('OSUBuildTest', 'daint:gpu', 'intel') ->
→'OSUDownloadTest'
WARNING: could not resolve dependency: ('OSUBuildTest', 'daint:gpu', 'nvidia') ->
→'OSUDownloadTest'
WARNING: skipping all dependent test cases
  - ('OSUBuildTest', 'daint:gpu', 'intel')
  - ('OSUBandwidthTest', 'daint:gpu', 'intel')
  - ('OSUBuildTest', 'daint:gpu', 'nvidia')
  - ('OSULatencyTest', 'daint:gpu', 'intel')
  - ('OSUAllreduceTest_3', 'daint:gpu', 'nvidia')
  - ('OSUBuildTest', 'daint:gpu', 'gnu')
  - ('OSUAllreduceTest_1', 'daint:gpu', 'nvidia')
  - ('OSUAllreduceTest_0', 'daint:gpu', 'intel')
  - ('OSUAllreduceTest_2', 'daint:gpu', 'nvidia')
  - ('OSUBandwidthTest', 'daint:gpu', 'gnu')
  - ('OSULatencyTest', 'daint:gpu', 'gnu')
```

```
   - ('OSUAllreduceTest_2', 'daint:gpu', 'intel')
   - ('OSUAllreduceTest_3', 'daint:gpu', 'intel')
   - ('OSUAllreduceTest_1', 'daint:gpu', 'intel')
   - ('OSUAllreduceTest_0', 'daint:gpu', 'nvidia')
   - ('OSUBandwidthTest', 'daint:gpu', 'nvidia')
   - ('OSULatencyTest', 'daint:gpu', 'nvidia')
   - ('OSUAllreduceTest_2', 'daint:gpu', 'gnu')
   - ('OSUAllreduceTest_1', 'daint:gpu', 'gnu')
   - ('OSUAllreduceTest_3', 'daint:gpu', 'gnu')
   - ('OSUAllreduceTest_0', 'daint:gpu', 'gnu')


[List of matched checks]
Found 0 check(s)


Log file(s) saved in '/tmp/rfm-12gjxnvc.log'
```

Listing Dependencies

As shown in the listing of `OSULatencyTest` before, the full dependency chain of the test is listed along with the test. Each target dependency is printed in a new line prefixed by the `^` character and indented proportionally to its level. If a target dependency appears in multiple paths, it will only be listed once.

The default test listing will list the dependencies at the test level or the *conceptual* dependencies. ReFrame generates multiple test cases from each test depending on the target system configuration. We have seen in the *Tutorial 1: Getting Started with ReFrame* already how the STREAM benchmark generated many more test cases when it was run in a HPC system with multiple partitions and programming environments. These are the *actual* depedencies and form the actual test case graph that will be executed by the runtime. The mapping of a test to its concrete test cases that will be executed on a system is called *test concretization*. You can view the exact concretization of the selected tests with `--list=concretized` or simply `-lC`. Here is how the OSU benchmarks of this tutorial are concretized on the system `daint`:

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

```
[ReFrame Setup]
  version:           4.0.0-dev.2
  command:           './bin/reframe -c tutorials/deps/osu_benchmarks.py -lC'
  launched by:       user@host
  working directory: '/home/user/Devel/reframe'
  settings files:    '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path: '/home/user/Devel/reframe/tutorials/deps/osu_benchmarks.py'
  stage directory:   '/home/user/Devel/reframe/stage'
  output directory:  '/home/user/Devel/reframe/output'
  log files:         '/tmp/rfm-fremrbwf.log'

[List of matched checks]
- OSUAllreduceTest %mpi_tasks=16 /7f033d39 @daint:gpu+gnu
    ^OSUBuildTest /19b4fb56 @daint:gpu+gnu
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=16 /7f033d39 @daint:gpu+intel
    ^OSUBuildTest /19b4fb56 @daint:gpu+intel
      ^OSUDownloadTest /7de668df @daint:login+builtin
```

```
- OSUAllreduceTest %mpi_tasks=16 /7f033d39 @daint:gpu+nvidia
    ^OSUBuildTest /19b4fb56 @daint:gpu+nvidia
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=8 /005fca19 @daint:gpu+gnu
    ^OSUBuildTest /19b4fb56 @daint:gpu+gnu
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=8 /005fca19 @daint:gpu+intel
    ^OSUBuildTest /19b4fb56 @daint:gpu+intel
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=8 /005fca19 @daint:gpu+nvidia
    ^OSUBuildTest /19b4fb56 @daint:gpu+nvidia
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=4 /84b85d90 @daint:gpu+gnu
    ^OSUBuildTest /19b4fb56 @daint:gpu+gnu
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=4 /84b85d90 @daint:gpu+intel
    ^OSUBuildTest /19b4fb56 @daint:gpu+intel
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=4 /84b85d90 @daint:gpu+nvidia
    ^OSUBuildTest /19b4fb56 @daint:gpu+nvidia
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=2 /9d550c4f @daint:gpu+gnu
    ^OSUBuildTest /19b4fb56 @daint:gpu+gnu
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=2 /9d550c4f @daint:gpu+intel
    ^OSUBuildTest /19b4fb56 @daint:gpu+intel
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUAllreduceTest %mpi_tasks=2 /9d550c4f @daint:gpu+nvidia
    ^OSUBuildTest /19b4fb56 @daint:gpu+nvidia
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUBandwidthTest /764cdb0b @daint:gpu+gnu
    ^OSUBuildTest /19b4fb56 @daint:gpu+gnu
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUBandwidthTest /764cdb0b @daint:gpu+intel
    ^OSUBuildTest /19b4fb56 @daint:gpu+intel
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSUBandwidthTest /764cdb0b @daint:gpu+nvidia
    ^OSUBuildTest /19b4fb56 @daint:gpu+nvidia
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSULatencyTest /14f35a43 @daint:gpu+gnu
    ^OSUBuildTest /19b4fb56 @daint:gpu+gnu
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSULatencyTest /14f35a43 @daint:gpu+intel
    ^OSUBuildTest /19b4fb56 @daint:gpu+intel
      ^OSUDownloadTest /7de668df @daint:login+builtin
- OSULatencyTest /14f35a43 @daint:gpu+nvidia
    ^OSUBuildTest /19b4fb56 @daint:gpu+nvidia
      ^OSUDownloadTest /7de668df @daint:login+builtin
Concretized 22 test case(s)

Log file(s) saved in '/tmp/rfm-fremrbwf.log'
```

Notice how the various test cases of the run benchmarks depend on the corresponding test cases of the build tests.

The concretization of test cases changes if a specifc partition or programming environment is passed from the command line or, of course, if the test is run on a different system. If we scope our programming environments to `gnu` and `builtin` only, ReFrame will generate 8 test cases only instead of 22:

> **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
```

```
[ReFrame Setup]
  version:            4.0.0-dev.2
  command:            './bin/reframe -c tutorials/deps/osu_benchmarks.py -n␣
→OSULatencyTest -L -p builtin -p gnu'
  launched by:        user@host
  working directory:  '/home/user/Devel/reframe'
  settings files:     '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path:  '/home/user/Devel/reframe/tutorials/deps/osu_benchmarks.py'
  stage directory:    '/home/user/Devel/reframe/stage'
  output directory:   '/home/user/Devel/reframe/output'
  log files:          '/tmp/rfm-7hnco47r.log'

[List of matched checks]
- OSULatencyTest /14f35a43 [variant: 0, file: '/home/user/Devel/reframe/tutorials/deps/
→osu_benchmarks.py']
    ^OSUBuildTest /19b4fb56 [variant: 0, file: '/home/user/Devel/reframe/tutorials/deps/
→osu_benchmarks.py']
      ^OSUDownloadTest /7de668df [variant: 0, file: '/home/user/Devel/reframe/tutorials/
→deps/osu_benchmarks.py']
Found 3 check(s)

Log file(s) saved in '/tmp/rfm-7hnco47r.log'
```

To gain a deeper understanding on how test dependencies work in Reframe, please refer to *How Test Dependencies Work In ReFrame*.

Depending on Parameterized Tests

As shown earlier in this section, tests define their dependencies by referencing the target tests by their unique name. This is straightforward when referring to regular tests, where their name matches the class name, but it becomes cumbersome trying to refer to a parameterized tests, since no safe assumption should be made as of the variant number of the test or how the parameters are encoded in the name. In order to safely and reliably refer to a parameterized test, you should use the `get_variant_nums()` and `variant_name()` class methods as shown in the following example:

```
# Copyright 2016-2022 Swiss National Supercomputing Centre (CSCS/ETH Zurich)
# ReFrame Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: BSD-3-Clause

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

(continues on next page)

```python
@rfm.simple_test
class TestA(rfm.RunOnlyRegressionTest):
    z = parameter(range(10))
    executable = 'echo'
    valid_systems = ['*']
    valid_prog_environs = ['*']

    @run_after('init')
    def set_exec_opts(self):
        self.executable_opts = [str(self.z)]

    @sanity_function
    def validate(self):
        return sn.assert_eq(
            sn.extractsingle(r'\d+', self.stdout, 0, int), self.z
        )


@rfm.simple_test
class TestB(rfm.RunOnlyRegressionTest):
    executable = 'echo'
    valid_systems = ['*']
    valid_prog_environs = ['*']
    sanity_patterns = sn.assert_true(1)

    @run_after('init')
    def setdeps(self):
        variants = TestA.get_variant_nums(z=lambda x: x > 5)
        for v in variants:
            self.depends_on(TestA.variant_name(v))
```

In this example, `TestB` depends only on selected variants of `TestA`. The `get_variant_nums()` method accepts a set of key-value pairs representing the target test parameters and selector functions and returns the list of the variant numbers that correspond to these variants. Using the `variant_name()` subsequently, we can get the actual name of the variant.

```
./bin/reframe -c tutorials/deps/parameterized.py -l
```

```
[ReFrame Setup]
  version:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -c tutorials/deps/parameterized.py -l'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
→tresa.py'
  check search path: '/home/user/Repositories/reframe/tutorials/deps/parameterized.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-u9ryq5d3.log'

[List of matched checks]
```

```
- TestB /cc291487
    ^TestA %z=9 /034f091a
    ^TestA %z=8 /a093d19f
    ^TestA %z=7 /77b4b8e6
    ^TestA %z=6 /40ce4759
- TestA %z=5 /aa0cffc9
- TestA %z=4 /83cd5dec
- TestA %z=3 /1c51609b
- TestA %z=2 /707b752c
- TestA %z=1 /c65657d5
- TestA %z=0 /1b9f44df
Found 11 check(s)

Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-u9ryq5d3.log'
```

2.3.4 Tutorial 4: Using Test Fixtures

New in version 3.9.0.

A fixture in ReFrame is a test that manages a resource of another test. Fixtures can be chained to create essentially a graph of dependencies. Similarly to test dependencies, the test that uses the fixture will not execute until its fixture has executed. In this tutorial, we will rewrite the OSU benchmarks example presented in *Tutorial 3: Using Dependencies in ReFrame Tests* using fixtures. We will cover only the basic concepts of fixtures that will allow you to start using them in your tests. For the full documentation of the test fixtures, you should refer to the *Test API Reference* documentation.

The full example of the OSU benchmarks using test fixtures is shown below with the relevant parts highlighted:

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


class fetch_osu_benchmarks(rfm.RunOnlyRegressionTest):
    descr = 'Fetch OSU benchmarks'
    version = variable(str, value='5.6.2')
    executable = 'wget'
    executable_opts = [
        f'http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-
→{version}.tar.gz'  # noqa: E501
    ]
    local = True

    @sanity_function
    def validate_download(self):
        return sn.assert_eq(self.job.exitcode, 0)


class build_osu_benchmarks(rfm.CompileOnlyRegressionTest):
    descr = 'Build OSU benchmarks'
    build_system = 'Autotools'
    build_prefix = variable(str)
    osu_benchmarks = fixture(fetch_osu_benchmarks, scope='session')
```

```python
    @run_before('compile')
    def prepare_build(self):
        tarball = f'osu-micro-benchmarks-{self.osu_benchmarks.version}.tar.gz'
        self.build_prefix = tarball[:-7]  # remove .tar.gz extension

        fullpath = os.path.join(self.osu_benchmarks.stagedir, tarball)
        self.prebuild_cmds = [
            f'cp {fullpath} {self.stagedir}',
            f'tar xzf {tarball}',
            f'cd {self.build_prefix}'
        ]
        self.build_system.max_concurrency = 8

    @sanity_function
    def validate_build(self):
        # If compilation fails, the test would fail in any case, so nothing to
        # further validate here.
        return True


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

    valid_systems = ['daint:gpu']
    valid_prog_environs = ['gnu', 'nvidia', 'intel']
    num_tasks = 2
    num_tasks_per_node = 1
    osu_binaries = fixture(build_osu_benchmarks, scope='environment')

    @sanity_function
    def validate_test(self):
        return sn.assert_found(r'^8', self.stdout)


@rfm.simple_test
class osu_latency_test(OSUBenchmarkTestBase):
    descr = 'OSU latency test'

    @run_before('run')
    def prepare_run(self):
        self.executable = os.path.join(
            self.osu_binaries.stagedir,
            self.osu_binaries.build_prefix,
            'mpi', 'pt2pt', 'osu_latency'
        )
        self.executable_opts = ['-x', '100', '-i', '1000']

    @performance_function('us')
    def latency(self):
        return sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
```

```python
@rfm.simple_test
class osu_bandwidth_test(OSUBenchmarkTestBase):
    descr = 'OSU bandwidth test'

    @run_before('run')
    def prepare_run(self):
        self.executable = os.path.join(
            self.osu_binaries.stagedir,
            self.osu_binaries.build_prefix,
            'mpi', 'pt2pt', 'osu_bw'
        )
        self.executable_opts = ['-x', '100', '-i', '1000']

    @performance_function('MB/s')
    def bandwidth(self):
        return sn.extractsingle(r'^4194304\s+(\S+)',
                                self.stdout, 1, float)


@rfm.simple_test
class osu_allreduce_test(OSUBenchmarkTestBase):
    mpi_tasks = parameter(1 << i for i in range(1, 5))
    descr = 'OSU Allreduce test'

    @run_before('run')
    def set_executable(self):
        self.num_tasks = self.mpi_tasks
        self.executable = os.path.join(
            self.osu_binaries.stagedir,
            self.osu_binaries.build_prefix,
            'mpi', 'collective', 'osu_allreduce'
        )
        self.executable_opts = ['-m', '8', '-x', '1000', '-i', '20000']

    @performance_function('us')
    def latency(self):
        return sn.extractsingle(r'^8\s+(\S+)', self.stdout, 1, float)
```

Let's start from the leaf tests, i.e. the tests that execute the benchmarks (osu_latency_test, osu_bandwidth_test and osu_allreduce_test). As in the dependencies example, all these tests derive from the OSUBenchmarkTestBase, where we define a fixture that will take care of generating the binaries of the tests:

```python
osu_binaries = fixture(build_osu_benchmarks, scope='environment')
```

A test defines a fixture using the `fixture()` builtin and assigns it a name by assigning the return value of the builtin to a test variable, here `osu_binaries`. This name will be used later to access the resource managed by the fixture.

As stated previously, a fixture is another full-fledged ReFrame test, here the `build_osu_benchmarks` which will take care of building the OSU benchmarks. Each fixture is associated with a scope. This practically indicates at which level a fixture is shared with other tests. There are four fixture scopes, which are listed below in decreasing order of generality:

- `session`: A fixture with this scope will be executed once per ReFrame run session and will be shared across the whole run.

- `partition`: A fixture with this scope will be executed once per partition and will be shared across all tests that run in that partition.

- `environment`: A fixture with this scope will be executed once per partition and environment combination and will be shared across all tests that run with this partition and environment combination.

- `test`: A fixture with this scope is private to the test and will be executed for each test case.

In this example, we need to build once the OSU benchmarks for each partition and environment combination, so we use the `environment` scope.

Accessing the fixture is very straightforward. The fixture's result is accessible after the *setup* pipeline stage through the corresponding variable in the test that is defining it. Since a fixture is a standard ReFrame test, you can access any information of the test. The individual benchmarks do exactly that:

```python
@run_before('run')
def prepare_run(self):
    self.executable = os.path.join(
        self.osu_binaries.stagedir,
        self.osu_binaries.build_prefix,
        'mpi', 'pt2pt', 'osu_latency'
    )
    self.executable_opts = ['-x', '100', '-i', '1000']
```

Here we construct the final executable path by accessing the standard *stagedir* attribute of the test as well as the custom-defined `build_prefix` variable of the `build_osu_benchmarks` fixture.

Let's inspect now the `build_osu_benchmarks` fixture:

```python
class build_osu_benchmarks(rfm.CompileOnlyRegressionTest):
    descr = 'Build OSU benchmarks'
    build_system = 'Autotools'
    build_prefix = variable(str)
    osu_benchmarks = fixture(fetch_osu_benchmarks, scope='session')

    @run_before('compile')
    def prepare_build(self):
        tarball = f'osu-micro-benchmarks-{self.osu_benchmarks.version}.tar.gz'
        self.build_prefix = tarball[:-7]  # remove .tar.gz extension

        fullpath = os.path.join(self.osu_benchmarks.stagedir, tarball)
        self.prebuild_cmds = [
            f'cp {fullpath} {self.stagedir}',
            f'tar xzf {tarball}',
            f'cd {self.build_prefix}'
        ]
        self.build_system.max_concurrency = 8

    @sanity_function
    def validate_build(self):
        # If compilation fails, the test would fail in any case, so nothing to
        # further validate here.
        return True
```

It is obvious that it is a normal ReFrame test except that it does not need to be decorated with the `@simple_test` decorator. This means that the test will only be executed if it is a fixture of another test. If it was decorated, it would be executed both as a standalone test and as a fixture of another test. Another detail is that this test does not define the

valid_systems and *valid_prog_environs* variables. Fixtures inherit those variables from the test that owns them depending on the scope.

Similarly to `OSUBenchmarkTestBase`, this test uses a fixture that fetches the OSU benchmarks sources. We could fetch the OSU benchmarks in this test, but we choose to separate the two primarily for demonstration purposes, but it would also make sense in cases that the data fetch is too slow.

The `osu_benchmarks` fixture is defined at session scope, since we only need to download the benchmarks once for the whole session:

```
osu_benchmarks = fixture(fetch_osu_benchmarks, scope='session')
```

The rest of the test is very straightforward.

Let's inspect the last fixture, the `fetch_osu_benchmarks`:

```python
class fetch_osu_benchmarks(rfm.RunOnlyRegressionTest):
    descr = 'Fetch OSU benchmarks'
    version = variable(str, value='5.6.2')
    executable = 'wget'
    executable_opts = [
        f'http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-
{version}.tar.gz'  # noqa: E501
    ]
    local = True

    @sanity_function
    def validate_download(self):
        return sn.assert_eq(self.job.exitcode, 0)
```

There is nothing special to this test – it is just an ordinary test – except that we force it to execute locally by setting its *local* variable. The reason for that is that a fixture at session scope can execute with any partition/environment combination, so ReFrame could have to spawn a job in case it has chosen a remote partition to launch this fixture on. For this reason, we simply force it to execute locally regardless of the chosen partition.

It is now time to run the new tests, but let us first list them:

```
reframe -c tutorials/fixtures/osu_benchmarks.py -l
```

```
[ReFrame Setup]
  version:           4.0.0-dev.2
  command:           './bin/reframe -c tutorials/fixtures/osu_benchmarks.py -l'
  launched by:       user@host
  working directory: '/home/user/Devel/reframe'
  settings files:    '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path: '/home/user/Devel/reframe/tutorials/fixtures/osu_benchmarks.py'
  stage directory:   '/home/user/Devel/reframe/stage'
  output directory:  '/home/user/Devel/reframe/output'
  log files:         '/tmp/rfm-to7wa4gh.log'

[List of matched checks]
- osu_allreduce_test %mpi_tasks=16 /1fe48834
    ^build_osu_benchmarks ~daint:gpu+gnu /f3269d42
      ^fetch_osu_benchmarks ~daint /79cd6023
    ^build_osu_benchmarks ~daint:gpu+intel /4d450880
```

(continues on next page)

```
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+nvidia /e9b8d152
          ^fetch_osu_benchmarks ~daint /79cd6023
- osu_allreduce_test %mpi_tasks=8 /ae01c137
       ^build_osu_benchmarks ~daint:gpu+gnu /f3269d42
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+intel /4d450880
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+nvidia /e9b8d152
          ^fetch_osu_benchmarks ~daint /79cd6023
- osu_allreduce_test %mpi_tasks=4 /2129dc34
       ^build_osu_benchmarks ~daint:gpu+gnu /f3269d42
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+intel /4d450880
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+nvidia /e9b8d152
          ^fetch_osu_benchmarks ~daint /79cd6023
- osu_allreduce_test %mpi_tasks=2 /9f29c081
       ^build_osu_benchmarks ~daint:gpu+gnu /f3269d42
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+intel /4d450880
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+nvidia /e9b8d152
          ^fetch_osu_benchmarks ~daint /79cd6023
- osu_bandwidth_test /026711a1
       ^build_osu_benchmarks ~daint:gpu+gnu /f3269d42
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+intel /4d450880
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+nvidia /e9b8d152
          ^fetch_osu_benchmarks ~daint /79cd6023
- osu_latency_test /d2c978ad
       ^build_osu_benchmarks ~daint:gpu+gnu /f3269d42
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+intel /4d450880
          ^fetch_osu_benchmarks ~daint /79cd6023
       ^build_osu_benchmarks ~daint:gpu+nvidia /e9b8d152
          ^fetch_osu_benchmarks ~daint /79cd6023
Found 6 check(s)

Log file(s) saved in '/tmp/rfm-to7wa4gh.log'
```

Notice how the `build_osu_benchmarks` fixture is populated three times, once for each partition and environment combination, and the `fetch_osu_benchmarks` is generated only once. The following figure shows visually the conceptual dependencies of the `osu_bandwidth_test`.

Fig. 1: Expanded fixtures and dependencies for the OSU benchmarks example.

A *scope* part is added to the base name of the fixture, which in this figure is indicated with red color.

Under the hood, fixtures use the test dependency mechanism which is described in *How Test Dependencies Work In ReFrame*. The dependencies listed by default and shown in the previous figure are conceptual. Depending on the

available partitions and environments, tests and fixtures can be concretized differently. Fixtures in particular are also more flexible in the way they can be concretized depending on their scope. The following listing and figure show the concretization of the osu_bandwidth_test:

```
reframe -c tutorials/fixtures/osu_benchmarks.py -n osu_bandwidth_test -lC
```

```
[ReFrame Setup]
  version:           4.0.0-dev.2
  command:           './bin/reframe -c tutorials/fixtures/osu_benchmarks.py -n osu_
→bandwidth_test -lC'
  launched by:       user@host
  working directory: '/home/user/Devel/reframe'
  settings files:    '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path: '/home/user/Devel/reframe/tutorials/fixtures/osu_benchmarks.py'
  stage directory:   '/home/user/Devel/reframe/stage'
  output directory:  '/home/user/Devel/reframe/output'
  log files:         '/tmp/rfm-m1w2t4eh.log'

[List of matched checks]
- osu_bandwidth_test /026711a1 @daint:gpu+gnu
    ^build_osu_benchmarks ~daint:gpu+gnu /f3269d42 @daint:gpu+gnu
      ^fetch_osu_benchmarks ~daint /79cd6023 @daint:gpu+gnu
- osu_bandwidth_test /026711a1 @daint:gpu+intel
    ^build_osu_benchmarks ~daint:gpu+intel /4d450880 @daint:gpu+intel
      ^fetch_osu_benchmarks ~daint /79cd6023 @daint:gpu+gnu
- osu_bandwidth_test /026711a1 @daint:gpu+nvidia
    ^build_osu_benchmarks ~daint:gpu+nvidia /e9b8d152 @daint:gpu+nvidia
      ^fetch_osu_benchmarks ~daint /79cd6023 @daint:gpu+gnu
Concretized 7 test case(s)

Log file(s) saved in '/tmp/rfm-m1w2t4eh.log'
```

Fig. 2: The actual dependencies for the OSU benchmarks example using fixtures.

The first thing to notice here is how the individual test cases of osu_bandwidth_test depend only the specific fixtures for their scope: when osu_bandwidth_test runs on the daint:gpu partition using the gnu compiler it will only depend on the build_osu_benchmarks~daint:gpu+gnu fixture. The second thing to notice is where the fetch_osu_benchmarks~daint fixture will run. Since this is a *session* fixture, ReFrame has arbitrarily chosen to run it on daint:gpu using the gnu environment. A session fixture can run on any combination of valid partitions and environments. The following listing and figure show how the test dependency DAG is concretized when we scope the valid programming environments from the command line using -p nvidia.

```
reframe -c tutorials/fixtures/osu_benchmarks.py -n osu_bandwidth_test -lC -p nvidia
```

```
[ReFrame Setup]
  version:           3.10.0-dev.3+605af31a
  command:           './bin/reframe -c tutorials/fixtures/osu_benchmarks.py -n osu_
→bandwidth_test -lC -p nvidia'
  launched by:       user@host
  working directory: '/home/user/Devel/reframe'
  settings file:     '/home/user/Devel/reframe/tutorials/config/settings.py'
  check search path: '/home/user/Devel/reframe/tutorials/fixtures/osu_benchmarks.py'
```

<div align="right">(continues on next page)</div>

```
  stage directory:   '/home/user/Devel/reframe/stage'
  output directory:  '/home/user/Devel/reframe/output'


[List of matched checks]
- osu_bandwidth_test @daint:gpu+nvidia
    ^build_osu_benchmarks ~daint:gpu+nvidia @daint:gpu+nvidia
      ^fetch_osu_benchmarks ~daint @daint:gpu+nvidia
Concretized 3 test case(s)

Log file(s) saved in '/tmp/rfm-dnfdagj8.log'
```

Fig. 3: The dependency graph concretized for the 'nvidia' environment only.

Notice how the `fetch_osu_benchmarks~daint` fixture is selected to run in the only valid partition/environment combination. This is an important difference compared to the same example written using raw dependencies in *How Test Dependencies Work In ReFrame*, in which case in order not to have unresolved dependencies, we would need to specify the valid programming environment of the test that fetches the sources. Fixtures do not need that, since you can impose less strict constraints by setting their scope accordingly.

Finally, let's run all the benchmarks at once:

```
[ReFrame Setup]
  version:           4.0.0-dev.2
  command:           './bin/reframe -c tutorials/fixtures/osu_benchmarks.py -r'
  launched by:       user@host
  working directory: '/home/user/Devel/reframe'
  settings files:    '<builtin>', '/home/user/Devel/reframe/tutorials/config/daint.py'
  check search path: '/home/user/Devel/reframe/tutorials/fixtures/osu_benchmarks.py'
  stage directory:   '/home/user/Devel/reframe/stage'
  output directory:  '/home/user/Devel/reframe/output'
  log files:         '/tmp/rfm-63lwmv4b.log'


[==========] Running 10 check(s)
[==========] Started on Tue Nov 15 18:27:17 2022


[----------] start processing checks
[ RUN      ] fetch_osu_benchmarks ~daint /79cd6023 @daint:gpu+gnu
[       OK ] ( 1/22) fetch_osu_benchmarks ~daint /79cd6023 @daint:gpu+gnu
[ RUN      ] build_osu_benchmarks ~daint:gpu+gnu /f3269d42 @daint:gpu+gnu
[ RUN      ] build_osu_benchmarks ~daint:gpu+intel /4d450880 @daint:gpu+intel
[ RUN      ] build_osu_benchmarks ~daint:gpu+nvidia /e9b8d152 @daint:gpu+nvidia
[       OK ] ( 2/22) build_osu_benchmarks ~daint:gpu+gnu /f3269d42 @daint:gpu+gnu
[ RUN      ] osu_allreduce_test %mpi_tasks=16 /1fe48834 @daint:gpu+gnu
[ RUN      ] osu_allreduce_test %mpi_tasks=8 /ae01c137 @daint:gpu+gnu
[ RUN      ] osu_allreduce_test %mpi_tasks=4 /2129dc34 @daint:gpu+gnu
[ RUN      ] osu_allreduce_test %mpi_tasks=2 /9f29c081 @daint:gpu+gnu
[ RUN      ] osu_bandwidth_test /026711a1 @daint:gpu+gnu
[ RUN      ] osu_latency_test /d2c978ad @daint:gpu+gnu
[       OK ] ( 3/22) build_osu_benchmarks ~daint:gpu+intel /4d450880 @daint:gpu+intel
[       OK ] ( 4/22) build_osu_benchmarks ~daint:gpu+nvidia /e9b8d152 @daint:gpu+nvidia
[ RUN      ] osu_allreduce_test %mpi_tasks=16 /1fe48834 @daint:gpu+intel
```

```
[ RUN      ] osu_allreduce_test %mpi_tasks=16 /1fe48834 @daint:gpu+nvidia
[ RUN      ] osu_allreduce_test %mpi_tasks=8 /ae01c137 @daint:gpu+intel
[ RUN      ] osu_allreduce_test %mpi_tasks=8 /ae01c137 @daint:gpu+nvidia
[ RUN      ] osu_allreduce_test %mpi_tasks=4 /2129dc34 @daint:gpu+intel
[ RUN      ] osu_allreduce_test %mpi_tasks=4 /2129dc34 @daint:gpu+nvidia
[ RUN      ] osu_allreduce_test %mpi_tasks=2 /9f29c081 @daint:gpu+intel
[ RUN      ] osu_allreduce_test %mpi_tasks=2 /9f29c081 @daint:gpu+nvidia
[ RUN      ] osu_bandwidth_test /026711a1 @daint:gpu+intel
[ RUN      ] osu_bandwidth_test /026711a1 @daint:gpu+nvidia
[ RUN      ] osu_latency_test /d2c978ad @daint:gpu+intel
[ RUN      ] osu_latency_test /d2c978ad @daint:gpu+nvidia
[       OK ] ( 5/22) osu_allreduce_test %mpi_tasks=2 /9f29c081 @daint:gpu+gnu
P: latency: 2.76 us (r:0, l:None, u:None)
[       OK ] ( 6/22) osu_allreduce_test %mpi_tasks=2 /9f29c081 @daint:gpu+intel
P: latency: 1.68 us (r:0, l:None, u:None)
[       OK ] ( 7/22) osu_allreduce_test %mpi_tasks=4 /2129dc34 @daint:gpu+intel
P: latency: 4.89 us (r:0, l:None, u:None)
[       OK ] ( 8/22) osu_latency_test /d2c978ad @daint:gpu+intel
P: latency: 1.54 us (r:0, l:None, u:None)
[       OK ] ( 9/22) osu_latency_test /d2c978ad @daint:gpu+gnu
P: latency: 1.17 us (r:0, l:None, u:None)
[       OK ] (10/22) osu_allreduce_test %mpi_tasks=4 /2129dc34 @daint:gpu+gnu
P: latency: 3.22 us (r:0, l:None, u:None)
[       OK ] (11/22) osu_allreduce_test %mpi_tasks=16 /1fe48834 @daint:gpu+gnu
P: latency: 13.84 us (r:0, l:None, u:None)
[       OK ] (12/22) osu_allreduce_test %mpi_tasks=16 /1fe48834 @daint:gpu+nvidia
P: latency: 30.77 us (r:0, l:None, u:None)
[       OK ] (13/22) osu_allreduce_test %mpi_tasks=4 /2129dc34 @daint:gpu+nvidia
P: latency: 5.74 us (r:0, l:None, u:None)
[       OK ] (14/22) osu_allreduce_test %mpi_tasks=16 /1fe48834 @daint:gpu+intel
P: latency: 14.77 us (r:0, l:None, u:None)
[       OK ] (15/22) osu_allreduce_test %mpi_tasks=2 /9f29c081 @daint:gpu+nvidia
P: latency: 4.5 us (r:0, l:None, u:None)
[       OK ] (16/22) osu_allreduce_test %mpi_tasks=8 /ae01c137 @daint:gpu+nvidia
P: latency: 33.93 us (r:0, l:None, u:None)
[       OK ] (17/22) osu_allreduce_test %mpi_tasks=8 /ae01c137 @daint:gpu+intel
P: latency: 20.9 us (r:0, l:None, u:None)
[       OK ] (18/22) osu_latency_test /d2c978ad @daint:gpu+nvidia
P: latency: 1.18 us (r:0, l:None, u:None)
[       OK ] (19/22) osu_allreduce_test %mpi_tasks=8 /ae01c137 @daint:gpu+gnu
P: latency: 10.14 us (r:0, l:None, u:None)
[       OK ] (20/22) osu_bandwidth_test /026711a1 @daint:gpu+gnu
P: bandwidth: 9785.43 MB/s (r:0, l:None, u:None)
[       OK ] (21/22) osu_bandwidth_test /026711a1 @daint:gpu+intel
P: bandwidth: 9841.26 MB/s (r:0, l:None, u:None)
[       OK ] (22/22) osu_bandwidth_test /026711a1 @daint:gpu+nvidia
P: bandwidth: 9824.01 MB/s (r:0, l:None, u:None)
[----------] all spawned checks have finished

[  PASSED  ] Ran 22/22 test case(s) from 10 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Tue Nov 15 18:30:34 2022
Run report saved in '/home/user/.reframe/reports/run-report-4.json'
```

```
Log file(s) saved in '/tmp/rfm-63lwmv4b.log'
```

Tip: A reasonable question is how to choose between fixtures and dependencies?

The rule of thumb is use fixtures if your test needs to use any resource of the target test and use dependencies if you simply want to impose an order of execution for your tests.

2.3.5 Tutorial 5: Using Build Automation Tools As a Build System

In this tutorial we will present how to use Easybuild and Spack as a build system for a ReFrame test. The example uses 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 and has a working knowledge of EasyBuild and Spack.

Using EasyBuild to Build the Test Code

New in version 3.5.0.

Let's consider a simple ReFrame test that installs `bzip2-1.0.6` given the easyconfig bzip2-1.0.6.eb and checks that the installed version is correct. The following code block shows the check, highlighting the lines specific to this tutorial:

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


@rfm.simple_test
class BZip2EBCheck(rfm.RegressionTest):
    descr = 'Demo test using EasyBuild to build the test code'
    valid_systems = ['*']
    valid_prog_environs = ['builtin']
    executable = 'bzip2'
    executable_opts = ['--help']
    build_system = 'EasyBuild'

    @run_before('compile')
    def setup_build_system(self):
        self.build_system.easyconfigs = ['bzip2-1.0.6.eb']
        self.build_system.options = ['-f']

    @run_before('run')
    def prepare_run(self):
        self.modules = self.build_system.generated_modules

    @sanity_function
    def assert_version(self):
        return sn.assert_found(r'Version 1.0.6', self.stderr)
```

The test looks pretty standard except for the highlighted blocks. Let's have a look first to the block in the `BZip2Check` class.

The first thing is to specify that the EasyBuild build system will be used. This is done by setting *build_system* to `'EasyBuild'`. Then, the software to be installed is passed as a list to *easyconfigs*. Here only one easyconfig is given, but more than one can be passed. Finally, through *options*, command line options can be passed to the `eb` executable. In this test we pass `-f` to make sure that `bzip2` will be built even if the module already exists externally.

For this test, ReFrame generates the following command to build and install the easyconfig:

```
export EASYBUILD_BUILDPATH={stagedir}/easybuild/build
export EASYBUILD_INSTALLPATH={stagedir}/easybuild
export EASYBUILD_PREFIX={stagedir}/easybuild
export EASYBUILD_SOURCEPATH={stagedir}/easybuild
eb bzip2-1.0.6.eb -f
```

ReFrame will keep all the files generated by EasyBuild (sources, temporary files, installed software and the corresponding modules) under the test's stage directory. For this reason it sets the relevant EasyBuild environment variables.

Tip: Users may set the EasyBuild prefix to a different location by setting the *prefix* attribute of the build system. This allows you to have the built software installed upon successful completion of the build phase, but if the test fails in a later stage (sanity, performance), the installed software will not be cleaned up automatically.

Note: ReFrame assumes that the `eb` executable is available on the system where the compilation is run (typically the local host where ReFrame is executed).

Now that we know everything related to building and installing the code, we can move to the part dealing with running it. To run the code, the generated modules need to be loaded in order to make the software available. The modules can be accessed through `generated_modules`, however, they are available only after EasyBuild completes the installation. This means that *modules* can be set only after the build phase finishes. For that, we can set *modules* in a class method wrapped by the `run_before()` built-in, specifying the `run` phase. This test will then run the following commands:

```
module load bzip/1.0.6
bzip2 --help
```

Packaging the installation

The EasyBuild build system offers a way of packaging the installation via EasyBuild's packaging support. To use this feature, the FPM package manager must be available. By setting the dictionary `package_opts` in the test, ReFrame will pass `--package-{key}={val}` to the EasyBuild invocation. For instance, the following can be set to package the installations as an rpm file:

```python
self.keep_files = ['easybuild/packages']
self.build_system.package_opts = {
    'type': 'rpm',
}
```

The packages are generated by EasyBuild in the stage directory. To retain them after the test succeeds, *keep_files* needs to be set.

Using Spack to Build the Test Code

New in version 3.6.1.

This example is the equivalent to the previous one, except that it uses Spack to build `bzip2`. Here is the test's code:

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


@rfm.simple_test
class BZip2SpackCheck(rfm.RegressionTest):
    descr = 'Demo test using Spack to build the test code'
    valid_systems = ['*']
    valid_prog_environs = ['builtin']
    executable = 'bzip2'
    executable_opts = ['--help']
    build_system = 'Spack'

    @run_before('compile')
    def setup_build_system(self):
        self.build_system.specs = ['bzip2@1.0.6']

    @sanity_function
    def assert_version(self):
        return sn.assert_found(r'Version 1.0.6', self.stderr)
```

When *build_system* is set to `'Spack'`, ReFrame will leverage Spack environments in order to build the test code. By default, ReFrame will create a new Spack environment in the test's stage directory and add the requested *specs* to it.

Note: Optional spec attributes, such as `target` and `os`, should be specified in *specs* and not as install options in *install_opts*.

Users may also specify an existing Spack environment by setting the *environment* attribute. In this case, ReFrame treats the environment as a *test resource* so it expects to find it under the test's *sourcesdir*, which defaults to `'src'`.

As with every other test, ReFrame will copy the test's resources to its stage directory before building it. ReFrame will then activate the generated environment (either the one provided by the user or the one generated by ReFrame), add the given specs using the `spack add` command and, finally, install the packages in the environment. Here is what ReFrame generates as a build script for this example:

```
spack env create -d rfm_spack_env
spack -e rfm_spack_env config add "config:install_tree:root:opt/spack"
spack -e rfm_spack_env add bzip2@1.0.6
spack -e rfm_spack_env install
```

As you might have noticed ReFrame expects that Spack is already installed on the system. The packages specified in the environment and the tests will be installed in the test's stage directory, where the environment is copied before building. Here is the stage directory structure:

```
stage/generic/default/builtin/BZip2SpackCheck/
├── rfm_spack_env
│   ├── spack
```

```
    │           └── opt
    │               └── spack
    │                   ├── bin
    │                   └── darwin-catalina-skylake
    ├── spack.lock
    └── spack.yaml
├── rfm_BZip2SpackCheck_build.err
├── rfm_BZip2SpackCheck_build.out
├── rfm_BZip2SpackCheck_build.sh
├── rfm_BZip2SpackCheck_job.err
├── rfm_BZip2SpackCheck_job.out
└── rfm_BZip2SpackCheck_job.sh
```

Finally, here is the generated run script that ReFrame uses to run the test, once its build has succeeded:

```bash
#!/bin/bash
spack env create -d rfm_spack_env
eval `spack -e rfm_spack_env load --sh bzip2@1.0.6`
bzip2 --help
```

From this point on, sanity and performance checking are exactly identical to any other ReFrame test.

Tip: While developing a test using Spack or EasyBuild as a build system, it can be useful to run ReFrame with the `--keep-stage-files` and `--dont-restage` options to prevent ReFrame from removing the test's stage directory upon successful completion of the test. For this particular type of test, these options will avoid having to rebuild the required package dependencies every time the test is retried.

2.3.6 Tutorial 6: 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. Generally, ReFrame will not print the full stack trace for user programming errors and will not block the test loading process. If a test has errors and cannot be loaded, an error message will be printed and the loading of the remaining tests will continue. In the following, we have inserted a small typo in the `hello2.py` tutorial example:

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

```
[ReFrame Setup]
  version:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -c tutorials/basics/hello -R -l'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
```

```
↪tresa.py'
  check search path: (R) '/home/user/Repositories/reframe/tutorials/basics/hello'
  stage directory:    '/home/user/Repositories/reframe/stage'
  output directory:   '/home/user/Repositories/reframe/output'
  log files:          '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-ldo5um3v.log'

WARNING: skipping test file '/home/user/Repositories/reframe/tutorials/basics/hello/
↪hello2.py': name error: tutorials/basics/hello/hello2.py:13: name 'paramter' is not␣
↪defined
    lang = paramter(['c', 'cpp'])
 (rerun with '-v' for more information)
[List of matched checks]
- HelloTest /2b3e4546
Found 1 check(s)

Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-ldo5um3v.log'
```

Notice how ReFrame prints also the source code line that caused the error. This is not always the case, however. ReFrame cannot always track a user error back to its source and this is particularly true for the ReFrame-specific syntactic elements, such as the class builtins. In such cases, ReFrame will just print the error message but not the source code context. In the following example, we introduce a typo in the argument of the @run_before decorator:

```
./bin/reframe: skipping test file '/Users/user/Repositories/reframe/tutorials/basics/
↪hello/hello2.py': reframe syntax error: invalid pipeline stage specified: 'compil'␣
↪(rerun with '-v' for more information)
[List of matched checks]
- HelloTest (found in '/Users/user/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 -l -v
```

```
[ReFrame Setup]
  version:            4.0.0-dev.2+5ea6b7a6
  command:            './bin/reframe -c tutorials/basics/hello -R -l -v'
  launched by:        user@host
  working directory:  '/home/user/Repositories/reframe'
  settings files:     '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
↪tresa.py'
  check search path: (R) '/home/user/Repositories/reframe/tutorials/basics/hello'
  stage directory:    '/home/user/Repositories/reframe/stage'
  output directory:   '/home/user/Repositories/reframe/output'
  log files:          '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-xs3l6jud.log'

WARNING: skipping test file '/home/user/Repositories/reframe/tutorials/basics/hello/
↪hello2.py': name error: tutorials/basics/hello/hello2.py:13: name 'paramter' is not␣
↪defined
    lang = paramter(['c', 'cpp'])
 (rerun with '-v' for more information)
```

```
Traceback (most recent call last):
  File "/home/user/Repositories/reframe/reframe/frontend/loader.py", line 205, in load_
→from_file
    util.import_module_from_file(filename, force)
  File "/home/user/Repositories/reframe/reframe/utility/__init__.py", line 109, in
→import_module_from_file
    return importlib.import_module(module_name)
  File "/usr/local/Cellar/python@3.10/3.10.7/Frameworks/Python.framework/Versions/3.10/
→lib/python3.10/importlib/__init__.py", line 126, in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
  File "<frozen importlib._bootstrap>", line 1050, in _gcd_import
  File "<frozen importlib._bootstrap>", line 1027, in _find_and_load
  File "<frozen importlib._bootstrap>", line 1006, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 688, in _load_unlocked
  File "<frozen importlib._bootstrap_external>", line 883, in exec_module
  File "<frozen importlib._bootstrap>", line 241, in _call_with_frames_removed
  File "/home/user/Repositories/reframe/tutorials/basics/hello/hello2.py", line 12, in
→<module>
    class HelloMultiLangTest(rfm.RegressionTest):
  File "/home/user/Repositories/reframe/tutorials/basics/hello/hello2.py", line 13, in
→HelloMultiLangTest
    lang = paramter(['c', 'cpp'])
NameError: name 'paramter' is not defined

Loaded 1 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
Final number of test cases: 2
[List of matched checks]
- HelloTest /2b3e4546
Found 1 check(s)

Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-xs3l6jud.log'
```

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

Debugging deferred expressions

Although deferred expressions that are used in sanity and performance functions 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.

```python
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 = ['*']

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

    @sanity_function
    def validate_output(self):
        return 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
```

```
[ReFrame Setup]
  version:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -C tutorials/config/tresa.py -c tutorials/basics/
→hello/hello2.py -r'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', 'tutorials/config/tresa.py'
  check search path: '/home/user/Repositories/reframe/tutorials/basics/hello/hello2.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-b22mnhb0.log'

[==========] Running 2 check(s)
[==========] Started on Sat Nov 12 19:00:58 2022

[----------] start processing checks
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @tresa:default+gnu
[ RUN      ] HelloMultiLangTest %lang=cpp /71bf65a3 @tresa:default+clang
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @tresa:default+gnu
[ RUN      ] HelloMultiLangTest %lang=c /7cfa870e @tresa:default+clang
rfm_job.out
rfm_job.out
[       OK ] (1/4) HelloMultiLangTest %lang=c /7cfa870e @tresa:default+gnu
rfm_job.out
rfm_job.out
[       OK ] (2/4) HelloMultiLangTest %lang=c /7cfa870e @tresa:default+clang
rfm_job.out
rfm_job.out
[       OK ] (3/4) HelloMultiLangTest %lang=cpp /71bf65a3 @tresa:default+gnu
rfm_job.out
rfm_job.out
```

(continues on next page)

```
[       OK ] (4/4) HelloMultiLangTest %lang=cpp /71bf65a3 @tresa:default+clang
[----------] all spawned checks have finished

[  PASSED  ] Ran 4/4 test case(s) from 2 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:01:00 2022
Run report saved in '/home/user/.reframe/reports/run-report-325.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-b22mnhb0.log'
```

Debugging sanity and performance patterns

When creating a new test that requires a complex output parsing for either the `sanity` or `performance` pipeline stages, tuning the functions decorated by `@sanity_function` or `@performance_function` may involve some trial and error to debug the complex regular expressions required. For lightweight tests which execute in a few seconds, this trial and error may not be an issue at all. However, when dealing with tests which take longer to run, this method can quickly become tedious and inefficient.

Tip: When dealing with `make`-based projects which take a long time to compile, you can use the command line option `--dont-restage` in order to speed up the compile stage in subsequent runs.

When a test fails, ReFrame will keep the test output in the stage directory after its execution, which means that one can load this output into a Python shell or another helper script without having to rerun the expensive test again. If the test is not failing but the user still wants to experiment or modify the existing sanity or performance functions, the command line option `--keep-stage-files` can be used when running ReFrame to avoid deleting the stage directory. With the executable's output available in the stage directory, one can simply use the re module to debug regular expressions as shown below.

```python
>>> import re

>>> # Read the test's output
>>> with open(the_output_file, 'r') as f:
...     test_output = ''.join(f.readlines())
...
>>> # Evaluate the regular expression
>>> re.find(the_regex_pattern, test_output)
```

Alternatively to using the re module, one could use all the `sanity` utility provided by ReFrame directly from the Python shell. In order to do so, if ReFrame was installed manually using the `bootstrap.sh` script, one will have to make all the Python modules from the `external` directory accessible to the Python shell as shown below.

```python
>>> import sys
>>> import os

>>> # Make the external modules available
>>> sys.path = [os.path.abspath('external')] + sys.path

>>> # Import ReFrame-provided sanity functions
>>> import reframe.utility.sanity as sn

>>> # Evaluate the regular expression
>>> assert sn.evaluate(sn.assert_found(the_regex_pattern, the_output_file))
```

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 -l -vv
```

```
Loading user configuration
Loading the generic configuration
Loading configuration file: ('tutorials/config/tresa.py',)
Detecting system using method: 'hostname'
Using standard hostname...
Retrieved hostname: 'host'
Looking for a matching configuration entry
Configuration found: picking system 'tresa'
Initializing runtime
Initializing system partition 'default'
Initializing system 'tresa'
Initializing modules system 'nomod'
detecting topology info for tresa:default
> found topology file '/home/user/.reframe/topology/tresa-default/processor.json';␣
→loading...
> device auto-detection is not supported
[ReFrame Environment]
  RFM_AUTODETECT_FQDN=<not set>
  RFM_AUTODETECT_METHOD=<not set>
  RFM_AUTODETECT_XTHOSTNAME=<not set>
  RFM_CHECK_SEARCH_PATH=<not set>
  RFM_CHECK_SEARCH_RECURSIVE=<not set>
  RFM_CLEAN_STAGEDIR=<not set>
  RFM_COLORIZE=n
  RFM_COMPRESS_REPORT=<not set>
  RFM_CONFIG_FILES=/home/user/Repositories/reframe/tutorials/config/tresa.py
  RFM_CONFIG_PATH=<not set>
  RFM_DUMP_PIPELINE_PROGRESS=<not set>
  RFM_GIT_TIMEOUT=<not set>
  RFM_HTTPJSON_URL=<not set>
  RFM_IGNORE_CHECK_CONFLICTS=<not set>
  RFM_IGNORE_REQNODENOTAVAIL=<not set>
  RFM_INSTALL_PREFIX=/home/user/Repositories/reframe
  RFM_KEEP_STAGE_FILES=<not set>
  RFM_MODULE_MAPPINGS=<not set>
  RFM_MODULE_MAP_FILE=<not set>
  RFM_NON_DEFAULT_CRAYPE=<not set>
  RFM_OUTPUT_DIR=<not set>
  RFM_PERFLOG_DIR=<not set>
  RFM_PIPELINE_TIMEOUT=<not set>
  RFM_PREFIX=<not set>
  RFM_PURGE_ENVIRONMENT=<not set>
  RFM_REMOTE_DETECT=<not set>
```

(continues on next page)

```
  RFM_REMOTE_WORKDIR=<not set>
  RFM_REPORT_FILE=<not set>
  RFM_REPORT_JUNIT=<not set>
  RFM_RESOLVE_MODULE_CONFLICTS=<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_TRAP_JOB_ERRORS=<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:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -C tutorials/config/tresa.py -c tutorials/basics/
→hello/hello2.py -l -vv'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', 'tutorials/config/tresa.py'
  check search path: '/home/user/Repositories/reframe/tutorials/basics/hello/hello2.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-3gcehyof.log'


Looking for tests in '/home/user/Repositories/reframe/tutorials/basics/hello/hello2.py'
Validating '/home/user/Repositories/reframe/tutorials/basics/hello/hello2.py': OK
  > Loaded 2 test(s)
Loaded 2 test(s)
Generated 4 test case(s)
Filtering test cases(s) by name: 4 remaining
Filtering test cases(s) by tags: 4 remaining
Filtering test cases(s) by other attributes: 4 remaining
Building and validating the full test DAG
Full test DAG:
  ('HelloMultiLangTest_1', 'tresa:default', 'gnu') -> []
  ('HelloMultiLangTest_1', 'tresa:default', 'clang') -> []
  ('HelloMultiLangTest_0', 'tresa:default', 'gnu') -> []
  ('HelloMultiLangTest_0', 'tresa:default', 'clang') -> []
Final number of test cases: 4
[List of matched checks]
- HelloMultiLangTest %lang=cpp /71bf65a3
- HelloMultiLangTest %lang=c /7cfa870e
Found 2 check(s)


Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-3gcehyof.log'
>>> profiler report [start] <<<
main: 0.053832 s
    test processing: 0.012268 s
        RegressionCheckLoader.load_all: 0.008720 s
            TestRegistry.instantiate_all: 0.003012 s
```

```
        generate_testcases: 0.000049 s
        main.<locals>._sort_testcases: 0.000012 s
        build_deps: 0.000072 s
        validate_deps: 0.000061 s
        toposort: 0.000091 s
        list_checks: 0.001080 s
>>> profiler report [ end ] <<<
```

You can see all the different phases ReFrame's frontend goes through when loading a test. 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',
             '-S strict_check=1',
             '--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.

Fig. 4: Complex test dependency graph. Nodes in red are set to fail.

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

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

```
[ReFrame Setup]
  version:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -c unittests/resources/checks_unlisted/deps_complex.
→py -r'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
→tresa.py'
  check search path: '/home/user/Repositories/reframe/unittests/resources/checks_
→unlisted/deps_complex.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-_008n_el.log'

[==========] Running 10 check(s)
[==========] Started on Sat Nov 12 19:01:00 2022

[----------] start processing checks
[ RUN      ] T0 /c9c2be9f @tresa:default+gnu
[ RUN      ] T0 /c9c2be9f @tresa:default+clang
[       OK ] ( 1/20) T0 /c9c2be9f @tresa:default+gnu
[       OK ] ( 2/20) T0 /c9c2be9f @tresa:default+clang
[ RUN      ] T4 /11ee5e9a @tresa:default+gnu
[ RUN      ] T4 /11ee5e9a @tresa:default+clang
[       OK ] ( 3/20) T4 /11ee5e9a @tresa:default+gnu
[       OK ] ( 4/20) T4 /11ee5e9a @tresa:default+clang
[ RUN      ] T5 /020d01e5 @tresa:default+gnu
[ RUN      ] T5 /020d01e5 @tresa:default+clang
[       OK ] ( 5/20) T5 /020d01e5 @tresa:default+gnu
[       OK ] ( 6/20) T5 /020d01e5 @tresa:default+clang
[ RUN      ] T1 /1f93603d @tresa:default+gnu
[ RUN      ] T1 /1f93603d @tresa:default+clang
```

(continues on next page)

```
[       OK ] (  7/20) T1 /1f93603d @tresa:default+gnu
[       OK ] (  8/20) T1 /1f93603d @tresa:default+clang
[ RUN      ] T8 /605fc1d6 @tresa:default+gnu
[     FAIL ] (  9/20) T8 /605fc1d6 @tresa:default+gnu
==> test failed during 'setup': test staged in '/home/user/Repositories/reframe/stage/
→tresa/default/gnu/T8'
[ RUN      ] T8 /605fc1d6 @tresa:default+clang
[     FAIL ] ( 10/20) T8 /605fc1d6 @tresa:default+clang
==> test failed during 'setup': test staged in '/home/user/Repositories/reframe/stage/
→tresa/default/clang/T8'
[     FAIL ] ( 11/20) T9 /78a78a4e @tresa:default+gnu
==> test failed during 'startup': test staged in None
[     FAIL ] ( 12/20) T9 /78a78a4e @tresa:default+clang
==> test failed during 'startup': test staged in None
[ RUN      ] T6 /6dbdaf93 @tresa:default+gnu
[ RUN      ] T6 /6dbdaf93 @tresa:default+clang
[       OK ] ( 13/20) T6 /6dbdaf93 @tresa:default+gnu
[       OK ] ( 14/20) T6 /6dbdaf93 @tresa:default+clang
[ RUN      ] T2 /0f617ba9 @tresa:default+gnu
[ RUN      ] T2 /0f617ba9 @tresa:default+clang
[ RUN      ] T3 /5dd67f7f @tresa:default+gnu
[ RUN      ] T3 /5dd67f7f @tresa:default+clang
[     FAIL ] ( 15/20) T2 /0f617ba9 @tresa:default+gnu
==> test failed during 'sanity': test staged in '/home/user/Repositories/reframe/stage/
→tresa/default/gnu/T2'
[     FAIL ] ( 16/20) T2 /0f617ba9 @tresa:default+clang
==> test failed during 'sanity': test staged in '/home/user/Repositories/reframe/stage/
→tresa/default/clang/T2'
[     FAIL ] ( 17/20) T7 /f005e93d @tresa:default+gnu
==> test failed during 'startup': test staged in None
[     FAIL ] ( 18/20) T7 /f005e93d @tresa:default+clang
==> test failed during 'startup': test staged in None
[       OK ] ( 19/20) T3 /5dd67f7f @tresa:default+gnu
[       OK ] ( 20/20) T3 /5dd67f7f @tresa:default+clang
[----------] all spawned checks have finished

[  FAILED  ] Ran 20/20 test case(s) from 10 check(s) (8 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:01:03 2022


================================================================================
SUMMARY OF FAILURES
--------------------------------------------------------------------------------
FAILURE INFO for T8
  * Expanded name: T8
  * Description:
  * System partition: tresa:default
  * Environment: gnu
  * Stage directory: /home/user/Repositories/reframe/stage/tresa/default/gnu/T8
  * Node list:
  * Job type: local (id=None)
  * Dependencies (conceptual): ['T1']
  * Dependencies (actual): [('T1', 'tresa:default', 'gnu')]
```

```
  * Maintainers: []
  * Failing phase: setup
  * Rerun with '-n /605fc1d6 -p gnu --system tresa:default -r'
  * Reason: exception
Traceback (most recent call last):
  File "/home/user/Repositories/reframe/reframe/frontend/executors/__init__.py", line
↪303, in _safe_call
    return fn(*args, **kwargs)
  File "/home/user/Repositories/reframe/reframe/core/hooks.py", line 101, in _fn
    getattr(obj, h.__name__)()
  File "/home/user/Repositories/reframe/reframe/core/hooks.py", line 32, in _fn
    func(*args, **kwargs)
  File "/home/user/Repositories/reframe/unittests/resources/checks_unlisted/deps_complex.
↪py", line 180, in fail
    raise Exception
Exception


-------------------------------------------------------------------------------
FAILURE INFO for T8
  * Expanded name: T8
  * Description:
  * System partition: tresa:default
  * Environment: clang
  * Stage directory: /home/user/Repositories/reframe/stage/tresa/default/clang/T8
  * Node list:
  * Job type: local (id=None)
  * Dependencies (conceptual): ['T1']
  * Dependencies (actual): [('T1', 'tresa:default', 'clang')]
  * Maintainers: []
  * Failing phase: setup
  * Rerun with '-n /605fc1d6 -p clang --system tresa:default -r'
  * Reason: exception
Traceback (most recent call last):
  File "/home/user/Repositories/reframe/reframe/frontend/executors/__init__.py", line
↪303, in _safe_call
    return fn(*args, **kwargs)
  File "/home/user/Repositories/reframe/reframe/core/hooks.py", line 101, in _fn
    getattr(obj, h.__name__)()
  File "/home/user/Repositories/reframe/reframe/core/hooks.py", line 32, in _fn
    func(*args, **kwargs)
  File "/home/user/Repositories/reframe/unittests/resources/checks_unlisted/deps_complex.
↪py", line 180, in fail
    raise Exception
Exception


-------------------------------------------------------------------------------
FAILURE INFO for T9
  * Expanded name: T9
  * Description:
  * System partition: tresa:default
  * Environment: gnu
  * Stage directory: None
```

```
  * Node list:
  * Job type: local (id=None)
  * Dependencies (conceptual): ['T8']
  * Dependencies (actual): [('T8', 'tresa:default', 'gnu')]
  * Maintainers: []
  * Failing phase: startup
  * Rerun with '-n /78a78a4e -p gnu --system tresa:default -r'
  * Reason: task dependency error: dependencies failed
--------------------------------------------------------------------------------
FAILURE INFO for T9
  * Expanded name: T9
  * Description:
  * System partition: tresa:default
  * Environment: clang
  * Stage directory: None
  * Node list:
  * Job type: local (id=None)
  * Dependencies (conceptual): ['T8']
  * Dependencies (actual): [('T8', 'tresa:default', 'clang')]
  * Maintainers: []
  * Failing phase: startup
  * Rerun with '-n /78a78a4e -p clang --system tresa:default -r'
  * Reason: task dependency error: dependencies failed
--------------------------------------------------------------------------------
FAILURE INFO for T2
  * Expanded name: T2
  * Description:
  * System partition: tresa:default
  * Environment: gnu
  * Stage directory: /home/user/Repositories/reframe/stage/tresa/default/gnu/T2
  * Node list: hostNone
  * Job type: local (id=59611)
  * Dependencies (conceptual): ['T6']
  * Dependencies (actual): [('T6', 'tresa:default', 'gnu')]
  * Maintainers: []
  * Failing phase: sanity
  * Rerun with '-n /0f617ba9 -p gnu --system tresa:default -r'
  * Reason: sanity error: 31 != 30
--------------------------------------------------------------------------------
FAILURE INFO for T2
  * Expanded name: T2
  * Description:
  * System partition: tresa:default
  * Environment: clang
  * Stage directory: /home/user/Repositories/reframe/stage/tresa/default/clang/T2
  * Node list: hostNone
  * Job type: local (id=59612)
  * Dependencies (conceptual): ['T6']
  * Dependencies (actual): [('T6', 'tresa:default', 'clang')]
  * Maintainers: []
  * Failing phase: sanity
  * Rerun with '-n /0f617ba9 -p clang --system tresa:default -r'
```

```
  * Reason: sanity error: 31 != 30
--------------------------------------------------------------------------------
FAILURE INFO for T7
  * Expanded name: T7
  * Description:
  * System partition: tresa:default
  * Environment: gnu
  * Stage directory: None
  * Node list:
  * Job type: local (id=None)
  * Dependencies (conceptual): ['T2']
  * Dependencies (actual): [('T2', 'tresa:default', 'gnu')]
  * Maintainers: []
  * Failing phase: startup
  * Rerun with '-n /f005e93d -p gnu --system tresa:default -r'
  * Reason: task dependency error: dependencies failed
--------------------------------------------------------------------------------
FAILURE INFO for T7
  * Expanded name: T7
  * Description:
  * System partition: tresa:default
  * Environment: clang
  * Stage directory: None
  * Node list:
  * Job type: local (id=None)
  * Dependencies (conceptual): ['T2']
  * Dependencies (actual): [('T2', 'tresa:default', 'clang')]
  * Maintainers: []
  * Failing phase: startup
  * Rerun with '-n /f005e93d -p clang --system tresa:default -r'
  * Reason: task dependency error: dependencies failed
--------------------------------------------------------------------------------
Run report saved in '/home/user/.reframe/reports/run-report-326.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-_008n_el.log'
```

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:

```
[ReFrame Setup]
  version:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe --restore-session --keep-stage-files -n T6 -r'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
↪tresa.py'
  check search path: '/home/user/Repositories/reframe/unittests/resources/checks_
↪unlisted/deps_complex.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-vtnok1ih.log'


[==========] Running 1 check(s)
[==========] Started on Sat Nov 12 19:01:06 2022

[----------] start processing checks
[ RUN      ] T6 /6dbdaf93 @tresa:default+gnu
[ RUN      ] T6 /6dbdaf93 @tresa:default+clang
[       OK ] (1/2) T6 /6dbdaf93 @tresa:default+gnu
[       OK ] (2/2) T6 /6dbdaf93 @tresa:default+clang
[----------] all spawned checks have finished


[  PASSED  ] Ran 2/2 test case(s) from 1 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:01:07 2022
Run report saved in '/home/user/.reframe/reports/run-report-328.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-vtnok1ih.log'
```

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
```

```
[ReFrame Setup]
  version:           4.0.0-dev.2+5ea6b7a6
  command:           './bin/reframe -c unittests/resources/checks_unlisted/deps_complex.
↪py -n T6 -r'
  launched by:       user@host
  working directory: '/home/user/Repositories/reframe'
  settings files:    '<builtin>', '/home/user/Repositories/reframe/tutorials/config/
↪tresa.py'
  check search path: '/home/user/Repositories/reframe/unittests/resources/checks_
↪unlisted/deps_complex.py'
  stage directory:   '/home/user/Repositories/reframe/stage'
  output directory:  '/home/user/Repositories/reframe/output'
  log files:         '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-8n8uvclh.log'


[==========] Running 5 check(s)
[==========] Started on Sat Nov 12 19:01:07 2022
```

(continues on next page)

```
[----------] start processing checks
[ RUN      ] T0 /c9c2be9f @tresa:default+gnu
[ RUN      ] T0 /c9c2be9f @tresa:default+clang
[       OK ] ( 1/10) T0 /c9c2be9f @tresa:default+gnu
[       OK ] ( 2/10) T0 /c9c2be9f @tresa:default+clang
[ RUN      ] T4 /11ee5e9a @tresa:default+gnu
[ RUN      ] T4 /11ee5e9a @tresa:default+clang
[       OK ] ( 3/10) T4 /11ee5e9a @tresa:default+gnu
[       OK ] ( 4/10) T4 /11ee5e9a @tresa:default+clang
[ RUN      ] T5 /020d01e5 @tresa:default+gnu
[ RUN      ] T5 /020d01e5 @tresa:default+clang
[       OK ] ( 5/10) T5 /020d01e5 @tresa:default+gnu
[       OK ] ( 6/10) T5 /020d01e5 @tresa:default+clang
[ RUN      ] T1 /1f93603d @tresa:default+gnu
[ RUN      ] T1 /1f93603d @tresa:default+clang
[       OK ] ( 7/10) T1 /1f93603d @tresa:default+gnu
[       OK ] ( 8/10) T1 /1f93603d @tresa:default+clang
[ RUN      ] T6 /6dbdaf93 @tresa:default+gnu
[ RUN      ] T6 /6dbdaf93 @tresa:default+clang
[       OK ] ( 9/10) T6 /6dbdaf93 @tresa:default+gnu
[       OK ] (10/10) T6 /6dbdaf93 @tresa:default+clang
[----------] all spawned checks have finished

[  PASSED  ] Ran 10/10 test case(s) from 5 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Sat Nov 12 19:01:08 2022
Run report saved in '/home/user/.reframe/reports/run-report-329.json'
Log file(s) saved in '/var/folders/h7/k7cgrdl13r996m4dmsvjq7v80000gp/T/rfm-8n8uvclh.log'
```

Implementing test workarounds efficiently

New in version 3.2.

Sometimes you may need to add a quick workaround in a test, because something in a system or an environment broken. The best way to implement this is through hooks, because you can easily disable any hook from the command-line and you don't need to update the test every time you want to check if the system is fixed and the workaround is not needed anymore.

Let's use one example from the previous tutorial and let's assume that there is something wrong with one of the environments and a special macro needs to be defined in order for the compilation to succeed. Instead of adding another flag in the `set_compilation_flags()` hook, it is better to add another hook containing just the workaround as shown below:

```python
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'
```

```python
    build_system = 'SingleSource'
    executable_opts = ['16']

    @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']

    @sanity_function
    def assert_num_messages(self):
        num_messages = sn.len(sn.findall(r'\[\s?\d+\] Hello, World\!',
                                         self.stdout))
        return sn.assert_eq(num_messages, 16)

    @run_before('compile')
    def fooenv_workaround(self):
        ce = self.current_environ.name
        if ce == 'foo':
            self.build_system.cppflags += [
                '-D__GCC_ATOMIC_TEST_AND_SET_TRUEVAL'
            ]
```

This way the test will start passing again allowing us to catch any new issues while waiting for the original issue to be fixed. Then we can run the test anytime using `--disable-hook=fooenv_workaround` to check if the workaround is not needed anymore.

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:

```yaml
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 in a child pipeline. The generated ReFrame command that will run each individual test reuses the `-C`, `-R`, `-v` and `--mode` options passed to the initial invocation of ReFrame that was used to generate the pipeline. Users can define CI-specific execution modes in their configuration in order to pass arbitrary options to the ReFrame invocation in the child pipeline.

Finally, we pass the generated CI pipeline file to second phase as an artifact and we are done! If `image` keyword is defined in `.gitlab-ci.yml`, the emitted pipeline will use the same image as the one defined in the parent pipeline. Besides, each job in the generated pipeline will output a separate junit report which can be used to create GitLab badges.

The following figure shows one part of the automatically generated pipeline for the test graph depicted *above*.
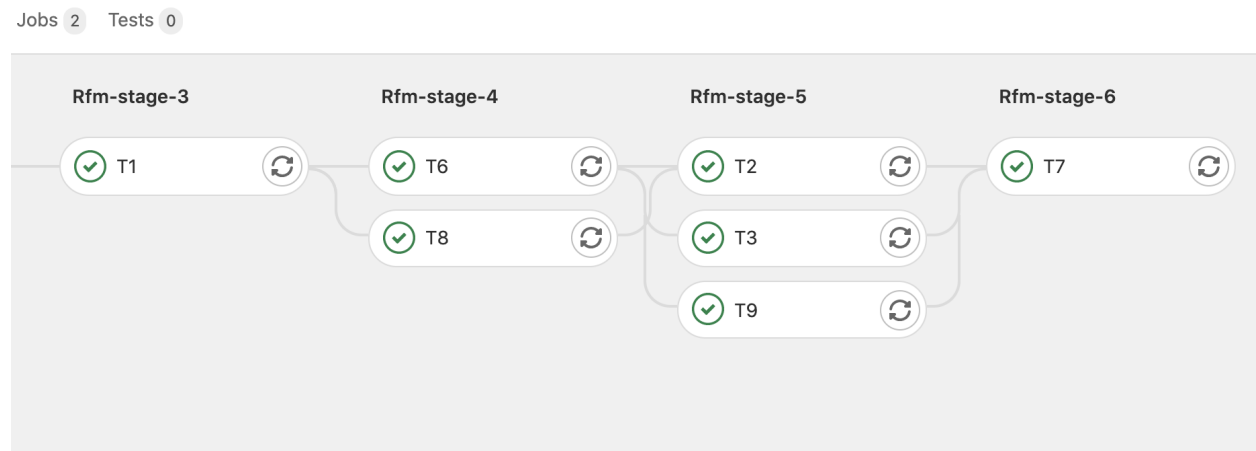
Fig. 5: Snapshot of a Gitlab pipeline generated automatically by ReFrame.

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

2.3.7 Tutorial 7: The Flux Framework Scheduler

This is a tutorial that will show how to use refame with Flux Framework. First, build the container here from the root of reframe.

```
$ docker build -f tutorials/flux/Dockerfile -t flux-reframe .
```

Then shell inside, optionally binding the present working directory if you want to develop.

```
$ docker run -it -v $PWD:/code flux-reframe
$ docker run -it flux-reframe
```

Note that if you build the local repository, you'll need to bootstrap and install again, as we have over-written the bin!

```
./bootstrap.sh
```

And then reframe will again be in the local `bin` directory:

```
# which reframe
/code/bin/reframe
```

Then we can run ReFrame with the custom config config.py for flux.

```
# What tests are under tutorials/flux?
$ cd tutorials/flux
$ reframe -c . -C settings.py -l
```

```
[ReFrame Setup]
  version:           4.0.0-dev.1
  command:           '/code/bin/reframe -c tutorials/flux -C tutorials/flux/settings.py -
↪l'
  launched by:       root@b1f6650222bc
  working directory: '/code'
  settings file:     'tutorials/flux/settings.py'
  check search path: '/code/tutorials/flux'
  stage directory:   '/code/stage'
  output directory:  '/code/output'

[List of matched checks]
- EchoRandTest /66b93401
Found 1 check(s)

Log file(s) saved in '/tmp/rfm-ilqg7fqg.log'
```

This also works

```
$ reframe -c tutorials/flux -C tutorials/flux/settings.py -l
```

And then to run tests, just replace `-l` (for list) with `-r` or `--run` (for run):

```
$ reframe -c tutorials/flux -C tutorials/flux/settings.py --run
```

```
root@b1f6650222bc:/code# reframe -c tutorials/flux -C tutorials/flux/settings.py --run
[ReFrame Setup]
```

```
  version:          4.0.0-dev.1
  command:          '/code/bin/reframe -c tutorials/flux -C tutorials/flux/settings.py -
→-run'
  launched by:      root@b1f6650222bc
  working directory: '/code'
  settings file:    'tutorials/flux/settings.py'
  check search path: '/code/tutorials/flux'
  stage directory:  '/code/stage'
  output directory: '/code/output'

[==========] Running 1 check(s)
[==========] Started on Fri Sep 16 20:47:15 2022

[----------] start processing checks
[ RUN      ] EchoRandTest /66b93401 @generic:default+builtin
[       OK ] (1/1) EchoRandTest /66b93401 @generic:default+builtin
[----------] all spawned checks have finished

[  PASSED  ] Ran 1/1 test case(s) from 1 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Fri Sep 16 20:47:15 2022
Run report saved in '/root/.reframe/reports/run-report.json'
Log file(s) saved in '/tmp/rfm-0avso9nb.log'
```

For advanced users or developers, here is how to run tests within the container:

Testing

```
./test_reframe.py --rfm-user-config=tutorials/flux/settings.py unittests/test_schedulers.
→py -xs
```

2.3.8 Online Tutorials

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

2.4 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.

ReFrame's configuration can be either in JSON or in Python format and can be split into multiple files. The Python 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 single Python-based configuration file also for historical reasons, since it was the only way to configure ReFrame in versions prior to 3.0.

Changed in version 4.0.0: The configuration can now be split into multiple files.

2.4.1 Loading the configuration

ReFrame builds its final configuration gradually by combining multiple configuration files. Each one can have different parts of the configuration, for example different systems, different environments, different general options or different logging handlers. This technique allows users to avoid having a single huge configuration file.

The first configuration file loaded in this chain is always the generic builtin configuration located under `${RFM_INSTALL_PREFIX}/reframe/core/settings.py`. This contains everything that ReFrame needs to run on a generic system, as well as basic settings for logging, so subsequent configuration files may skip defining some configuration sections altogether, if they are not relevant.

ReFrame continues on looking for configuration files in the directories defined in *RFM_CONFIG_PATH*. For each directory, will look within it for a `settings.py` or `settings.json` file (in that order), and if it finds one, it will load it.

Finally, ReFrame processes the *--config-file* option or the *RFM_CONFIG_FILES* environment variable to load any specific configuration files passed from the command line.

2.4.2 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 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': 'daint',
            'descr': 'Piz Daint Supercomputer',
            'hostnames': ['daint'],
            'modules_system': 'tmod32',
            'partitions': [
                {
                    'name': 'login',
                    'descr': 'Login nodes',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['builtin', 'gnu', 'intel', 'nvidia', 'cray'],
                },
                {
                    'name': 'gpu',
                    'descr': 'Hybrid nodes',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['-C gpu', '-A csstaff'],
                    'environs': ['gnu', 'intel', 'nvidia', 'cray'],
                    'max_jobs': 100,
                },
                {
```

(continues on next page)

```
                'name': 'mc',
                'descr': 'Multicore nodes',
                'scheduler': 'slurm',
                'launcher': 'srun',
                'access': ['-C mc', '-A csstaff'],
                'environs': ['gnu', 'intel', 'nvidia', 'cray'],
                'max_jobs': 100,
                'resources': [
                    {
                        'name': 'memory',
                        'options': ['--mem={size}']
                    }
                ]
            }
        ]
    }
],
'environments': [
    {
        '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': 'nvidia',
        'modules': ['PrgEnv-nvidia'],
        'cc': 'cc',
        'cxx': 'CC',
        'ftn': 'ftn',
        'target_systems': ['daint']
    },
    {
        'name': 'builtin',
```

```
        'cc': 'cc',
        'cxx': 'CC',
        'ftn': 'ftn',
        'target_systems': ['daint']
    }
] # end of environments
}
```

There are three required sections that the final ReFrame configuration must have: `systems`, `environments` and `logging`, but in most cases you will define only the first two, as ReFrame's builtin configuration already defines a reasonable logging configuration. We will first cover these sections and then move on to the optional ones.

Tip: These configuration sections may not all be defined in the same configuration file, but can reside in any configuration file that is being loaded. This is the case of the example configuration shown above, where the `logging` section is "missing" as it's defined in ReFrame's builtin configuration.

Systems Configuration

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

```
'systems': [
    {
        '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', 'nvidia', 'cray'],
            },
            {
                'name': 'gpu',
                'descr': 'Hybrid nodes',
                'scheduler': 'slurm',
                'launcher': 'srun',
                'access': ['-C gpu', '-A csstaff'],
                'environs': ['gnu', 'intel', 'nvidia', 'cray'],
                'max_jobs': 100,
            },
            {
                'name': 'mc',
                'descr': 'Multicore nodes',
                'scheduler': 'slurm',
                'launcher': 'srun',
                'access': ['-C mc', '-A csstaff'],
```

```
                    'environs': ['gnu', 'intel', 'nvidia', 'cray'],
                    'max_jobs': 100,
                    'resources': [
                        {
                            'name': 'memory',
                            'options': ['--mem={size}']
                        }
                    ]
                }
            ]
        }
    ],
```

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 environment 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', 'nvidia', 'cray'],
                    'max_jobs': 100,
```

The basic properties of a partition are the following:

- **name**: The name of the partition. This should be an alphanumeric string (dashes - are allowed) and it will be used to refer to this partition in other contexts.

- **descr**: A detailed description of the system partition.

- **scheduler**: The workload manager (job scheduler) used in this partition for launching parallel jobs. In this particular example, the Slurm scheduler is used. For a complete list of the supported job schedulers, see here.

- **launcher**: The parallel job launcher used in this partition. In this case, the **srun** command will be used. For a complete list of the supported parallel job launchers, see here.

- **access**: A list of scheduler options that will be passed to the generated job script for gaining access to that logical partition. Notice how in this case, the nodes are selected through a constraint and not an actual scheduler partition.

- **environs**: The list of environments that ReFrame will use to run regression tests on this partition. These are just symbolic names that refer to environments defined in the **environments** section described below.

- **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.

For more partition configuration options, 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',
        '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': 'nvidia',
        'modules': ['PrgEnv-nvidia'],
        'cc': 'cc',
        'cxx': 'CC',
```

```
        'ftn': 'ftn',
        'target_systems': ['daint']
    },
    {
        'name': 'builtin',
        'cc': 'cc',
        'cxx': 'CC',
        'ftn': 'ftn',
        'target_systems': ['daint']
    }
] # end of environments
```

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 the builtin configuration:

```
'logging': [
    {
        'handlers$': [
            {
                'type': 'stream',
                'name': 'stdout',
                'level': 'info',
                'format': '%(message)s'
            },
        ],
        'handlers': [
            {
                'type': 'file',
                'level': 'debug2',
                '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,%(version)s,'
                    '%(check_display_name)s,%(check_system)s,'
                    '%(check_partition)s,%(check_environ)s,'
                    '%(check_jobid)s,%(check_result)s,%(check_perfvalues)s'
                ),
                'format_perfvars': (
                    '%(check_perf_value)s,%(check_perf_unit)s,'
                    '%(check_perf_ref)s,%(check_perf_lower)s,'
                    '%(check_perf_upper)s,'
                ),
                'append': True
            }
        ]
    }
]   # end of logging
```

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 its logging mechanism and that's why there is the special `handlers$` property. The handler defined in this property, in the builtin configuration shown here, defines how exactly the output of ReFrame will be printed. You will not have to override this in your configuration files, unless you really need to change how ReFrame's output look like.

As a user you might need to override the `handlers` property to define different sinks for ReFrame logs and/or output using different verbosity levels. Note that you can use multiple handlers at the same time. 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_variables` or the `perf_patterns` attributes. 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` and `format_perfvars` properties, such that it can be easily parsed from post processing tools. Apart from file logging, ReFrame offers more advanced performance logging capabilities through Syslog, Graylog and HTTP.

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 is finally one additional optional configuration section that is not discussed here:

The `modes` section defines different execution modes for the framework. Execution modes are discussed in the *How ReFrame Executes Tests* page.

2.4.3 Building the Final Configuration

New in version 4.0.0.

As mentioned above ReFrame can build its final configuration incrementally from a series of user-specified configuration files starting from the basic builtin configuration. We discussed briefly at the beginning of this page how ReFrame locates and loads these configuration files and the documentation of the `-C` option provides more detailed information. But how are these configuration files actually combined? This is what we will discuss in this section.

Configuration objects in the top-level configuration sections can be split in two categories: *named* and *unnamed*. Named objects are the systems, the environments and the modes and the rest are unnamed. The named object have a `name` property. When ReFrame builds its final configuration, named objects from newer configuration files are either appended or prepended in their respective sections, but unnamed objects are merged based on their `target_systems`. More specifically, new systems are *prepended* in the list of the already defined, whereas environments and modes are *appended*. The reason for that is that systems are tried from the beginning of the list until a match is found. See *Picking the Right System Configuration* for more information on how ReFrame picks the right system. If a system is redefined, ReFrame will warn about it, but it will still use the new definition. This is done for backward compatibility with the old configuration mechanism, where users had to redefine also the builtin systems and environments in their configuration. Similarly, if an environment or a mode is redefined, ReFrame will issue a warning, but only if the redefinition is at the same scope as the conflicting one. Again this is done for backward compatibility.

Given the Piz Daint configuration shown in this section and the ReFrame's builtin configuration, ReFrame will build internally the following configuration:

```
site_configuration = {
    'systems': [
        {
            # from the Daint config
            'name': 'daint',
            ...
        },
        {
            # from the builtin config
            'name': 'generic',
            ...
        }
    ],
    'environments': [
        {
            # from the builtin config
            'name': 'builtin'
             ...
        },
        {
            # from the Daint config
            'name': 'gnu',
            ...
        }
    ],
    'logging': [
        # from the builtin config
    ]
}
```

You might wonder why would I need to define multiple objects in sections such as `logging` or `general`. As mentioned above, ReFrame merges them if they refer to the same target systems, but if they don't they can serve as scopes for the configuration parameters they define. Imagine the following `general` section:

```
'general': [
    {
        'git_timeout': 5
    },
    {
        'git_timeout': 10,
        'target_systems': ['daint']
    },
    {
        'git_timeout': 20,
        'target_systems': ['tresa']
    }
]
```

This means that the default value for `git_timeout` is 5 seconds for any system, but it is 10 for `daint` and 20 for `tresa`. The nice thing is that you can spread that in multiple configuration files and ReFrame will combine them internally in a single one with the various configuration options indexed by their scope.

2.4.4 Picking the Right 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.

ReFrame uses an auto-detection mechanism to get information about the host it is running on and uses that information to pick the right system configuration. Currently, only one auto-detection method is supported that retrieves the hostname. Based on this, 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.

The auto-detection process can be controlled through the *RFM_AUTODETECT_METHOD*, *RFM_AUTODETECT_FQDN* and *RFM_AUTODETECT_XTHOSTNAME* environment variables.

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']
    },
    {
        '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`.

2.4.5 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",
      "nvidia",
      "cray"
    ],
    "max_jobs": 10
  },
  {
    "name": "gpu",
    "descr": "Hybrid nodes",
    "scheduler": "slurm",
    "launcher": "srun",
    "access": [
      "-C gpu",
      "-A csstaff"
    ],
    "environs": [
      "gnu",
      "intel",
      "nvidia",
      "cray"
    ],
    "max_jobs": 100
  },
  {
    "name": "mc",
    "descr": "Multicore nodes",
    "scheduler": "slurm",
    "launcher": "srun",
    "access": [
      "-C mc",
      "-A csstaff"
    ],
    "environs": [
      "gnu",
      "intel",
      "nvidia",
      "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",
      "nvidia",
      "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.

2.4.6 Auto-detecting processor information

New in version 3.7.0.

ReFrame is able to detect the processor topology of both local and remote partitions automatically. The processor and device information are made available to the tests through the corresponding attributes of the `current_partition` allowing a test to modify its behavior accordingly. Currently, ReFrame supports auto-detection of the local or remote processor information only. It does not support auto-detection of devices, in which cases users should explicitly specify this information using the `devices` configuration option. The processor information auto-detection works as follows:

1. If the `processor` configuration option is defined, then no auto-detection is attempted.

2. If the `processor` configuration option is not defined, ReFrame will look for a processor configuration meta-data file in `~/.reframe/topology/{system}-{part}/processor.json`. If the file is found, the topology information is loaded from there. These files are generated automatically by ReFrame from previous runs.

3. If the corresponding metadata files are not found, the processor information will be auto-detected. If the system partition is local (i.e., `local` scheduler + `local` launcher), the processor information is auto-detected unconditionally and stored in the corresponding metadata file for this partition. If the partition is remote, ReFrame will not try to auto-detect it unless the `RFM_REMOTE_DETECT` or the `remote_detect` configuration option is set. In that case, the steps to auto-detect the remote processor information are the following:

    a. ReFrame creates a fresh clone of itself in a temporary directory created under `.` by default. This temporary directory prefix can be changed by setting the `RFM_REMOTE_WORKDIR` environment variable.

    b. ReFrame changes to that directory and launches a job that will first bootstrap the fresh clone and then run that clone with `{launcher} ./bin/reframe --detect-host-topology=topo.json`. The `--detect-host-topology` option causes ReFrame to detect the topology of the current host, which in this case would be the remote compute nodes.

    In case of errors during auto-detection, ReFrame will simply issue a warning and continue.

2.5 Advanced Topics

2.5.1 How ReFrame Executes Tests

A ReFrame test will be normally tried for different programming environments and different partitions within the same ReFrame run. These can be defined in the test's class body, in a post-init hook or in its `__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. 6: 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. 7: 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 Compile Phase

During this phase a job script for the compilation of the test will be created and it will be submitted for execution. 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 execution policy.

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 build and run phases, since the execution blocks until the associated test job finishes.

Fig. 8: The serial execution policy.

In the asynchronous execution policy, multiple tests can be simultaneously on-the-fly. When a test enters the build or 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. 9: 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.

ReFrame's runtime internally encapsulates each test in a task, which is scheduled for execution. This task can be in different states and is responsible for executing the test's pipeline. The following state diagram shows how test tasks are scheduled, as well as when the various test pipeline stages are executed.

Fig. 10: State diagram of the execution of test tasks with annotations for the execution of the actual pipeline stages.

There are a number of things to notice in this diagram:

- If a test encounters an exception it is marked as a failure. Even normal failures, such as dependency failures and sanity or performance failures are also exceptions raised explicitly by the framework during a pipeline stage.

- The pipeline stages that are executed asynchronously, namely the `compile` and `run` stages, are split in sub-stages for submitting the corresponding job and for checking or waiting its completion. This is why in ReFrame error messages you may see `compile_complete` or `run_complete` being reported as the failing stage.

- The execution of a test may be stalled if there are not enough execution slots available for submitting compile or run jobs on the target partition.

- Although a test is officially marked as "completed" only when its cleanup phase is executed, it is reported as success or failure as soon as it is "retired," i.e., as soon as its performance stage has passed successfully.

- For successful tests, the `cleanup` stage is executed *after* the test is reported as a "success," since a test may not clean up its resources until all of its immediate dependencies finish also successfully. If the `cleanup` phase fails, the test is not marked as a failure, but this condition is marked as an error.

Changed in version 3.10.0: The `compile` stage is now also executed asynchronously.

Where each pipeline stage is executed?

There are two executions contexts where a pipeline stage can be executed: the ReFrame execution context and the partition execution context. The *ReFrame execution context* is where ReFrame executes. This is always the local host. The *partition execution context* can either be local or remote depending on how the partition is configured. The following table show in which context each pipeline stage executes:

Pipeline Stage	Execution Context
Setup	ReFrame
Compile	ReFrame if `build_locally` or `local` is `True`, partition otherwise.
Run	ReFrame if `local` is `True`, partition otherwise.
Sanity	ReFrame
Performance	ReFrame
Cleanup	ReFrame

It should be noted that even if the partition execution context is local, it is treated differently from the ReFrame execution context. For example, a test executing in the ReFrame context will not respect the `max_jobs` partition configuration option, even if the partition is local. To control the concurrency of the ReFrame execution context, users should set the `max_local_jobs` option instead.

Changed in version 3.10.0: Execution contexts were formalized.

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.

2.5.2 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:

```python
@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']

    @run_after('init')
    def setup_deps(self):
        self.depends_on('T0')
```

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

Fig. 11: 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. 12: 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:

```python
import reframe.utility.udeps as udeps


@rfm.simple_test
class T1(rfm.RegressionTest):
    ...
    @run_after('init')
    def setup_deps(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:

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.

Fig. 13: Fully dependent test cases.

Split by partition

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

Fig. 14: 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. 15: 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.

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.

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`:

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


@rfm.simple_test
class T1(rfm.RegressionTest):
```

(continues on next page)

Fig. 16: Test case dependencies partitioned by exclusive partition.

Fig. 17: Test case dependencies partitioned by exclusive environment.

(continued from previous page)

```
    ...
    @run_after('init')
    def setup_deps(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.

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:

```
    @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']
```

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

Fig. 18: Test case dependencies partitioned by exclusive case.

Fig. 19: Custom test case dependencies.

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:

```python
@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.

2.5.3 Understanding the Mechanism of Deferrable Functions

This section describes the mechanism behind deferrable functions, which in ReFrame, they are used for sanity and performance checking. Generally, writing a new sanity function in a *RegressionTest* is as straightforward as decorating a simple member function with the built-in `sanity_function()` decorator. Behind the scenes, this decorator will convert the Python function into a deferrable function and schedule its evaluation for the sanity stage of the test. However, when dealing with more complex scenarios such as a deferrable function taking as an argument the results from other deferrable functions, it is crucial to understand how a deferrable function differs from a regular Python function, and when is it actually evaluated.

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 `deferrable()` when decorating a member function of a class derived from `RegressionMixin`, or alternatively, the `reframe.utility.sanity.deferrable()` decorator can be used for any other function. The example below demonstrates a simple scenario:

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

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

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

```python
>>> 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:

```python
>>> 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:

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

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

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

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

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

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

```python
>>> 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.deferrable
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,)
```

(continues on next page)

```
>>> 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.deferrable
... 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 deferrable() 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 `@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.

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

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

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

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

```python
>>> 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. 20: 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.deferrable
... 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:

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

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

@sn.deferrable
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.

Ready to Go Deferrable Functions

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

Deferrable functions vs 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:

```python
>>> @sn.deferrable
... 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
```

(continues on next page)

```
>>> expr
<reframe.core.deferrable._DeferredExpression object at 0x2abc630abc18>
```

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

- Generators are iterator objects, while deferred expressions are not. As a result, you can trigger the evaluation of a generator expression using the `next` builtin function. For a deferred expression you should use *evaluate* instead.

- A generator object is iterable, whereas a deferrable object will be iterable if and only if the result of its evaluation is iterable.

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

Here is an example demonstrating this difference:

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

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

2.6 ReFrame Manuals

2.6.1 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.

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.

--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.

`--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.

`--maintainer``=MAINTAINER`

> Filter tests by maintainer.
>
> `MAINTAINER` is interpreted as a [Python Regular Expression](#); all tests that have at least a matching maintainer will be selected. `MAINTAINER` being a regular expression has the implication that `--maintainer 'foo'` will select also tests that define `'foobar'` as a maintainer. To restrict the selection to tests defining only `'foo'`, you should use `--maintainer 'foo$'`.
>
> This option may be specified multiple times, in which case only tests defining or matching *all* maintainers will be selected.
>
> New in version 3.9.1.
>
> Changed in version 4.1.0: The `MAINTAINER` pattern is matched anywhere in the maintainer's name and not at its beginning. If you want to match at the beginning of the name, you should prepend `^`.

`-n, --name``=NAME`

> Filter tests by name.
>
> `NAME` is interpreted as a [Python Regular Expression](#); any test whose *display name* matches `NAME` will be selected. The display name of a test encodes also any parameterization information. See *Test Naming Scheme* for more details on how the tests are automatically named by the framework.
>
> Before matching, any whitespace will be removed from the display name of the test.
>
> 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'`.
>
> If the special notation `<test_name>@<variant_num>` is passed as the `NAME` argument, then an exact match will be performed selecting the variant `variant_num` of the test `test_name`.
>
> You may also select a test by its hash code using the notation `/<test-hash>` for the `NAME` argument.
>
> ---
>
> **Note:** Fixtures cannot be selected.
>
> ---
>
> Changed in version 3.10.0: The option's behaviour was adapted and extended in order to work with the updated test naming scheme.
>
> Changed in version 4.0.0: Support selecting tests by their hash code.
>
> Changed in version 4.1.0: The `NAME` pattern is matched anywhere in the test name and not at its beginning. If you want to match at the beginning of a test name, you should prepend `^`.

`-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.

--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.

--skip-system-check

Do not filter tests against the selected system.

-T, --exclude-tag=TAG

Exclude tests by tags.

TAG is interpreted as a [Python Regular Expression](); any test with tags matching TAG will be excluded.

This option may be specified multiple times, in which case tests with *any* of the specified tags will be excluded: -T TAG1 -T TAG2 is therefore equivalent to -T 'TAG1|TAG2'.

Changed in version 4.1.0: The TAG pattern is matched anywhere in the tag name and not at its beginning. If you want to match at the beginning of a tag, you should prepend ^.

-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.

Changed in version 4.1.0: The TAG pattern is matched anywhere in the tag name and not at its beginning. If you want to match at the beginning of a tag, you should prepend ^.

-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'.

Changed in version 4.1.0: The NAME pattern is matched anywhere in the test name and not at its beginning. If you want to match at the beginning of a test name, you should prepend ^.

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.

--describe

Print a detailed description of the *selected tests* in JSON format and exit.

Note: The generated test description corresponds to its state after it has been initialized. If any of its attributes are changed or set during its execution, their updated values will not be shown by this listing.

New in version 3.10.0.

`--dry-run`

Dry run the selected tests.

The dry-run mode will try to execute as much of the test pipeline as possible. More specifically, the tests will not be submitted and will not be run for real, but their stage directory will be prepared and the corresponding job script will be emitted. Similarly, the sanity and performance functions will not be evaluated but all the preparation will happen. Tests run in dry-run mode will not fail unless there is a programming error in the test or if the test tries to use a resource that is not produced in dry run mode (e.g., access the standard output or a resource produced by a dependency outside any sanity or performance function). In this case, users can call the `is_dry_run()` method in their test and take a specific action if the test is run in dry-run mode.

New in version 4.1.

`-L, --list-detailed`[=T|C]

List selected tests providing more details for each test.

The unique id of each test (see also `unique_name`) as well as the file where each test is defined are printed.

This option accepts optionally a single argument denoting what type of listing is requested. Please refer to `-l` for an explanation of this argument.

New in version 3.10.0: Support for different types of listing is added.

Changed in version 4.0.5: The variable names to which fixtures are bound are also listed. See *Test Naming Scheme* for more information.

`-l, --list`[=T|C]

List selected tests and their dependencies.

This option accepts optionally a single argument denoting what type of listing is requested. There are two types of possible listings:

- *Regular test listing* (T, the default): This type of listing lists the tests and their dependencies or fixtures using their `display_name`. A test that is listed as a dependency of another test will not be listed separately.

- *Concretized test case listing* (C): This type of listing lists the exact test cases and their dependencies as they have been concretized for the current system and environment combinations. This listing shows practically the exact test DAG that will be executed.

New in version 3.10.0: Support for different types of listing is added.

Changed in version 4.0.5: The variable names to which fixtures are bound are also listed. See *Test Naming Scheme* for more information.

`--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, the precedence order is the following:

```
--describe > --list-detailed > --list > --list-tags > --ci-generate
```

Options controlling ReFrame output

--compress-report

Compress the generated run report (see `--report-file`). The generated report is a JSON file formatted in a human readable form. If this option is enabled, the generated JSON file will be a single stream of text without additional spaces or new lines.

This option can also be set using the `RFM_COMPRESS_REPORT` environment variable or the `compress_report` general configuration parameter.

New in version 3.12.0.

--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.

--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.

-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.

--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.

--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.

--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 where each retry is treated as an individual testsuite.
>
> 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.
>
> Changed in version 3.6.1: Added support for retries in the JUnit XML report.

-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.

--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.

--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.

Options controlling ReFrame execution

--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.

--distribute[=NODESTATE]

> Distribute the selected tests on all the nodes in state NODESTATE in their respective valid partitions.
>
> ReFrame will parameterize and run the tests on the selected nodes. Effectively, it will dynamically create new tests that inherit from the original tests and add a new parameter named $nid which contains the list of nodes that the test must run on. The new tests are named with the following pattern {orig_test_basename}_{partition_fullname}.
>
> When determining the list of nodes to distribute the selected tests, ReFrame will take into account any job options passed through the -J option.
>
> You can optionally specify the state of the nodes to consider when distributing the test through the NODESTATE argument:
>
> - all: Tests will run on all the nodes of their respective valid partitions regardless of the nodes' state.
> - idle: Tests will run on all *idle* nodes of their respective valid partitions.
> - NODESTATE: Tests will run on all the nodes in state NODESTATE of their respective valid partitions. If NODESTATE is not specified, idle will be assumed.
>
> The state of the nodes will be determined once, before beginning the execution of the tests, so it might be different at the time the tests are actually submitted.
>
> ---
>
> **Note:** Currently, only single-node jobs can be distributed and only local or the Slurm-based backends support this feature.
>
> ---
>
> ---
>
> **Note:** Distributing tests with dependencies is not supported, but you can distribute tests that use fixtures.
>
> ---
>
> New in version 3.11.0.

--exec-order=ORDER

> Impose an execution order for the independent tests. The ORDER argument can take one of the following values:
>
> - name: Order tests by their display name.
> - rname: Order tests by their display name in reverse order.
> - uid: Order tests by their unique name.
> - ruid: Order tests by their unique name in reverse order.
> - random: Randomize the order of execution.
>
> If this option is not specified the order of execution of independent tests is implementation defined. This option can be combined with any of the listing options (-l or -L) to list the tests in the order.
>
> New in version 4.0.0.

--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 build and run phases 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 build or run phase have 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 build or 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.

--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.

--mode=MODE

> ReFrame execution mode to use.
>
> An execution mode is simply a predefined set of options that is set in the *modes configuration parameter*. Additional options can be passed to the command line, in which case they will be combined with the options defined in the selected execution mode. More specifically, any additional ReFrame options will be *appended* to the command line options of the selected mode. As a result, if a normal option is specified both inside the execution mode and the in the command line, the command line option will take precedence. On the other hand, if an option that is allowed to be specified multiple times, e.g., the `-S` option, is passed both inside the execution mode and in the command line, their values will be combined. For example, if the execution mode `foo` defines `-S modules=foo`, the invocation `--mode=foo -S num_tasks=10` is the equivalent of `-S modules=foo -S num_tasks=10`.
>
> Changed in version 4.1: Options that can be specified multiple times are now combined between execution modes and the command line.

--repeat=N

> Repeat the selected tests N times. This option can be used in conjunction with the `--distribute` option in which case the selected tests will be repeated multiple times and distributed on individual nodes of the system's partitions.

> **Note:** Repeating tests with dependencies is not supported, but you can repeat tests that use fixtures.

> New in version 3.12.0.

--restore-session [REPORT1[,REPORT2,...]]

> Restore a testing session that has run previously.

REPORT1 etc. are a run report files generated by ReFrame. If a 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.

Multiple reports may be passed as a comma-separated list. ReFrame will try to restore any required test case by looking it up in each report sequentially. If it cannot find it, it will issue an error and exit.

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.

Changed in version 3.6.1: Multiple report files are now accepted.

-S, --setvar=[TEST.]VAR=VAL

Set variable `VAR` in all tests or optionally only in test `TEST` to `VAL`.

`TEST` can have the form `[TEST.][FIXT.]*`, in which case `VAR` will be set in fixture `FIXT` of `TEST`. Note that this syntax is recursive on fixtures, so that a variable can be set in a fixture arbitrarily deep. `TEST` prefix refers to the test class name, *not* the test name and `FIXT` refers to the fixture *variable name* inside the referenced test, i.e., the test variable to which the fixture is bound. The fixture variable name is referred to as `'<varname>` when listing tests with the `-l` and `-L` options.

Multiple variables can be set at the same time by passing this option multiple times. This option *cannot* change arbitrary test attributes, but only test variables declared with the `variable` built-in. If an attempt is made to change an inexistent variable or a test parameter, a warning will be issued.

ReFrame will try to convert `VAL` to the type of the variable. If it does not succeed, a warning will be issued and the variable will not be set. `VAL` can take the special value `@none` to denote that the variable must be set to `None`. Boolean variables can be set in one of the following ways:

- By passing `true`, `yes` or `1` to set them to `True`.
- By passing `false`, `no` or `0` to set them to `False`.

Passing any other value will issue an error.

Note: Boolean variables in a test must be declared of type *Bool* and *not* of the built-in `bool` type, in order to adhere to the aforementioned behaviour. If a variable is defined as `bool` there is no way you can set it to `False`, since all strings in Python evaluate to `True`.

Sequence and mapping types can also be set from the command line by using the following syntax:

- Sequence types: `-S seqvar=1,2,3,4`
- Mapping types: `-S mapvar=a:1,b:2,c:3`

Conversions to arbitrary objects are also supported. See *ConvertibleType* for more details.

Variable assignments passed from the command line happen *before* the test is instantiated and is the exact equivalent of assigning a new value to the variable *at the end* of the test class body. This has a number of implications that users of this feature should be aware of:

- In the following test, `num_tasks` will have always the value 1 regardless of any command-line assignment of the variable `foo`:

```
@rfm.simple_test
class my_test(rfm.RegressionTest):
    foo = variable(int, value=1)
    num_tasks = foo
```

Tip: In cases where the class body expresses logic as a function of a variable and this variable, as well as its dependent logic, need to be controlled externally, the variable's default value (i.e. the value set through the value argument) may be modified as follows through an environment variable and not through the *-S* option:

```
import os

@rfm.simple_test
class my_test(rfm.RegressionTest):
    max_nodes = variable(int, value=int(os.getenv('MAX_NODES', 1)))
    # Parameterise number of nodes
    num_nodes = parameter((1 << i for i in range(0, int(max_nodes))))
```

- If the variable is set in any pipeline hook, the command line assignment will have an effect until the variable assignment in the pipeline hook is reached. The variable will be then overwritten.

- The *test filtering* happens *after* a test is instantiated, so the only way to scope a variable assignment is to prefix it with the test class name. However, this has some positive side effects:

  - Passing `-S valid_systems='*'` and `-S valid_prog_environs='*'` is the equivalent of passing the `--skip-system-check` and `--skip-prgenv-check` options.

  - Users could alter the behavior of tests based on tag values that they pass from the command line, by changing the behavior of a test in a post-init hook based on the value of the `tags` attribute.

  - Users could force a test with required variables to run if they set these variables from the command line. For example, the following test could only be run if invoked with `-S num_tasks=<NUM>`:

```
@rfm.simple_test
class my_test(rfm.RegressionTest):
    num_tasks = required
```

New in version 3.8.0.

Changed in version 3.9.3: Proper handling of boolean variables.

Changed in version 3.11.1: Allow setting variables in fixtures.

--skip-performance-check

Skip performance checking phase.

The phase is completely skipped, meaning that performance data will *not* be logged.

--skip-sanity-check

Skip sanity checking phase.

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, --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.

-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.

--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.

--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.

--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.

--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.

-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.

Miscellaneous options

-C, --config-file=FILE

Use FILE as configuration file for ReFrame.

This option can be passed multiple times, in which case multiple configuration files will be read and loaded successively. The base of the configuration chain is always the builtin configuration file, namely the ${RFM_INSTALL_PREFIX}/reframe/core/settings.py. At any point, the user can "break" the chain of configuration files by prefixing the configuration file name with a colon as in the following example: -C :/path/to/new_config.py. This will ignore any previously loaded configuration file and will only load the one specified. Note, however, that the builtin configuration file cannot be overriden; It will always be loaded first in the chain.

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

In order to determine its final configuration, ReFrame first loads the builtin configuration file unconditionally and then starts looking for possible configuration file locations defined in the *RFM_CONFIG_PATH* environment variable. For each directory defined in the *RFM_CONFIG_PATH*, ReFrame looks for a file named settings.py or settings.json inside it and loads it. If both a settings.py and a settings.json files are found, the Python configuration will be preferred. ReFrame, finally, processes any configuration files specified in the command line or in the *RFM_CONFIG_FILES* environment variable.

Changed in version 4.0.0.

--detect-host-topology[=FILE]

Detect the local host processor topology, store it to FILE and exit.

If no FILE is specified, the standard output will be used.

New in version 3.7.0.

--failure-stats

Print failure statistics at the end of the run.

-h, --help

Print a short help message and exit.

--nocolor

Disable output coloring.

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

--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.

-q, --quiet

Decrease the verbosity level.

This option can be specified multiple times. Every time this option is specified, the verbosity level will be decreased by one. This option can be combined arbitrarily with the `-v` option, in which case the final verbosity level will be determined by the final combination. For example, specifying -qv will not change the verbosity level, since the two options cancel each other, but -qqv is equivalent to -q. For a list of ReFrame's verbosity levels, see the description of the `-v` option.

New in version 3.9.3.

--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.

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

--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, --version

>   Print version and exit.

-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.

Test Naming Scheme

New in version 3.10.0.

This section describes the test naming scheme. This scheme has superseded the old one in ReFrame 4.0.

Each ReFrame test is assigned a unique name, which will be used internally by the framework to reference the test. Any test-specific path component will use that name, too. It is formed as follows for the various types of tests:

- *Regular tests*: The unique name is simply the test class name. This implies that you cannot load two tests with the same class name within the same run session even if these tests reside in separate directories.

- *Parameterized tests*: The unique name is formed by the test class name followed by an _ and the variant number of the test. Each point in the parameter space of the test is assigned a unique variant number.

- *Fixtures*: The unique name is formed by the test class name followed by an _ and a hash. The hash is constructed by combining the information of the fixture variant (if the fixture is parameterized), the fixture's scope and any fixture variables that were explicitly set.

Since unique names can be cryptic, they are not listed by the *-l* option, but are listed when a detailed listing is requested by using the *-L* option.

A human readable version of the test name, which is called the *display name*, is also constructed for each test. This name encodes all the parameterization information as well as the fixture-specific information (scopes, variables). The format of the display name is the following in BNF notation:

```
<display_name> ::= <test_class_name> (<params>)* (<scope>)?
<params> ::= "%" <parametrization> "=" <pvalue>
<parametrization> ::= (<fname> ".")* <pname>
<scope> ::= "~" <scope_descr>
<scope_descr> ::= <first> ("+" <second>)*

<test_class_name> ::= (* as in Python *)
<fname> ::= (* string *)
<pname> ::= (* string *)
<pvalue> ::= (* string *)
<first> ::= (* string *)
<second> ::= (* string *)
```

The following is an example of a fictitious complex test that is itself parameterized and depends on parameterized fixtures as well.

```
import reframe as rfm


class MyFixture(rfm.RunOnlyRegressionTest):
    p = parameter([1, 2])


class X(rfm.RunOnlyRegressionTest):
    foo = variable(int, value=1)


@rfm.simple_test
class TestA(rfm.RunOnlyRegressionTest):
    f = fixture(MyFixture, scope='test', action='join')
    x = parameter([3, 4])
    t = fixture(MyFixture, scope='test')
    l = fixture(X, scope='environment', variables={'foo': 10})
    valid_systems = ['*']
    valid_prog_environs = ['*']
```

Here is how this test is listed where the various components of the display name can be seen:

```
- TestA %x=4 %l.foo=10 %t.p=2 /8804be5d
    ^MyFixture %p=1 ~TestA_3 't 'f /f027ee75
    ^MyFixture %p=2 ~TestA_3 't 'f /830323a4
    ^X %foo=10 ~generic:default+builtin 'l /7dae3cc5
- TestA %x=3 %l.foo=10 %t.p=2 /89f6f5d1
    ^MyFixture %p=1 ~TestA_2 't 'f /02368516
    ^MyFixture %p=2 ~TestA_2 't 'f /854b99b5
    ^X %foo=10 ~generic:default+builtin 'l /7dae3cc5
- TestA %x=4 %l.foo=10 %t.p=1 /af9b2941
    ^MyFixture %p=2 ~TestA_1 't 'f /f0383f7f
    ^MyFixture %p=1 ~TestA_1 't 'f /d07f4281
    ^X %foo=10 ~generic:default+builtin 'l /7dae3cc5
- TestA %x=3 %l.foo=10 %t.p=1 /a9e50aa3
    ^MyFixture %p=2 ~TestA_0 't 'f /b894ab05
    ^MyFixture %p=1 ~TestA_0 't 'f /ca376ca8
    ^X %foo=10 ~generic:default+builtin 'l /7dae3cc5
Found 4 check(s)
```

Notice that the variable name to which every fixture is bound in its parent test is also listed as '<varname>. This is useful for setting variables down the fixture hierarchy using the –S option.

Display names may not always be unique. Assume the following test:

```
class MyTest(RegressionTest):
    p = parameter([1, 1, 1])
```

This generates three different tests with different unique names, but their display name is the same for all: MyTest %p=1. Notice that this example leads to a name conflict with the old naming scheme, since all tests would be named MyTest_1.

Each test is also associated with a hash code that is derived from the test name, its parameters and their values. As in

the example listing above, the hash code of each test is printed with the `-l` option and individual tests can be selected by their hash using the `-n` option, e.g., `-n /1c51609b`. The stage and output directories, as well as the performance log file of the `filelog` performance log handler will use the hash code for the test-specific directories and files. This might lead to conflicts for tests as the one above when executing them with the asynchronous execution policy, but ensures consistency of performance record files when parameter values are added to or deleted from a test parameter. More specifically, the test's hash will not change if a new parameter value is added or deleted or even if the parameter values are shuffled. Test variants on the other side are more volatile and can change with such changes. Also users should not rely on how the variant numbers are assigned to a test, as this is an implementation detail.

Changed in version 4.0.0: A hash code is associated with each test.

Differences from the old naming scheme

Prior to version 3.10, ReFrame used to encode the parameter values of an instance of parameterized test in its name. It did so by taking the string representation of the value and replacing any non-alphanumeric character with an underscore. This could lead to very large and hard to read names when a test defined multiple parameters or the parameter type was more complex. Very large test names meant also very large path names which could also lead to problems and random failures. Fixtures followed a similar naming pattern making them hard to debug.

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`, `y` (case insensitive) or `1` to denote true and any value of `false`, `no`, `n` (case insensitive) or `0` to denote false.

Changed in version 3.9.2: Values `1` and `0` are now valid for boolean environment variables.

Here is an alphabetical list of the environment variables recognized by ReFrame. Whenever an environment variable is associated with a configuration option, its default value is omitted as it is the same.

RFM_AUTODETECT_FQDN

Use the fully qualified domain name as the hostname. This is a boolean variable and defaults to `0`.

Associated command line option	N/A
Associated configuration parameter	N/A

New in version 3.11.0.

Changed in version 4.0.0: This variable now defaults to `0`.

RFM_AUTODETECT_METHOD

Method to use for detecting the current system and pick the right configuration. The following values can be used:

- `hostname`: The `hostname` command will be used to detect the current system. This is the default value, if not specified.

Associated command line option	N/A
Associated configuration parameter	N/A

New in version 3.11.0.

RFM_AUTODETECT_XTHOSTNAME

Use /etc/xthostname file, if present, to retrieve the current system's name. If the file cannot be found, the hostname will be retrieved using the `hostname` command. This is a boolean variable and defaults to `0`.

This option meaningful for Cray systems.

Associated command line option	N/A
Associated configuration parameter	N/A

New in version 3.11.0.

Changed in version 4.0.0: This variable now defaults to `0`.

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`

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`

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`

RFM_COLORIZE

Enable output coloring.

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

RFM_COMPRESS_REPORT

Compress the generated run report file.

Associated command line option	`--compress-report`
Associated configuration parameter	`compress_report`

New in version 3.12.0.

RFM_CONFIG_FILE

Set the configuration file for ReFrame.

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

Deprecated since version 4.0.0: Please use the `RFM_CONFIG_FILES` instead.

RFM_CONFIG_FILES

A colon-separated list of configuration files to load. Refer to the documentation of the `--config-file` option for a detailed description on how ReFrame loads its configuration.

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

New in version 4.0.0.

RFM_CONFIG_PATH

A colon-separated list of directories that contain ReFrame configuration files. Refer to the documentation of the `--config-file` option for a detailed description on how ReFrame loads its configuration.

Associated command line option	N/A
Associated configuration parameter	N/A

New in version 4.0.0.

RFM_GIT_TIMEOUT

Timeout value in seconds used when checking if a git repository exists.

Associated command line option	N/A
Associated configuration parameter	git_timeout

New in version 3.9.0.

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

New in version 3.1.

RFM_HTTPJSON_URL

The URL of the server to send performance logs in JSON format. The URL is specified in `scheme://host:port/path` format.

Associated command line option	N/A
Associated configuration parameter	url

New in version 3.6.1.

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

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`

RFM_MODULE_MAP_FILE

A file containing module mappings.

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

RFM_MODULE_MAPPINGS

A comma-separated list of module mappings.

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

RFM_NON_DEFAULT_CRAYPE

Test a non-default Cray Programming Environment.

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

RFM_OUTPUT_DIR

Directory prefix for test output files.

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

RFM_PERF_INFO_LEVEL

Logging level at which the immediate performance information is logged.

Associated command line option	n/a
Associated configuration parameter	`perf_info_level`

RFM_PERFLOG_DIR

Directory prefix for logging performance data.

Associated command line option	`--perflogdir`
Associated configuration parameter	`basedir`

RFM_PREFIX

General directory prefix for ReFrame-generated directories.

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

RFM_PURGE_ENVIRONMENT

Unload all environment modules before acting on any tests.

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

RFM_REMOTE_DETECT

Auto-detect processor information of remote partitions as well.

Associated command line option	N/A
Associated configuration parameter	*remote_detect*

New in version 3.7.0.

RFM_REMOTE_WORKDIR

The temporary directory prefix that will be used to create a fresh ReFrame clone, in order to auto-detect the processor information of a remote partition.

Associated command line option	N/A
Associated configuration parameter	*remote_workdir*

New in version 3.7.0.

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*

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*

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*

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*

RFM_STAGE_DIR

Directory prefix for staging test resources.

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

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`

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`

RFM_TRAP_JOB_ERRORS

Trap job errors in submitted scripts and fail tests automatically.

Associated configuration parameter	`trap_job_errors`

New in version 3.9.0.

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`

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`

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`

RFM_VERBOSE

Set the verbosity level of output.

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

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/reframe-hpc/reframe>

See Also

See full documentation online: <https://reframe-hpc.readthedocs.io/>

2.6.2 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. The final configuration for ReFrame is validated against this schema.

The syntax we use to describe the different configuration objects follows the convention: `OBJECT[.OBJECT]*.`
`PROPERTY`. Even if a configuration object contains a list of other objects, this is not reflected in the above syntax,
as all objects in a certain list are homogeneous. For example, by `systems.partitions.name` we designate the `name`
property of any partition object inside the `partitions` property of any system object inside the top level `systems`
object. If we were to use indices, that would be rewritten as `systems[i].partitions[j].name` where `i` indexes the
systems and `j` indexes the partitions of the i-th system. For cases, where the objects in a list are not homogeneous, e.g.,
the logging handlers, we surround the object type with `...` For example, the `logging.handlers..filelog..name`
syntax designates the `name` attribute of the `filelog` logging handler.

Top-level Configuration

The top-level configuration object is essentially the full configuration of ReFrame. It consists of the following proper-
ties, which we also call conventionally *configuration sections*:

`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*.

`modes`

> **Required**
> > No
>
> A list of *execution mode configuration objects*.

`general`

> **Required**
> > No
>
> A list of *general configuration objects*.

Warning: Changed in version 4.0.0: The `schedulers` section is removed. Scheduler options should be set per
partition using the `sched_options` attribute.

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.`max_local_jobs`

The maximum number of forced local build or run jobs allowed.

Forced local jobs run within the execution context of ReFrame.

> **Required**
>> No
>
> **Default**
>> 8

New in version 3.10.0.

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.**env_vars**

> **Required**
>> No
>
> **Default**
>> []

A list of environment variables to be set always when running on this system. These variables modify the ReFrame 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.

New in version 4.0.0.

systems.**variables**

> Deprecated since version 4.0.0: Please use `env_vars` instead. If specified in conjunction with `env_vars`, it will be ignored.

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 `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.

systems.`sched_options`

> **Required**
>> No
>
> **Default**
>> `{}`

Scheduler options for the local scheduler that is associated with the ReFrame's execution context. To understand the difference between the different execution contexts, please refer to "*Where each pipeline stage is executed?*" For the available scheduler options, see the `sched_options` in the partition configuration below.

New in version 4.0.0.

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.

- `flux`: Jobs will be launched using the Flux Framework scheduler.

- `oar`: Jobs will be launched using the OAR scheduler.

- `pbs`: Jobs will be launched using the PBS Pro scheduler.

- `sge`: Jobs will be launched using the Sun Grid Engine 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.

- `torque`: Jobs will be launched using the Torque scheduler.

- `lsf`: Jobs will be launched using the LSF scheduler.

New in version 3.7.2: Support for the SGE scheduler is added.

New in version 3.8.2: Support for the OAR scheduler is added.

New in version 3.11.0: Support for the LSF scheduler is added.

Note: The way that multiple node jobs are submitted using the SGE scheduler can be very site-specific. For this reason, the `sge` scheduler backend does not try to interpret any related arguments, e.g., `num_tasks`, `num_tasks_per_node` etc. Users must specify how these resources are to be requested by setting the *resources* partition configuration parameter and then request them from inside a test using the *extra_resources* test attribute. Here is an example configuration for a system partition named `foo` that defines different ways for submitting MPI-only, OpenMP-only and MPI+OpenMP jobs:

```
{
    'name': 'foo',
    'scheduler': 'sge',
    'resources': [
        {
            'name': 'smp',
            'options': ['-pe smp {num_slots}']
        },
        {
            'name': 'mpi',
            'options': ['-pe mpi {num_slots}']
        },
        {
            'name': 'mpismp',
            'options': ['-pe mpismp {num_slots}']
        }
    ]
}
```

Each test then can request the different type of slots as follows:

```
self.extra_resouces = {
    'smp': {'num_slots': self.num_cpus_per_task},
    'mpi': {'num_slots': self.num_tasks},
    'mpismp': {'num_slots': self.num_tasks*self.num_cpus_per_task}
}
```

Notice that defining *extra_resources* allows the test to be portable to other systems that have different schedulers; the *extra_resources* will be simply ignored in this case and the scheduler backend will interpret the different test fields in the appropriate way.

---

systems.partitions.**sched_options**

> **Required**
> > No
>
> **Default**
> > {}

Scheduler-specific options for this partition. See below for the available options.

New in version 4.0.0.

systems.partitions.sched_options.**ignore_reqnodenotavail**

> **Required**
> > No
>
> **Default**
> > false

Ignore the `ReqNodeNotAvail` Slurm state.

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.

This option is relevant for the Slurm backends only.

systems.partitions.sched_options.**job_submit_timeout**

> **Required**
> > No
>
> **Default**
> > 60

Timeout in seconds for the job submission command.

If timeout is reached, the test issuing that command will be marked as a failure.

systems.partitions.sched_options.**resubmit_on_errors**

> **Required**
> > No
>
> **Default**
> > []

If any of the listed errors occur, 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.

This option is relevant for the Slurm backends only.

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.

`systems.partitions.sched_options.`**`use_nodes_option`**

> **Required**
>> No
>
> **Default**
>> `false`

Always emit the `--nodes` Slurm option in the preamble of the job script.

This option is relevant for the Slurm backends only.

`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.

- `lrun`: Parallel programs will be launched using LC Launcher's `lrun` command.

- `lrun-gpu`: Parallel programs will be launched using LC Launcher's `lrun -M "-gpu"` command that enables the CUDA-aware Spectrum MPI.

- `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.

Tip: New in version 4.0.0: ReFrame also allows you to register your own custom launchers simply by defining them in the configuration. You can follow a small tutorial *here*.

---

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.**env_vars**

> **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.

New in version 4.0.0.

`systems.partitions.`**`variables`**

> Deprecated since version 4.0.0: Please use `env_vars` instead. If specified in conjunction with `env_vars`, it will be ignored.

`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*. If not set, ReFrame will try to auto-detect this information (see *Auto-detecting processor information* for more information).

New in version 3.5.0.

Changed in version 3.7.0: ReFrame is now able to detect the processor information automatically.

systems.partitions.**devices**

> **Required**
>> No
>
> **Default**
>> []

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

New in version 3.5.0.

systems.partitions.**features**

> **Required**
>> No
>
> **Default**
>> []

User defined features of the partition. These are accessible through the `features` attribute of the `current_partition` and can also be selected through the extended syntax of `valid_systems`. The values of this list must be alphanumeric strings starting with a non-digit character and may also contain a -.

New in version 3.11.0.

systems.partitions.**extras**

> **Required**
>> No
>
> **Default**
>> {}

User defined attributes of the partition. These are accessible through the `extras` attribute of the `current_partition` and can also be selected through the extended syntax of `valid_systems`. The attributes of this object must be alphanumeric strings starting with a non-digit character and their values can be of any type.

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:

- `Apptainer`: The Apptainer container runtime.
- `Docker`: The Docker container runtime.
- `Sarus`: The Sarus container runtime.
- `Shifter`: The Shifter container runtime.
- `Singularity`: The Singularity container runtime.

`systems.partitions.container_platforms.`**`default`**

> **Required**
>> No

If set to `true`, this is the default container platform of this partition. If not specified, the default container platform is assumed to be the first in the list of `container_platforms`.

New in version 3.12.0.

`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.`**`env_vars`**

> **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.

New in version 4.0.0.

`systems.partitions.container_platforms.`**`variables`**

Deprecated since version 4.0.0: Please use `env_vars` instead. If specified in conjunction with `env_vars`, it will be ignored.

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 needs 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.**env_vars**

> **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.
>
> New in version 4.0.0.

environments.**variables**

> Deprecated since version 4.0.0: Please use `env_vars` instead. If specified in conjunction with `env_vars`, it will be ignored.

environments.**features**

> **Required**
> > No
>
> **Default**
> > []
>
> User defined features of the environment. These are accessible through the `features` attribute of the `current_environ` and can also be selected through the extended syntax of `valid_prog_environs`. The values of this list must be alphanumeric strings starting with a non-digit character and may also contain a `-`.
>
> New in version 3.11.0.

environments.**extras**

> **Required**
> > No
>
> **Default**
> > {}

User defined attributes of the environment. These are accessible through the `extras` attribute of the *current_environ* and can also be selected through the extended syntax of *valid_prog_environs*. The attributes of this object must be alphanumeric strings starting with a non-digit character and their values can be of any type.

New in version 3.9.1.

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; does 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.`perflog_compat`

> **Required**
>> No
>
> **Default**
>> `false`

Emit a separate log record for each performance variable. Set this option to `true` if you want to keep compatibility with the performance logging prior to ReFrame 4.0.

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, have a look at the `target_systems` definition for environments.

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.
- `httpjson`: This handler sends log records in JSON format using HTTP post requests. 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 attributes:

Table 1: :header: "Log record attribute", "Description"

%(check_build_locally)s	The value of the *build_locally* attribute.
%(check_build_time_limit)s	The value of the *build_time_limit* attribute.
%(check_descr)s	The value of the *descr* attribute.
%(check_display_name)s	The value of the *display_name* attribute.
%(check_environ)s	The name of the test's *current_environ*.
%(check_env_vars)s	The value of the *env_vars* attribute.
%(check_exclusive_access)s	The value of the *exclusive_access* attribute.
%(check_executable)s	The value of the *executable* attribute.
%(check_executable_opts)s	The value of the *executable_opts* attribute.
%(check_extra_resources)s	The value of the *extra_resources* attribute.
%(check_hashcode)s	The unique hash associated with this test.
%(check_info)s	Various information about this test; essentially the return value of the test's *info()* fu
%(check_job_completion_time)s	Same as the (check_job_completion_time_unix)s but formatted according to da
%(check_job_completion_time_unix)s	The completion time of the associated run job (see *completion_time*).
%(check_job_exitcode)s	The exit code of the associated run job.
%(check_job_nodelist)s	The list of nodes that the associated run job has run on.
%(check_jobid)s	The ID of the associated run job.
%(check_keep_files)s	The value of the *keep_files* attribute.
%(check_local)s	The value of the *local* attribute.
%(check_maintainers)s	The value of the *maintainers* attribute.
%(check_max_pending_time)s	The value of the *max_pending_time* attribute.
%(check_modules)s	The value of the *modules* attribute.
%(check_name)s	The value of the *name* attribute.
%(check_num_cpus_per_task)s	The value of the *num_cpus_per_task* attribute.
%(check_num_gpus_per_node)s	The value of the *num_gpus_per_node* attribute.
%(check_num_tasks)s	The value of the *num_tasks* attribute.
%(check_num_tasks_per_core)s	The value of the *num_tasks_per_core* attribute.
%(check_num_tasks_per_node)s	The value of the *num_tasks_per_node* attribute.
%(check_num_tasks_per_socket)s	The value of the *num_tasks_per_socket* attribute.
%(check_outputdir)s	The value of the *outputdir* attribute.
%(check_partition)s	The name of the test's *current_partition*.
%(check_perfvalues)s	All the performance variables of the test combined. These will be formatted according
%(check_postbuild_cmds)s	The value of the *postbuild_cmds* attribute.
%(check_postrun_cmds)s	The value of the *postrun_cmds* attribute.
%(check_prebuild_cmds)s	The value of the *prebuild_cmds* attribute.
%(check_prefix)s	The value of the *prefix* attribute.
%(check_prerun_cmds)s	The value of the *prerun_cmds* attribute.
%(check_readonly_files)s	The value of the *readonly_files* attribute.
%(check_short_name)s	The value of the *short_name* attribute.
%(check_sourcepath)s	The value of the *sourcepath* attribute.
%(check_sourcesdir)s	The value of the *sourcesdir* attribute.
%(check_stagedir)s	The value of the *stagedir* attribute.
%(check_strict_check)s	The value of the *strict_check* attribute.
%(check_system)s	The name of the test's *current_system*.
%(check_tags)s	The value of the *tags* attribute.
%(check_time_limit)s	The value of the *time_limit* attribute.
%(check_unique_name)s	The value of the *unique_name* attribute.
%(check_use_multithreading)s	The value of the *use_multithreading* attribute.
%(check_valid_prog_environs)s	The value of the *valid_prog_environs* attribute.
%(check_valid_systems)s	The value of the *valid_systems* attribute.

co

Table 1 – continued from previous page

`%(check_variables)s`	DEPRECATED: Please use `%(check_env_vars)s` instead.
`%(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.

ReFrame allows you to log any test variable, parameter or property if they are marked as "loggable". The log record attribute will have the form `%(check_NAME)s` where `NAME` is the variable name, the parameter name or the property name that is marked as loggable.

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.

Changed in version 3.11.0: Limit the number of attributes that can be logged. User attributes or properties must be explicitly marked as "loggable" in order to be selectable for logging.

`logging.handlers.format_perfvars`

`logging.handlers_perflog.format_perfvars`

> Format specifier for logging the performance variables.
>
> This defines how the `%(check_perfvalues)s` will be formatted. Since a test may define multiple performance variables, the formatting specified in this field will be repeated for each performance variable sequentially in the *same* line.
>
> ---
>
> **Important:** The last character of this format will be interpreted as the final delimiter of the formatted performance variables to the rest of the record.
>
> ---
>
> The following log record attributes are defined additionally by this format specifier:

Log record attribute	Description
`%(check_perf_lower_thres)s`	The lower threshold of the logged performance variable.
`%(check_perf_ref)s`	The reference value of the logged performance variable.
`%(check_perf_unit)s`	The measurement unit of the logged performance variable.
`%(check_perf_upper)s`	The upper thresholds of the logged performance variable.
`%(check_perf_value)s`	The actual value of the logged performance variable.
`%(check_perf_var)s`	The name of the logged performance variable.

> ---
>
> **Important:** ReFrame versions prior to 4.0 logged a separate line for each performance variable and the `%(check_perf_*)s` attributes could be used directly in the `format`. You can re-enable this behavior by setting the `config.logging.perflog_compat` logging configuration parameter.
>
> ---
>
> New in version 4.0.0.

`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..file..`**`name`**

`logging.handlers_perflog..file..`**`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..file..`**`append`**

`logging.handlers_perflog..file..`**`ppend`**

> **Required**
>> No

> **Default**
>> `false`

> Controls whether this handler should append to its file or not.

`logging.handlers..file..`**`timestamp`**

`logging.handlers_perflog..file..`**`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 test in one or more files. The additional properties for the `filelog` handler are the following:

`logging.handlers..filelog..`**`basedir`**

`logging.handlers_perflog..filelog..`**`basedir`**

> **Required**
>> No

Default
        "./perflogs"

    The base directory of performance data log files.

`logging.handlers..filelog..`**`prefix`**

`logging.handlers_perflog..filelog..`**`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_short_name.log
        partition2/
            test_short_name.log
        ...
    system2/
    ...
```

`logging.handlers..filelog..`**`append`**

`logging.handlers_perflog..filelog..`**`append`**

        Required
            No
        Default
            true

    Open each log file in append mode.

Changed in version 4.0.0: The `filelog` handler is very cautious when generating a test log file: if a change is detected
in the information that is being logged, the hanlder will not append to the same file, but it will instead create a new one,
saving the old file using the `.h<N>` suffix, where `N` is an integer that is increased every time a new file is being created
due to such changes. Examples of changes in the logged information are when the log record format changes or a new
performance metric is added, deleted or has its name changed. This behavior guarantees that each log file is consistent
and it will not break existing parsers.

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..graylog..`**`address`**

`logging.handlers_perflog..graylog..`**`address`**

        Required
            Yes

The address of the Graylog server defined as `host:port`.

`logging.handlers..graylog..`**extras**

`logging.handlers_perflog..graylog..`**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* attribute 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..stream..`**name**

`logging.handlers_perflog..stream..`**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..syslog..`**`socktype`**

`logging.handlers_perflog..syslog..`**`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..syslog..`**`facility`**

`logging.handlers_perflog..syslog..`**`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..syslog..`**`address`**

`logging.handlers_perflog..syslog..`**`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.

The `httpjson` log handler

This handler sends log records in JSON format to a server using HTTP POST requests. The additional properties for the `httpjson` handler are the following:

`logging.handlers..httpjson..`**`url`**

`logging.handlers_perflog..httpjson..`**`url`**

> **Required**
> > Yes

> The URL to be used in the HTTP(S) request server.

`logging.handlers..httpjson..`**`extras`**

`logging.handlers_perflog..httpjson..`**`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.

`logging.handlers..httpjson..`**`ignore_keys`**

`logging.handlers_perflog..httpjson..`**`ignore_keys`**

> > **Required**
> > > No
> >
> > **Default**
> > > []
>
> These keys will be excluded from the log record that will be sent to the server.

The `httpjson` handler sends log messages in JSON format using an HTTP POST request to the specified URL.

An example configuration of this handler for performance logging is shown here:

```
{
    'type': 'httpjson',
    'url': 'http://httpjson-server:12345/rfm',
    'level': 'info',
    'extras': {
        'facility': 'reframe',
        'data-version': '1.0'
    },
    'ignore_keys': ['check_perfvalues']
}
```

This handler transmits the whole log record, meaning that all the information will be available and indexable at the remote end.

`logging.handlers..httpjson..`**`debug`**

`logging.handlers_perflog..httpjson..`**`debug`**

> > **Required**
> > > No
> >
> > **Default**
> > > false
>
> If set, the `httpjson` handler will not attempt to send the data to the server, but it will instead dump the JSON record in the current directory. The filename has the following form: `httpjson_record_<timestamp>.json`.
>
> New in version 4.1.

`logging.handlers..httpjson..`**`json_formatter`**

`logging.handlers_perflog..httpjson..`**`json_formatter`**
> A callable for converting the log record into JSON.
>
> The formatter's signature is the following:

json_formatter(*record: object*, *extras: Dict[str, str]*, *ignore_keys: Set[str]*) → str

> **Parameters**
>
> - **record** – The prepared log record. The log record is a simple Python object with all the attributes listed in *format*, as well as all the default Python log record attributes. In addition to those, there is also the special `__rfm_check__` attribute that contains a reference to the actual test for which the performance is being logged.
> - **extras** – Any extra attributes specified in *extras*.
> - **ignore_keys** – The set of keys specified in *ignore_keys*. ReFrame always adds the default Python log record attributes in this set.
>
> **Returns**
>
> A string representation of the JSON record to be sent to the server or None if the record should not be sent to the server.

Note: This configuration parameter can only be used in a Python configuration file.

New in version 4.1.

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 *only* that this execution mode is valid for. For a detailed description of this property, have a look at the *target_systems* definition for environments.

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.`compress_report`

>   **Required**
>
>>   No
>
>   **Default**
>
>>   `false`

Compress the generated run report file. See the documentation of the `--compress-report` option for more information.

New in version 3.12.0.

general.`git_timeout`

>   **Required**
>
>>   No

> **Default**
> > 5

Timeout value in seconds used when checking if a git repository exists.

general.`dump_pipeline_progress`

> Dump pipeline progress for the asynchronous execution policy in `pipeline-progress.json`. This option is meant for debug purposes only.
>
> > **Required**
> > > No
> >
> > **Default**
> > > `False`
>
> New in version 3.10.0.

general.`pipeline_timeout`

> Timeout in seconds for advancing the pipeline in the asynchronous execution policy.
>
> ReFrame's asynchronous execution policy will try to advance as many tests as possible in their pipeline, but some tests may take too long to proceed (e.g., due to copying of large files) blocking the advancement of previously started tests. If this timeout value is exceeded and at least one test has progressed, ReFrame will stop processing new tests and it will try to further advance tests that have already started.
>
> > **Required**
> > > No
> >
> > **Default**
> > > `10`
>
> New in version 3.10.0.

general.`perf_info_level`

> > **Required**
> > > No
> >
> > **Default**
> > > `"info"`
>
> The log level at which the immediate performance info will be printed.
>
> As soon as a performance test is finished, ReFrame will log its performance on the standard output immediately. This option controls at which verbosity level this info will appear.
>
> For a list of available log levels, refer to the `level` logger configuration parameter.
>
> New in version 4.0.0.

general.`remote_detect`

> > **Required**
> > > No
> >
> > **Default**
> > > `false`
>
> Try to auto-detect processor information of remote partitions as well. This may slow down the initialization of the framework, since it involves submitting auto-detection jobs to the remote partitions. For more information on how ReFrame auto-detects processor information, you may refer to *Auto-detecting processor information*.
>
> New in version 3.7.0.

general.`remote_workdir`

>   **Required**
>   No
>
>   **Default**
>   `"."`

The temporary directory prefix that will be used to create a fresh ReFrame clone, in order to auto-detect the processor information of a remote partition.

New in version 3.7.0.

general.`ignore_check_conflicts`

>   **Required**
>   No
>
>   **Default**
>   `false`

Ignore test name conflicts when loading tests.

Deprecated since version 3.8.0: This option will be removed in a future version.

general.`trap_job_errors`

>   **Required**
>   No
>
>   **Default**
>   `false`

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

New in version 3.2.

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-{sessionid}.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.

Changed in version 4.0.0: Default value was reverted back to generate a new file per run.

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, have a look at the `target_systems` definition for environments.

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

Set the verbosity level of the output. The higher the number, the more verbose the output will be. If set to a negative number, this will decrease the verbosity level.

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:

environments.modules.**name**

systems.modules.**name**

systems.partitions.modules.**name**

systems.partitions.container_platforms.modules.**name**

> **Required**
>> Yes

The name of the module.

environments.modules.**collection**

systems.modules.**collection**

systems.partitions.modules.**collection**

systems.partitions.container_platforms.modules.**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.

`environments.modules.`**`path`**

`systems.modules.`**`path`**

`systems.partitions.modules.`**`path`**

`systems.partitions.container_platforms.modules.`**`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`.

Note: In the following the term *logical CPUs* refers to the smallest processing unit recognized by the OS. Depending on the microarchitecture, this can either be a core or a hardware thread in processors that support simultaneous multithreading and this feature is enabled. Therefore, properties such as `num_cpus_per_core` may have a value greater than one.

`systems.partitions.processor.`**`arch`**

> **Required**
>> No
>
> **Default**
>> `None`

The microarchitecture of the processor.

`systems.partitions.processor.`**`num_cpus`**

> **Required**
>> No
>
> **Default**
>> `None`

Number of logical CPUs.

`systems.partitions.processor.`**`num_cpus_per_core`**

>   >   **Required**
>   >   >   No
>   >
>   >   **Default**
>   >   >   None
>
>   Number of logical CPUs per core.

`systems.partitions.processor.`**`num_cpus_per_socket`**

>   >   **Required**
>   >   >   No
>   >
>   >   **Default**
>   >   >   None
>
>   Number of logical CPUs per socket.

`systems.partitions.processor.`**`num_sockets`**

>   >   **Required**
>   >   >   No
>   >
>   >   **Default**
>   >   >   None
>
>   Number of sockets.

`systems.partitions.processor.`**`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`.

`systems.partitions.devices.`**`type`**

> **Required**
>> No
>
> **Default**
>> None

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

`systems.partitions.devices.`**`arch`**

> **Required**
>> No
>
> **Default**
>> None

The microarchitecture of the device.

`systems.partitions.devices.`**`num_devices`**

> **Required**
>> No
>
> **Default**
>> None

Number of devices of this type inside the system partition.

2.6.3 Programming APIs

Test API Reference

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.

Test Base Classes

class reframe.core.pipeline.**CompileOnlyRegressionTest**(**args*, ***kwargs*)

> Bases: *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.

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: *RegressionMixin*, 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.
>
> This class accepts parameters at the *class definition*, i.e., the test class can be defined as follows:

```
class MyTest(RegressionTest, param='foo', ...):
```

> where param is one of the following:
>
> > **Parameters**
> >
> > - **pin_prefix** – lock the test prefix to the directory where the current class lives.
> >
> > - **require_version** – a list of ReFrame version specifications that this test is allowed to run. A version specification string can have one of the following formats:
> >
> >   - VERSION: Specifies a single version.
> >
> >   - {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.
> >
> >   - V1..V2: Specifies a range of versions.
> >
> >   The test will be selected if *any* of the versions is satisfied, even if the versions specifications are conflicting.
> >
> > - **special** – allow pipeline stage methods to be overriden in this class.

> **Note:** Changed in version 2.19: Base constructor takes no arguments.
>
> New in version 3.3: The pin_prefix class definition parameter is added.
>
> New in version 3.7.0: The require_verion class definition parameter is added.

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

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.
> >
> > - *reframe.core.exceptions.ReframeSyntaxError* – If the sanity function cannot be resolved due to ambiguous syntax.

> **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_complete()

Check if the build phase has completed.

> **Returns**
>   `True` if the associated build 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.

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 = _NoRuntime

New in version 2.20.

The container platform to be used for launching this test.

This field is set automatically by the default container runtime associated with the current system partition. Users may also set this, explicitly overriding any partition setting. If the *image* attribute of *container_platform* is set, then the test will run inside a container using the specified container runtime.

```
self.container_platform = 'Singularity'
self.container_platform.image = 'docker://ubuntu:18.04'
self.container_platform.command = 'cat /etc/os-release'
```

If the test will run inside a container, the *executable* and *executable_opts* attributes are ignored. The container platform's *command* will be used instead.

Note: Only the run phase of the test will run inside the container. If you enable the containerized run in a non run-only test, the compilation phase will still run natively.

> **Type**
> str or *ContainerPlatform*.

> **Default**
> the container runtime specified in the current system partition's configuration (see also *Container Platform Configuration*).

Changed in version 3.12.0: This field is now set automatically from the current partition's configuration.

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 interdependent 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.

Changed in version 4.0.0: Passing an integer to the `how` argument is no longer supported.

descr

A detailed description of the test.

> **Type**
>> `str`
>
> **Default**
>> `''`

Changed in version 4.0: The default value is now the empty string.

property display_name

A human-readable version of the name this test.

This name contains a string representation of the various parameters of this specific test variant.

> **Type**
>> `str`

> **Note:** The display name may not be unique.

New in version 3.10.0.

env_vars = {}

Environment variables to be set before running this test.

The value of the environment variables can be of any type. ReFrame will invoke `str()` on it whenever it needs to emit it in a script.

> **Type**
> Dict[str, object]
>
> **Default**
> {}

New in version 4.0.0.

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.

If this variable is undefined when entering the compile pipeline stage, it will be set to `os.path.join('.', self.unique_name)`. Classes that override the compile stage may leave this variable undefined.

> **Type**
> str
>
> **Default**
> required

Changed in version 3.7.3: Default value changed from `os.path.join('.', self.unique_name)` to `required`.

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.

---

property fixture_variant
    The point in the fixture space for the test.

This can be seen as an index to the fixture space representing a unique combination of the fixture variants. This number is directly mapped from `variant_num`.

> **Type**
> > `int`

`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.
>
> Changed in version 3.8.0: Setting `environ` or `part` to `'*'` will skip the match check on the environment and partition, respectively.

`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_dry_run`()

> Check if the test runs in dry-run mode.
>
> New in version 4.1.

`is_fixture`()

> Check if the test is a fixture.

`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.

`is_performance_check`()

> Return `True` if the test is a performance test.

`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.

2.6. ReFrame Manuals   **211**

> **Type**
>> List[str] or Dict[str, object]
>
> **Default**
>> []

property name

> The name of the test.
>
> This is an alias of *display_name*.

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 = None

> 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 or None
>>
>> **Default**
>>> None
>
> Changed in version 4.0.0: The default value changed to None.

num_tasks = 1

> Number of tasks required by this test.
>
> If the number of tasks is set to zero or a negative value, 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 zero is equivalent to setting it to *-num_tasks_per_node*.
>
> Setting *num_tasks* to None has a scheduler-specific interpretation, but in principle, passes the responsibility of producing a correct job script to the user by setting the appropriate scheduler options. More specifically, the different backends interpret the None *num_tasks* as follows:
>
> - flux: not applicable.
> - local: not applicable.
> - lsf: Neither the -nnodes nor the -n will be emitted.
> - oar: Resets it to 1.
> - pbs: Resets it to 1.
> - sge: not applicable.
> - slurm: Neither the --ntasks nor the --nodes option (if the *use_nodes_option* is specified) will be emitted.
> - squeue: See slurm backend.
> - torque: See pbs backend.

Type
  integral or None

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.

Changed in version 4.1: Allow *num_tasks* to be None.

---

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.

property param_variant

  The point in the parameter space for the test.

This can be seen as an index to the paraemter space representing a unique combination of the parameter values. This number is directly mapped from `variant_num`.

> **Type**
> > `int`

perf_patterns

> Patterns for verifying the performance of this test.
>
> 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`

> **Warning:** You are advised to follow the new syntax for defining performance variables in your tests using either the `@performance_function` builtin or the `perf_variables`, as `perf_patterns` will likely be deprecated in the future.

perf_variables = {}

> The performance variables associated with the test.
>
> In this context, a performance variable is a key-value pair, where the key is the desired variable name and the value is the deferred performance expression (i.e. the result of a *deferrable performance function*) that computes or extracts the performance variable's value.
>
> By default, ReFrame will populate this field during the test's instantiation with all the member functions decorated with the `@performance_function` decorator. If no performance functions are present in the class, no performance checking or reporting will be carried out.
>
> This mapping may be extended or replaced by other performance variables that may be defined in any pipeline hook executing before the performance stage. To this end, deferred performance functions can be created inline using the utility `make_performance_function()`.
>
> Refer to the *ReFrame Tutorials* for concrete usage examples.
>
> > **Type**
> > > A dictionary with keys of type `str` and deferred performance expressions as values (see *Deferrable performance functions*).
> >
> > **Default**
> > > Collection of performance variables associated to each of the member functions decorated with the `@performance_function` decorator.
>
> New in version 3.8.0.

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 the parallel launch command.
>
> 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 the parallel launch command.
>
> 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:

```python
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')
    }
}
```

 To better understand how to set the performance reference tuple, here are some examples with both positive and negative reference values:

Performance Tuple	Expected	Lowest	Highest
`(100, -0.01, 0.02, 'MB/s')`	100 MB/s	99 MB/s	102 MB/s
`(100, -0.01, None, 'MB/s')`	100 MB/s	99 MB/s	inf MB/s
`(100, None, 0.02, 'MB/s')`	100 MB/s	-inf MB/s	102 MB/s
`(-100, -0.01, 0.02, 'C')`	-100 C	-101 C	-98 C
`(-100, -0.01, None, 'C')`	-100 C	-101 C	inf C
`(-100, None, 0.02, 'C')`	-100 C	-inf C	-98 C

 During the performance stage of the pipeline, the reference tuple elements, except the unit, are passed to the *assert_reference()* function along with the obtained performance value in order to actually assess whether the test passes the performance check or not.

 Type

 A scoped dictionary with system names as scopes, performance variables as keys and reference tuples as values. The elements of reference tuples cannot be deferrable expressions.

 Note: Changed in version 3.0: The measurement unit is required. The user should explicitly specify `None` if no unit is available.

 Changed in version 3.8.0: The unit in the reference tuple is again optional, but it is recommended to use it for clarity.

 Changed in version 4.0.0: Deferrable expressions are not allowed in reference tuples.

---

require_reference = False

 Require that a reference is defined for each system that this test is run on.

 If this is set and a reference is not found for the current system, the test will fail.

 Type

 boolean

> **Default**
> > False

New in version 4.0.0.

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 may be raised during sanity checking if no other sanity checking functions already exist.
>
> > **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`.

set_var_default(*name*, *value*)

Set the default value of a variable if variable is undefined.

A variable is undefined if it is declared and required and no value is yet assigned to it.

> **Parameters**
>
> - **name** – The name of the variable.
>
> - **value** – The value to set the variable to.
>
> **Raises**
> `ValueError` – If the variable does not exist

New in version 3.10.1.

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.

property short_name

A short version of the test's display name.

The shortened version coincides with the *unique_name* for simple tests and combines the test's class name and a hash code for parameterised tests.

New in version 4.0.0.

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.

skip_if_no_procinfo(*msg=None*)

> Skip test if no processor topology information is available.
>
> This method has effect only if called after the `setup` stage.
>
>> **Parameters**
>>
>> **msg** – A message explaining why the test was skipped. If not specified, a default message will be used.
>
> New in version 3.9.1.

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.

property unique_name

> The unique name of this test.
>
> > **Type**
> > > [str](#)
>
> New in version 3.10.0.

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.
>
> The syntax of this attribute is exactly the same as of the `valid_systems` except that the `a:b` entries are invalid.
>
> > **Type**
> > > List[str]
> >
> > **Default**
> > > required
>
> **See also:**
>
> - Environment features
> - Environment extras
>
> 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`.
>
> Changed in version 3.11.0: Extend syntax to support features and key/value pairs.

---

valid_systems

List of systems or system features or system properties required by this test.

Each entry in this list is a requirement and can have one of the following forms:

- `sysname`: The test is valid for system named `sysname`.

- `sysname:partname`: The test is valid for the partition `partname` of system `sysname`.

- `*`: The test is valid for any system.

- `*:partname`: The test is valid for any partition named `partname` in any system.

- `+feat`: The test is valid for all partitions that define feature `feat` as a feature.

- `-feat`: The test is valid for all partitions that do not define feature `feat` as a feature.

- `%key=val`: The test is valid for all partitions that define the extra property `key` with the value `val`.

Multiple features and key/value pairs can be included in a single entry of the *valid_systems* list, in which case an AND operation on these constraints is implied. For example, the test defining the following will be valid for all systems that have define both `feat1` and `feat2` and set `foo=1`

```
valid_systems = ['+feat1 +feat2 %foo=1']
```

For key/value pairs comparisons, ReFrame will automatically convert the value in the key/value spec to the type of the value of the corresponding entry in the partitions `extras` property. In the above example, if the type of `foo` property is integer, 1 will be converted to an integer value. If a conversion to the target type is not possible, then the requested key/value pair is not matched.

Multiple entries in the *valid_systems* list are implicitly ORed, such that the following example implies that the test is valid for either `sys1` or for any other system that does not define `feat`.

```
valid_systems = ['sys1', '-feat']
```

Type
  List[str]

Default
  None

See also:

- System partition features
- System partition extras

Changed in version 3.3: Default value changed from `[]` to `None`.

Changed in version 3.6: Default value changed from `None` to `required`.

Changed in version 3.11.0: Extend syntax to support features and key/value pairs.

variables = {}

Environment variables to be set before running this test.

This is an alias of *env_vars*.

Deprecated since version 4.0.0: Please use *env_vars* instead.

property variant_num

>   The variant number of the test.

>   This number should be treated as a unique ID representing a unique combination of the available parameter and fixture variants.

>   > **Type**
>   >
>   > > [int](#)

class reframe.core.pipeline.**RunOnlyRegressionTest**(**args*, ***kwargs*)

>   Bases: *[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.

Test Decorators

@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.

Builtins

New in version 3.4.2.

ReFrame test base classes and, in particular, the [reframe.core.pipeline.RegressionMixin](#) class, define a set of functions and decorators that can be used to define essential test elements, such as variables, parameters, fixtures, pipeline hooks etc. These are called *builtins* because they are directly available for use inside the test class body that is being defined without the need to import any module. However, almost all of these builtins are also available from the `reframe.core.builtins` module. The use of this module is required only when creating new tests programmatically using the [make_test()](#) function.

reframe.core.pipeline.RegressionMixin.**bind**(*func*, *name=None*)

> Bind a free function to a regression test.
>
> By default, the function is bound with the same name as the free function. However, the function can be bound using a different name with the `name` argument.
>
> > **Parameters**
> >
> > - **func** – external function to be bound to a class.
> >
> > - **name** – bind the function under a different name.
>
> ---
>
> **Note:** This is the only builtin that is not available through the `reframe.core.builtins` module. The reason is that the *bind()* method needs to access the class namespace directly in order to bind the free function to the class.
>
> ---
>
> New in version 3.6.2.

@reframe.core.builtins.**deferrable**

> Convert the decorated function to a deferred expression.
>
> See *Deferrable Functions Reference* for further information on deferrable functions.

reframe.core.builtins.**fixture**(*cls*, ***, *scope='test'*, *action='fork'*, *variants='all'*, *variables=None*)

> Insert a new fixture in the current test.
>
> A fixture is a regression test that creates, prepares and/or manages a resource for another regression test. Fixtures may contain other fixtures and so on, forming a directed acyclic graph. A parent fixture (or a regular regression test) requires the resources managed by its child fixtures in order to run, and it may only access these fixture resources after its `setup` pipeline stage. The execution of parent fixtures is postponed until all their respective children have completed execution. However, the destruction of the resources managed by a fixture occurs in reverse order, only after all the parent fixtures have been destroyed. This destruction of resources takes place during the `cleanup` pipeline stage of the regression test. Fixtures must not define the members *valid_systems* and *valid_prog_environs*. These variables are defined based on the values specified in the parent test, ensuring that the fixture runs with a suitable system partition and programming environment combination. A fixture's *name* attribute may be internally mangled depending on the arguments passed during the fixture declaration. Hence, manually setting or modifying the *name* attribute in the fixture class is disallowed, and breaking this restriction will result in undefined behavior.
>
> ---
>
> **Warning:** The fixture name mangling is considered an internal framework mechanism and it may change in future versions without any notice. Users must not express any logic in their tests that relies on a given fixture name mangling scheme.
>
> ---
>
> By default, the resources managed by a fixture are private to the parent test. However, it is possible to share these resources across different tests by passing the appropriate fixture `scope` argument. The different scope levels are independent from each other and a fixture only executes once per scope, where all the tests that belong to that same scope may use the same resources managed by a given fixture instance. The available scopes are:
>
> - **session**: This scope encloses all the tests and fixtures that run in the full ReFrame session. This may include tests that use different system partition and programming environment combinations. The fixture class must derive from *RunOnlyRegressionTest* to avoid any implicit dependencies on the partition or the programming environment used.
>
> - **partition**: This scope spans across a single system partition. This may include different tests that run on the same partition but use different programming environments. Fixtures with this scope

must be independent of the programming environment, which restricts the fixture class to derive from
RunOnlyRegressionTest.

- **environment**: The extent of this scope covers a single combination of system partition and programming
  environment. Since the fixture is guaranteed to have the same partition and programming environment as
  the parent test, the fixture class can be any derived class from *RegressionTest*. * **test**: This scope covers
  a single instance of the parent test, where the resources provided by the fixture are exclusive to each parent
  test instance. The fixture class can be any derived class from *RegressionTest*.

Rather than specifying the scope at the fixture class definition, ReFrame fixtures set the scope level from the
consumer side (i.e. when used by another test or fixture). A test may declare multiple fixtures using the same
class, where fixtures with different scopes are guaranteed to point to different instances of the fixture class. On
the other hand, when two or more fixtures use the same fixture class and have the same scope, these different
fixtures will point to the same underlying resource if the fixtures refer to the same *variant* of the fixture class.
The example below illustrates the different fixture scope usages:

```python
class MyFixture(rfm.RunOnlyRegressionTest):
    my_var = variable(int, value=1)
    ...


@rfm.simple_test
class TestA(rfm.RegressionTest):
    valid_systems = ['p1', 'p2']
    valid_prog_environs = ['e1', 'e2']

    # Fixture shared throughout the full session
    f1 = fixture(MyFixture, scope='session')

    # Fixture shared for each supported partition
    f2 = fixture(MyFixture, scope='partition')

    # Fixture shared for each supported part+environ
    f3 = fixture(MyFixture, scope='environment')

    # Fixture private evaluation of MyFixture
    f4 = fixture(MyFixture, scope='test')
    ...


@rfm.simple_test
class TestB(rfm.RegressionTest):
    valid_systems = ['p1']
    valid_prog_environs = ['e1']

    # Another private instance of MyFixture
    f1 = fixture(MyFixture, scope='test')

    # Same as f3 in TestA for p1 + e1
    f2 = fixture(MyFixture, scope='environment')

    # Same as f1 in TestA
    f3 = fixture(MyFixture, scope='session')
    ...
```

```
    @run_after('setup')
    def access_fixture_resources(self):
        # Dummy pipeline hook to illustrate fixture resource access
        assert self.f1.my_var is not self.f2.my_var
        assert self.f1.my_var is not self.f3.my_var
```

`TestA` supports two different valid systems and another two valid programming environments. Assuming that both environments are supported by each of the system partitions `'p1'` and `'p2'`, this test will execute a total of four times. This test uses the very simple `MyFixture` fixture multiple times using different scopes, where fixture `f1` (session scope) will be shared across the four test instances, and fixture `f4` (test scope) will be executed once per test instance. On the other hand, `f2` (partition scope) will run once per partition supported by test `TestA`, and the multiple per-partition executions (i.e. for each programming environment) will share the same underlying resource for `f2`. Lastly, `f3` will run a total of four times, which is once per partition and environment combination. This simple `TestA` shows how multiple instances from the same test can share resources, but the real power behind fixtures is illustrated with `TestB`, where this resource sharing is extended across different tests. For simplicity, `TestB` only supports a single partition `'p1'` and programming environment `'e1'`, and similarly to `TestA`, `f1` (test scope) causes a private evaluation of the fixture `MyFixture`. However, the resources managed by fixtures `f2` (environment scope) and `f3` (session scope) are shared with `Test1`.

Fixtures are treated by ReFrame as first-class ReFrame tests, which means that these classes can use the same built-in functionalities as in regular tests decorated with *@rfm.simple_test*. This includes the `parameter()` built-in, where fixtures may have more than one *variant*. When this occurs, a parent test may select to either treat a parameterized fixture as a test parameter, or instead, to gather all the fixture variants from a single instance of the parent test. In essence, fixtures implement *fork-join* model whose behavior may be controlled through the `action` argument. This argument may be set to one of the following options:

- **fork**: This option parameterizes the parent test as a function of the fixture variants. The fixture handle will resolve to a single instance of the fixture.

- **join**: This option gathers all the variants from a fixture into a single instance of the parent test. The fixture handle will point to a list containing all the fixture variants.

A test may declare multiple fixtures with different `action` options, where the default `action` option is `'fork'`. The example below illustrates the behavior of these two different options.

```
class ParamFix(rfm.RegressionTest):
    p = parameter(range(5)) # A simple test parameter
    ...


@rfm.simple_test
class TestC(rfm.RegressionTest):
    # Parameterize TestC for each ParamFix variant
    f = fixture(ParamFix, action='fork')
    ...

    @run_after('setup')
    def access_fixture_resources(self):
        print(self.f.p) # Prints the fixture's variant parameter value


@rfm.simple_test
class TestD(rfm.RegressionTest):
```

```
    # Gather all fixture variants into a single test
    f = fixture(ParamFix, action='join')
    ...


    @run_after('setup')
    def reduce_range(self):
        # Sum all the values of p for each fixture variant
        res = functools.reduce(lambda x, y: x+y,
                               (fix.p for fix in self.f))
        n = len(self.f)-1
        assert res == (n*n + n)/2
```

Here `ParamFix` is a simple fixture class with a single parameter. When the test `TestC` uses this fixture with a `'fork'` action, the test is implicitly parameterized over each variant of `ParamFix`. Hence, when the `access_fixture_resources()` post-setup hook accesses the fixture `f`, it only access a single instance of the `ParamFix` fixture. On the other hand, when this same fixture is used with a `'join'` action by `TestD`, the test is not parameterized and all the `ParamFix` instances are gathered into `f` as a list. Thus, the post-setup pipeline hook `reduce_range()` can access all the fixture variants and compute a reduction of the different `p` values.

When declaring a fixture, a parent test may select a subset of the fixture variants through the `variants` argument. This variant selection can be done by either passing an iterable containing valid variant indices (see *Test variants* for further information on how the test variants are indexed), or instead, passing a mapping with the parameter name (of the fixture class) as keys and filtering functions as values. These filtering functions are unary functions that return the value of a boolean expression on the values of the specified parameter, and they all must evaluate to `True` for at least one of the fixture class variants. See the example below for an illustration on how to filter-out fixture variants.

```
class ComplexFixture(rfm.RegressionTest):
    # A fixture with 400 different variants.
    p0 = parameter(range(100))
    p1 = parameter(['a', 'b', 'c', 'd'])
    ...


@rfm.simple_test
class TestE(rfm.RegressionTest):
    # Select the fixture variants with boolean conditions
    foo = fixture(ComplexFixture,
                  variants={'p0': lambda x: x<10,
                            'p1': lambda x: x=='d'})

    # Select the fixture variants by index
    bar = fixture(ComplexFixture, variants=range(300,310))
    ...
```

A parent test may also specify the value of different variables in the fixture class to be set before its instantiation. Each variable must have been declared in the fixture class with the `variable()` built-in, otherwise it is silently ignored. This variable specification is equivalent to deriving a new class from the fixture class, and setting these variable values in the class body of a newly derived class. Therefore, when fixture declarations use the same fixture class and pass different values to the `variables` argument, the fixture class is interpreted as a different class for each of these fixture declarations. See the example below.

```
class Fixture(rfm.RegressionTest):
    v = variable(int, value=1)
```

```
        if self.foo.x == 1:
            self.valid_systems = ['sys1]
        else:
            self.valid_systems = ['sys2']

    @run_after('setup')
    def normal_access(self):
        # Any test attribute of the associated fixture test can be
        # accessed here
        self.executable_opts = [
            '-i', os.path.join(self.foo.stagedir, 'input.txt')
        ]
```

During test initialization, ReFrame binds the `foo` name to a proxy object that holds the parameterization of the target fixture. This proxy object is recursive, so that if fixture `foo` contained another fixture named `bar`, it would allow you to access any parameters of that fixture with `self.foo.bar.param`.

During the test setup stage, the `foo`'s binding changes and it is now bound to the exact test instance that was executed for the target test instance.

> **Parameters**
>
> - **cls** – A class derived from *RegressionTest* that manages a given resource. The base from this class may be further restricted to other derived classes of *RegressionTest* depending on the `scope` parameter.
>
> - **scope** – Sets the extent to which other regression tests may share the resources managed by a fixture. The available scopes are, from more to less restrictive, `'test'`, `'environment'`, `'partition'` and `'session'`. By default a fixture's scope is set to `'test'`, which makes the resource private to the test that uses the fixture. This means that when multiple regression tests use the same fixture class with a `'test'` scope, the fixture will run once per regression test. When the scope is set to `'environment'`, the resources managed by the fixture are shared across all the tests that use the fixture and run on the same system partition and use the same programming environment. When the scope is set to `'partition'`, the resources managed by the fixture are shared instead across all the tests that use the fixture and run on the same system partition. Lastly, when the scope is set to `'session'`, the resources managed by the fixture are shared across the full ReFrame session. Fixtures with either `'partition'` or `'session'` scopes may be shared across different regression tests under different programming environments, and for this reason, when using these two scopes, the fixture class `cls` is required to derive from *RunOnlyRegressionTest*.
>
> - **action** – Set the behavior of a parameterized fixture to either `'fork'` or `'join'`. With a `'fork'` action, a parameterized fixture effectively parameterizes the regression test. On the other hand, a `'join'` action gathers all the fixture variants into the same instance of the regression test. By default, the `action` parameter is set to `'fork'`.
>
> - **variants** – Filter or sub-select a subset of the variants from a parameterized fixture. This argument can be either an iterable with the indices from the desired variants, or a mapping containing unary functions that return the value of a boolean expression on the values of a given parameter.
>
> - **variables** – Mapping to set the values of fixture's variables. The variables are set after the fixture class has been created (i.e. after the class body has executed) and before the fixture class is instantiated.

New in version 3.9.0.

Changed in version 3.11.0: Allow early access of fixture objects.

@reframe.core.builtins.**loggable_as**(*name*)

Mark a property as loggable.

> **Parameters**
>> **name** – An alternative name that will be used for logging this property. If None, the name of the decorated property will be used.
>
> **Raises**
>> **ValueError** – if the decorated function is not a property.

New in version 3.10.2.

@reframe.core.builtins.**loggable**

> Equivalent to *loggable_as(None)*.

reframe.core.builtins.**parameter**(*values=None*, *inherit_params=False*, *filter_params=None*, *fmt=None*, *loggable=False*)

Inserts a new test parameter.

At the class level, these parameters are stored in a separate namespace referred to as the *parameter space*. 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:

```python
class Foo(rfm.RegressionTest):
    variant = parameter(['A', 'B'])

    # print(variant)
    # Error: a parameter may only be accessed from the class instance

    @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:

```python
class Bar(Foo):
    @run_after('init')
    def do_something(self):
        if self.variant == 'A':
            override_this()
        else:
            override_other()
```

Moreover, a derived class may extend, partially extend and/or modify the parameter values provided in the base class as shown below.

```python
class ExtendVariant(Bar):
    # Extend the full set of inherited variant parameter values
    # to ['A', 'B', 'C']
    variant = parameter(['C'], inherit_params=True)


class PartiallyExtendVariant(Bar):
    # Extend a subset of the inherited variant parameter values
    # to ['A', 'D']
    variant = parameter(['D'], inherit_params=True,
                        filter_params=lambda x: x[:1])


class ModifyVariant(Bar):
    # Modify the variant parameter values to ['AA', 'BA']
    variant = parameter(inherit_params=True,
                        filter_params=lambda x: map(lambda y: y+'A', x))
```

A parameter with no values is referred to as an *abstract parameter* (i.e. a parameter that is declared but not defined). Therefore, classes with at least one abstract parameter are considered abstract classes.

```python
class AbstractA(Bar):
    variant = parameter()


class AbstractB(Bar):
    variant = parameter(inherit_params=True, filter_params=lambda x: [])
```

> **Parameters**
>
> - **values** – An iterable containing the parameter values.
>
> - **inherit_params** – If True, the parameter values defined in any base class will be inherited. In this case, the parameter values provided in the current class will extend the set of inherited parameter values. If the parameter does not exist in any of the parent parameter spaces, this option has no effect.
>
> - **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 iterable argument and return an iterable. It will be called with the inherited parameter values and it must return the filtered set of parameter values. This function will only have an effect if used with inherit_params=True.
>
> - **fmt** – A formatting function that will be used to format the values of this parameter in the test's *display_name*. This function should take as argument the parameter value and return a string representation of the value. If the returned value is not a string, it will be converted using the str() function.
>
> - **loggable** – Mark this parameter as loggable. If True, this parameter will become a log record attribute under the name check_NAME, where NAME is the name of the parameter.
>
> **Returns**
> A new test parameter.

New in version 3.10.0: The fmt argument is added.

New in version 3.11.0: The loggable argument is added.

@reframe.core.builtins.**performance_function**(*unit*, *, *perf_key=None*)

> Decorate a test member function to mark it as a performance metric function.
>
> This decorator converts the decorated method into a performance deferrable function (see "*Deferrable performance functions*" for more details) whose evaluation is deferred to the performance stage of the regression test. The decorated function must take a single argument without a default value (i.e. self) and any number of arguments with default values. A test may decorate multiple member functions as performance functions, where each of the decorated functions must be provided with the unit of the performance quantity to be extracted from the test. Any performance function may be overridden in a derived class and multiple bases may define their own performance functions. In the event of a name conflict, the derived class will follow Python's MRO to choose the appropriate performance function. However, defining more than one performance function with the same name in the same class is disallowed.
>
> The full set of performance functions of a regression test is stored under `perf_variables` as key-value pairs, where, by default, the key is the name of the decorated member function, and the value is the deferred performance function itself. Optionally, the key under which a performance function is stored in `perf_variables` can be customised by passing the desired key as the `perf_key` argument to this decorator.
>
> > **Parameters**
> > **unit** – A string representing the measurement unit of this metric.
>
> New in version 3.8.0.

@reframe.core.builtins.**require_deps**

> Decorator to denote that a function 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 `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.

> > **Warning:** Changed in version 3.7.0: Using this functionality from the `reframe` or `reframe.core.decorators` modules is now deprecated. You should use the built-in function described here.
> >
> > Changed in version 4.0.0: You may only use this function as framework built-in.

@reframe.core.builtins.**run_after**(*stage*)

> Attach the decorated function after a certain pipeline stage.
>
> This is analogous to `run_before()`, except that the hook will execute right after the stage it was attached to. This decorator also supports `'init'` as a valid `stage` argument, where in this case, the hook will execute right after the test is initialized (i.e. after the `__init__()` method is called) and before entering the test's pipeline. In essence, a post-init hook is equivalent to defining additional `__init__()` functions in the test. The following code

```python
class MyTest(rfm.RegressionTest):
    @run_after('init')
    def foo(self):
        self.x = 1
```

> is equivalent to

```
class MyTest(rfm.RegressionTest):
    def __init__(self):
        self.x = 1
```

Changed in version 3.5.2: Add support for post-init hooks.

@reframe.core.builtins.**run_before**(*stage*)

> Attach the decorated function before a certain pipeline stage.
>
> The function will run just before the specified pipeline stage and it cannot accept any arguments except `self`. This decorator can be stacked, in which case the function will be attached to multiple pipeline stages. See above for the valid `stage` argument values.
>
> > **Parameters**
> >
> > **stage** – The pipeline stage where this function will be attached to. See *Pipeline Hooks* for the list of valid stage values.

@reframe.core.builtins.**sanity_function**

> Decorate a test member function to mark it as a sanity check.
>
> This decorator will convert the given function into a `deferrable()` and mark it to be executed during the test's sanity stage. When this decorator is used, manually assigning a value to `sanity_patterns` in the test is not allowed.
>
> Decorated functions may be overridden by derived classes, and derived classes may also decorate a different method as the test's sanity function. Decorating multiple member functions in the same class is not allowed. However, a `RegressionTest` may inherit from multiple `RegressionMixin` classes with their own sanity functions. In this case, the derived class will follow Python's MRO to find a suitable sanity function.
>
> New in version 3.7.0.

reframe.core.builtins.**variable**(**args*, ***kwargs*)

> Insert a new 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

    @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' # This will work, but is 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.

```python
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!

    @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*.

```python
# Test as written in the library
class EchoBaseTest(rfm.RunOnlyRegressionTest):
    what = variable(str)

    valid_systems = ['*']
    valid_prog_environs = ['*']

  @run_before('run')
  def set_executable(self):
      self.executable = f'echo {self.what}'

  @sanity_function
  def assert_what(self):
      return 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
```

(continues on next page)

```python
@rfm.simple_test
class FoodTest(EchoBaseTest):
    param = parameter(['Bacon', 'Eggs'])


    @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. However, this `required` keyword is only available in the class body.

```python
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.

Finally, variables may alias each other. If a variable is an alias of another one it behaves in the exact same way as its target. If a change is made to the target variable, this is reflected to the alias and vice versa. However, alias variables are independently loggable: an alias may be logged but not its target and vice versa. Aliased variables are useful when you want to rename a variable and you want to keep the old one for compatibility reasons.

> **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`. The provided field validator by this argument must derive from `reframe.core.fields.Field`.
> - **alias** – the target variable if this variable is an alias. This must refer to an already declared variable and neither default value nor a field can be specified for an alias variable.
> - **loggable** – Mark this variable as loggable. If `True`, this variable will become a log record attribute under the name `check_NAME`, where `NAME` is the name of the variable.
> - **kwargs** – keyword arguments to be forwarded to the constructor of the field validator.
>
> **Returns**
> A new test variable.

New in version 3.10.2: The `loggable` argument is added.

New in version 4.0.0: Alias variable are introduced.

Changed in version 3.7.0: Expose *@deferrable* as a builtin.

Changed in version 3.11.0: Builtins are now available also through the `reframe.core.builtins` module.

Pipeline Hooks

ReFrame provides a mechanism to allow attaching arbitrary functions to run before or after a given stage of the execution pipeline. This is achieved through the *@run_before* and *@run_after* test builtins. Once attached to a given stage, these functions are referred to as *pipeline hooks*. A hook may be attached to multiple pipeline stages and multiple hooks may also be attached to the same pipeline stage. Pipeline hooks attached to multiple stages will be executed on each pipeline stage the hook was attached to. Pipeline stages with multiple hooks attached will execute these hooks in the order in which they were attached to the given pipeline stage. A derived class will inherit all the pipeline hooks defined in its bases, except for those whose hook function is overridden by the derived class. A function that overrides a pipeline hook from any of the base classes will not be a pipeline hook unless the overriding function is explicitly reattached to any pipeline stage. In the event of a name clash arising from multiple inheritance, the inherited pipeline hook will be chosen following Python's MRO.

A function may be attached to any of the following stages (listed in order of execution): `init`, `setup`, `compile`, `run`, `sanity`, `performance` and `cleanup`. The `init` stage refers to the test's instantiation and it runs before entering the execution pipeline. Therefore, a test function cannot be attached to run before the `init` stage. Hooks attached to any other 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.

Pipeline hooks are executed in reverse MRO order, i.e., the hooks of the least specialized class will be executed first. In the following example, `BaseTest.x()` will execute before `DerivedTest.y()`:

```
class BaseTest(rfm.RegressionTest):
    @run_after('setup')
    def x(self):
        '''Hook x'''

class DerivedTest(BaseTeset):
    @run_after('setup')
    def y(self):
        '''Hook y'''
```

See also:

- *@run_before*, *@run_after* decorators

Note: Pipeline hooks do not execute in the test's stage directory, but in the directory that ReFrame executes in. However, the test's *stagedir* can be accessed by explicitly changing the working directory from within the hook function itself (see the *change_dir* utility for further details):

```
import reframe.utility.osext as osext

class MyTest(rfm.RegressionTest):
    ...
    @run_after('run')
    def my_post_run_hook(self):
        # Access the stage directory
        with osext.change_dir(self.stagedir):
            ...
```

> **Warning:** Changed in version 3.7.0: Declaring pipeline hooks using the same name functions from the `reframe` or `reframe.core.decorators` modules is now deprecated. You should use the builtin functions described in the *Builtins* section..
>
> Changed in version 4.0.0: Pipeline hooks can only be defined through the built-in functions described in this section.

> **Warning:** Changed in version 3.9.2: Execution of pipeline hooks until this version was implementation-defined. In practice, hooks of a derived class were executed before those of its parents.
>
> This version defines the execution order of hooks, which now follows a strict reverse MRO order, so that parent hooks will execute before those of derived classes. Tests that relied on the execution order of hooks might break with this change.

Test variants

Through the `parameter()` and `fixture()` builtins, a regression test may store multiple versions or *variants* of a regression test at the class level. During class creation, the test's parameter and fixture spaces are constructed and combined, assigning a unique index to each of the available test variants. In most cases, the user does not need to be aware of all the internals related to this variant indexing, since ReFrame will run by default all the available variants for each of the registered tests. On the other hand, in more complex use cases such as setting dependencies across different test variants, or when performing some complex variant sub-selection on a fixture declaration, the user may need to access some of this low-level information related to the variant indexing. Therefore, classes that derive from the base `RegressionMixin` provide *classmethods* and properties to query these data.

> **Warning:** When selecting test variants through their variant index, no index ordering should ever be assumed, being the user's responsibility to ensure on each ReFrame run that the selected index corresponds to the desired parameter and/or fixture variants.

RegressionMixin.**num_variants**

> Total number of variants of the test.

classmethod RegressionMixin.**get_variant_nums**(***conditions*)

> Get the variant numbers that meet the specified conditions.
>
> The given conditions enable filtering the parameter space of the test. Filtering the fixture space is not allowed.

```
# Filter out the test variants where my_param is greater than 3
cls.get_variant_nums(my_param=lambda x: x < 4)
```

> The returned list of variant numbers can be passed to `variant_name()` in order to retrieve the actual test name.
>
> > **Parameters**
> >
> > > **conditions** – keyword arguments where the key is the test parameter name and the value is either a single value or a unary function that evaluates to `True` if the parameter point must be kept, `False` otherwise. If a single value is passed this is implicitly converted to the equality function, such that
> > >
> > > ```
> > > get_variant_nums(p=10)
> > > ```
> > >
> > > is equivalent to

```
get_variant_nums(p=lambda x: x == 10)
```

classmethod RegressionMixin.**variant_name**(*variant_num=None*)

Return the name of the test variant with a specific variant number.

> **Parameters**
> **variant_num** – An integer in the range of [0, cls.num_variants).

Dynamic Creation of Tests

New in version 3.10.0.

reframe.core.meta.**make_test**(*name*, *bases*, *body*, *methods=None*, ***kwargs*)

Define a new test class programmatically.

Using this method is completely equivalent to using the class to define the test class. More specifically, the following:

```
hello_cls = rfm.make_test(
    'HelloTest', (rfm.RunOnlyRegressionTest,),
    {
        'valid_systems': ['*'],
        'valid_prog_environs': ['*'],
        'executable': 'echo',
        'sanity_patterns': sn.assert_true(1)
    }
)
```

is completely equivalent to

```
class HelloTest(rfm.RunOnlyRegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    executable = 'echo'
    sanity_patterns: sn.assert_true(1)

hello_cls = HelloTest
```

Test *builtins* can also be used when defining the body of the test by accessing them through the reframe.core.builtins. Methods can also be bound to the newly created tests using the methods argument. The following is an example:

```
import reframe.core.builtins as builtins


def set_message(obj):
    obj.executable_opts = [obj.message]


def validate(obj):
    return sn.assert_found(obj.message, obj.stdout)


hello_cls = rfm.make_test(
    'HelloTest', (rfm.RunOnlyRegressionTest,),
```

```
    {
        'valid_systems': ['*'],
        'valid_prog_environs': ['*'],
        'executable': 'echo',
        'message': builtins.variable(str)
    },
    methods=[
        builtins.run_before('run')(set_message),
        builtins.sanity_function(validate)
    ]
)
```

Parameters

- **name** – The name of the new test class.
- **bases** – A tuple of the base classes of the class that is being created.
- **body** – A mapping of key/value pairs that will be inserted as class attributes in the newly created class.
- **methods** – A list of functions to be bound as methods to the class that is being created. The functions will be bound with their original name.
- **kwargs** – Any keyword arguments to be passed to the `RegressionTestMeta` metaclass.

New in version 3.10.0.

Changed in version 3.11.0: Added the `methods` arguments.

Environments and Systems

class reframe.core.environments.**Environment**(*name*, *modules=None*, *env_vars=None*, *extras=None*, *features=None*)

Bases: `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 env_vars

The environment variables associated with this environment.

> **Type**
> `OrderedDict[str, str]`

New in version 4.0.0.

property extras

User defined properties specified in the configuration.

New in version 3.9.1.

> **Type**
>   Dict[str, object]

property features

Used defined features specified in the configuration.

New in version 3.11.0.

> **Type**
>   List[str]

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.

Deprecated since version 4.0.0: Please *env_vars* instead.

class reframe.core.environments.**ProgEnvironment**(*name*, *modules=None*, *env_vars=None*, *extras=None*, *features=None*, *cc='cc'*, *cxx='CC'*, *ftn='ftn'*, *nvcc='nvcc'*, *cppflags=None*, *cflags=None*, *cxxflags=None*, *fflags=None*, *ldflags=None*, ***kwargs*)

Bases: *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: *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.**DeviceInfo**(*info*)

> Bases: _ReadOnlyInfo, JSONSerializable
>
> A representation of a device inside ReFrame.
>
> You can access all the keys of the device configuration object.
>
> New in version 3.5.0.
>
> > **Warning:** Users may not create `DeviceInfo` objects directly.
>
> **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.**ProcessorInfo**(*info*)

> Bases: _ReadOnlyInfo, JSONSerializable
>
> A representation of a processor inside ReFrame.
>
> You can access all the keys of the processor configuration object.
>
> New in version 3.5.0.
>
> > **Warning:** Users may not create `ProcessorInfo` objects directly.
>
> **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_numa_nodes

    Number of NUMA nodes.

        Type

            integral or None

class reframe.core.systems.**System**(*name*, *descr*, *hostnames*, *modules_system*, *preload_env*, *prefix*, *outputdir*, *resourcesdir*, *stagedir*, *partitions*)

Bases: 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_runtime, container_environs, resources, local_env, environs, max_jobs, prepare_cmds, processor, devices, extras, features, time_limit*)

Bases: 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 container_runtime

    The default container runtime of this partition.

    Type

        str or None

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.DeviceInfo]`

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 properties associated with this partition.
>
> These extras are defined in the configuration.
>
> New in version 3.5.0.
>
> > **Type**
> >
> > > `Dict[str, object]`

property features

> User defined features associated with this partition.
>
> These features are defined in the configuration.
>
> New in version 3.11.0.
>
> > **Type**
> >
> > > `List[str]`

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_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.ProcessorInfo*

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.

---

select_devices(*devtype*)

Return all devices of the requested type:

> **Parameters**
>> **devtype** – The type of the device info objects to return.

> **Returns**
>> A list of *DeviceInfo* objects of the specified type.

property time_limit

The time limit that will be used when submitting jobs to this partition.

> **Type**
>> `str` or `None`

New in version 3.11.0.

Jobs and Parallel Launchers

class reframe.core.schedulers.**Job**(*args*, ***kwargs*)

Bases: `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 cli_options

The scheduler options passed through the `-J` command line options.

property completion_time

The completion time of this job as a floating point number expressed in seconds since the epoch, in UTC.

This attribute is `None` if the job hasn't been finished yet, or if ReFrame runtime hasn't perceived it yet.

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.

> **Type**
>> `float` or None

property exception

The last exception that this job encountered.

The scheduler will raise this exception the next time the status of this job is queried.

exclusive_access = False

Request exclusive access on the nodes for this job.

> **Type**
>> `bool`
>
> **Default**
>> `false`

> **Note:** This attribute is set by the framework just before submitting the job based on the test information.

New in version 3.11.0.

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:

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


@run_after('setup')
def set_launcher(self):
    self.job.launcher = getlauncher('local')()
```

> **Type**
>> *reframe.core.launchers.JobLauncher*

max_pending_time = None

Maximum pending time for this job.

See *reframe.core.pipeline.RegressionTest.max_pending_time* for more details.

Note: This attribute is set by the framework just before submitting the job based on the test information.

New in version 3.11.0.

property name

The name of this job.

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`

num_cpus_per_task = None

Number of processing elements associated with each task for this job.

> **Type**
>> integral or `NoneType`
>
> **Default**
>> `None`

Note: This attribute is set by the framework just before submitting the job based on the test information.

New in version 3.11.0.

num_tasks = 1

Number of tasks for this job.

> **Type**
>> integral
>
> **Default**
>> 1

Note: This attribute is set by the framework just before submitting the job based on the test information.

New in version 3.11.0.

Changed in version 4.1: Allow `None` values

num_tasks_per_core = None

Number of tasks per core for this job.

> **Type**
>> integral or `NoneType`
>
> **Default**
>> `None`

Note: This attribute is set by the framework just before submitting the job based on the test information.

New in version 3.11.0.

num_tasks_per_node = None

Number of tasks per node for this job.

> **Type**
>> integral or `NoneType`

Default
None

Note: This attribute is set by the framework just before submitting the job based on the test information.

New in version 3.11.0.

num_tasks_per_socket = None

Number of tasks per socket for this job.

> **Type**
> integral or `NoneType`
>
> **Default**
> None

Note: This attribute is set by the framework just before submitting the job based on the test information.

New in version 3.11.0.

options = []

Arbitrary options to be passed to the backend job scheduler.

> **Type**
> `List[str]`
>
> **Default**
> `[]`

pin_nodes = []

Pin the jobs on the given nodes.

The list of nodes will be transformed to a suitable string and be passed to the scheduler's options. Currently it will have an effect only for the Slurm scheduler.

> **Type**
> `List[str]`
>
> **Default**
> `[]`

New in version 3.11.0.

property sched_access

The partition's *access* options.

property sched_flex_alloc_nodes

The argument of the `--flex-alloc-nodes` command line option.

property scheduler

The scheduler where this job is assigned to.

property script_filename

The filename of the generated job script.

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

property stderr

> The file where the standard error of the job is saved.

property stdout

> The file where the standard output of the job is saved.

property submit_time

> The submission time of this job as a floating point number expressed in seconds since the epoch, in UTC.
>
> This attribute is None if the job hasn't been submitted yet.
>
> This attribute is set right after the job is submitted and can vary significantly from the time the jobs starts running, depending on the scheduler.
>
> > **Type**
> > > float or None

time_limit = None

> Time limit for this job.
>
> See *reframe.core.pipeline.RegressionTest.time_limit* for more details.
>
> ---
>
> **Note:** This attribute is set by the framework just before submitting the job based on the test information.
>
> ---
>
> New in version 3.11.0.

use_smt = None

> Enable SMT for this job.
>
> > **Type**
> > > bool or NoneType
> >
> > **Default**
> > > None
>
> ---
>
> **Note:** This attribute is set by the framework just before submitting the job based on the test information.
>
> ---
>
> New in version 3.11.0.

property workdir

> The working directory for this job.

class reframe.core.launchers.**JobLauncher**

> Bases: 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.

Note: Changed in version 4.0.0: Users may create job launchers directly.

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: *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:

```
@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*, *default=None*)

Get a configuration option.

Parameters

- **option** – The option to be retrieved.

- **default** – The value to return if option cannot be retrieved.

Returns

The value of the option.

Changed in version 3.11.0: Add default named argument.

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_config**(*system*)

> Bases: [object]()
>
> Context manager to temporarily switch to specific configuration.

class reframe.core.runtime.**temp_environment**(*modules=None*, *env_vars=None*)

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

>   Bases: *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.

configuredir = .

The directory of the configure script.

This can be changed to do an out of source build without copying the entire source tree.

> **Type**
>> `str`
>
> **Default**
>> `'.'`

class reframe.core.buildsystems.**BuildSystem**(**args*, ***kwargs*)

Bases: `object`

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 = True**
>
> > 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.**BuildSystemMeta**(*name*, *bases*, *namespace*, ***kwargs*)

> Bases: RegressionTestMeta, ABCMeta
>
> Build systems metaclass.

class reframe.core.buildsystems.**CMake**(**args*, ***kwargs*)

>Bases: *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**(**args*, ***kwargs*)

>Bases: *BuildSystem*

>Abstract base class for configured-based build systems.

>**builddir = None**

>>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 = 1**

>>Same as for the *Make* build system.

>>**Type**
>>>integer

>>**Default**
>>>1

>**srcdir = None**

>>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.**CustomBuild**(**args*, ***kwargs*)

> Bases: *BuildSystem*

> Custom build system.

> This build system backend allows users to use custom build scripts to build the test code. It does not do any interpretation of the current test environment and it simply runs the supplied *commands*.

> Users should use this build system with caution, because environment management, reproducibility and any potential side effects are all controlled by the user's custom build system.

> New in version 3.11.0.

> **commands**

>> The commands to run for building the test code.

>> **Type**
>>> List[str]

class reframe.core.buildsystems.**EasyBuild**(**args*, ***kwargs*)

> Bases: *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 = False**

>> Instruct EasyBuild to emit a package for the built software. This will essentially pass the `--package` option to eb.

>> **Type**
>>> bool

>> **Default**
>>> False

> **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 = easybuild

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

Bases: *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 corresponding attributes and set also the *BuildSystem.flags_from_environ* flag to `False`.

makefile = None

Instruct build system to use this Makefile. This option is useful when having non-standard Makefile names.

Type

str

Default

None

max_concurrency = 1

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 = None

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**(*args, **kwargs*)

Bases: `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 = None

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 = None

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 = None

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

class reframe.core.buildsystems.**Spack**(**args*, ***kwargs*)

Bases: *BuildSystem*

A build system for building test code using Spack.

ReFrame will use a user-provided Spack environment in order to build and test a set of specs.

New in version 3.6.1.

emit_load_cmds = True

Emit the necessary `spack load` commands before running the test.

> **Type**
>> bool
>
> **Default**
>> True

environment = None

> The Spack environment to use for building this test.
>
> ReFrame will activate and install this environment. This environment will also be used to run the test.
>
> ```
> spack env activate -V -d <environment directory>
> ```
>
> ReFrame looks for environments in the test's *sourcesdir*.
>
> If this field is *None*, the default, the environment name will be automatically set to *rfm_spack_env*.
>
> > **Type**
> > > str or None
> >
> > **Default**
> > > None
>
> ---
>
> **Note:** Changed in version 3.7.3: The field is no longer required and the Spack environment will be automatically created if not provided.
>
> ---

install_opts = []

> Options to pass to spack install
>
> > **Type**
> > > List[str]
> >
> > **Default**
> > > []

install_tree = None

> The directory where Spack will install the packages requested by this test.
>
> After activating the Spack environment, ReFrame will set the *install_tree* Spack configuration in the given environment with the following command:
>
> ```
> spack config add "config:install_tree:root:<install tree>"
> ```
>
> Relative paths are resolved against the test's stage directory. If this field and the Spack environment are both *None*, the default, the install directory will be automatically set to *opt/spack*. If this field *None* but the Spack environment is not, then *install_tree* will not be set automatically and the install tree of the given environment will not be overridden.
>
> > **Type**
> > > str or None
> >
> > **Default**
> > > None
>
> New in version 3.7.3.

specs = []

> A list of additional specs to build and install within the given environment.
>
> ReFrame will add the specs to the active environment by emitting the following command:
>
> ```
> spack add spec1 spec2 ... specN
> ```
>
> If no spec is passed, ReFrame will simply install what is prescribed by the environment.

Type
    List[str]

Default
    []

Container Platforms

New in version 2.20.

class reframe.core.containers.**Apptainer**

> Bases: *Singularity*

> Container platform backend for running containers with Apptainer.

> New in version 4.0.0.

class reframe.core.containers.**ContainerPlatform**

> Bases: 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

> **classmethod create**(*name*)

> > Factory method to create a new container by name.

> **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.

        Type

            [str](#)

        Default

            `/rfm_workdir`

    Changed in version 3.12.0: This attribute is no more deprecated.

class `reframe.core.containers.`**Docker**

    Bases: [`ContainerPlatform`](#)

    Container platform backend for running containers with [Docker](#).

class `reframe.core.containers.`**Sarus**

    Bases: [`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: [`Sarus`](#)

    Container platform backend for running containers with [Shifter](#).

class `reframe.core.containers.`**Singularity**

    Bases: [`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.**simple_test**

> See *@reframe.core.decorators.simple_test*.

Mapping of Test Attributes to Job Scheduler Backends

Test at-tribute	Slurm option	Torque option	PBS option
num_tasks	--ntasks[1]	-l nodes={num_tasks// num_tasks_per_node}:ppn={num_tasks_per_node*num_cpus_per_task}	-l select={num_tasks// num_tasks_per_node}:mpiprocs={num_tasks_per_node}:ncpu
num_tasks_per_node	--ntasks-per-node	see num_tasks	see num_tasks
num_tasks_per_core	--ntasks-per-core	n/a	n/a
num_tasks_per_socket	--ntasks-per-socket	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.

Deferrable Functions Reference

Deferrable functions are the functions whose execution may be postponed to a later time after they are called. The key characteristic of these functions is that they store their arguments when they are called, and the execution itself does not occur until the function is evaluated either explicitly or implicitly.

ReFrame provides an ample set of deferrable utilities and it also allows users to write their own deferrable functions when needed. Please refer to "*Understanding the Mechanism of Deferrable Functions*" for a hands-on explanation on how deferrable functions work and how to create custom deferrable functions.

Explicit evaluation of deferrable functions

Deferrable functions may be evaluated at any time by calling `evaluate()` on their return value or by passing the deferred function itself to the `evaluate()` free function. These `evaluate()` functions take an optional `bool` argument `cache`, which can be used to cache the evaluation of the deferrable function. Hence, if caching is enabled on a given deferrable function, any subsequent calls to `evaluate()` will simply return the previously cached results.

Changed in version 3.8.0: Support of cached evaluation is added.

Implicit evaluation of deferrable functions

Deferrable 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 deferrable 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 deferrable 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 deferrable function will automatically trigger its evaluation.

Categories of deferrable functions

Currently ReFrame provides three broad categories of deferrable 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 `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. 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.

Deferrable performance functions

New in version 3.8.0.

Deferrable performance functions are a special type of deferrable functions which are intended for measuring a given quantity. Therefore, this kind of deferrable functions have an associated unit that can be used to interpret the return values from these functions. The unit of a deferrable performance function can be accessed through the public member `unit`. Regular deferrable functions can be promoted to deferrable performance functions using the `make_performance_function()` utility. Also, this utility allows to create performance functions directly from any callable.

List of deferrable functions and utilities

@reframe.utility.sanity.**deferrable**(*func*)

> Deferrable decorator.

> Converts the decorated free function into a deferrable function.

```
import reframe.utility.sanity as sn

@sn.deferrable
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*, *cache=False*)

> Evaluate a deferred expression.
>
> If `expr` is not a deferred expression, it will be returned as is. If `expr` is a deferred expression and `cache` is `True`, the results of the deferred expression will be cached and subsequent calls to *evaluate()* on this deferred expression (when `cache=False`) will simply return the previously cached result.
>
> > **Parameters**
> >
> > - **expr** – The expression to be evaluated.
> >
> > - **cache** – Cache the result of this evaluation.
>
> ---
>
> **Note:** When the `cache` argument is passed as `True`, a deferred expression will always be evaluated and its results will be re-cached. This may replace any other results that may have been cached in previous evaluations.
>
> ---
>
> New in version 2.21.
>
> Changed in version 3.8.0: The `cache` argument is added.

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.**make_performance_function**(*func*, *unit*, **args*, ***kwargs*)

Convert a callable or deferred expression into a performance function.

If `func` is a deferred expression, the performance function will be built by extending this deferred expression into a deferred performance expression. Otherwise, a new deferred performance expression will be created from the function `func()`. The argument `unit` is the unit associated with the deferrable performance expression, and `*args` and `**kwargs` are the arguments to be captured by this deferred expression. See *deferrable functions reference* for further information on deferrable functions.

New in version 3.8.0.

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:

```python
@sanity_function
def my_sanity_fn(self):
    return sn.assert_eq(
        sn.count(sn.print(sn.extractall(...))), 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: 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: MutableSet
>
> An ordered set.
>
> This container behaves like a normal set but remembers the insertion order of its elements. It can also inter-operate 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: 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'
```

(continues on next page)

```
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: `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.

Note: You can concatenate a *SequenceView* with a container of the same type as the underlying container of the view, in which case a new container with the concatenated elements will be returned.

---

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.**cache_return_value**(*fn*)

> Decorator that caches the return value of the decorated function.
>
> The function will only be called once and then the cached value will be returned each time.

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'))

    @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):
    module_info = parameter(my_find_modules('GROMACS'))
```

```
    @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.**is_trivially_callable**(*fn*, ***, *non_def_args=0*)

> Check that a callable object is trivially callable.
>
> An object is trivially callable when it can be invoked by providing just an expected number of non-default arguments to its call method. For example, (non-static) member functions expect a single argument without a default value, which will passed as `cls` or `self` during invocation depending on whether the function is a classmethod or not, respectively. On the other hand, member functions that are static methods are not passed any values by default when invoked. Therefore, these functions can only be trivially callable when their call method expects no arguments by default.
>
> > **Parameters**

- **fn** – A callable to be tested if its trivially callable.

- **non_def_args** – The number of non-default arguments the callable fn expects when invoked.

> **Returns**
>> This function returns True if the expected number of arguments matches the value of non_def_args. Otherwise, it returns False.

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.**nodelist_expand**(*nodespec*)

> Expand the nodes in nodespec to a list of nodes.

>> **Parameters**
>>> **nodespec** – A node specification as the one returned by *nodelist_abbrev()*

>> **Returns**
>>> The list of nodes corresponding to the given node specification.

> New in version 4.0.0.

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.

class reframe.utility.**temp_setattr**(*obj*, *attr*, *val*)

> Bases: `object`

> Context manager to temporarily change the attribute value of an object.

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 either the Cray Development Toolkit (CDT) version, the Cray Programming Environment (CPE) 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*, *opts=None*, *timeout=5*)

Clone a git repository from a URL.

Parameters

- **url** – The URL to clone from.

- **opts** – List of options to be passed to the *git clone* command

- **timeout** – Timeout in seconds when checking if the url is a valid repository.

- **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.**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*, ***kwargs*)

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.
>
> - **kwargs** – Keyword arguments to be passed `run_command_async()`.
>
> **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):

```
      type
        |
        |
        |
      List
     /   |
    /    |
   /     |
list  List[T]
```
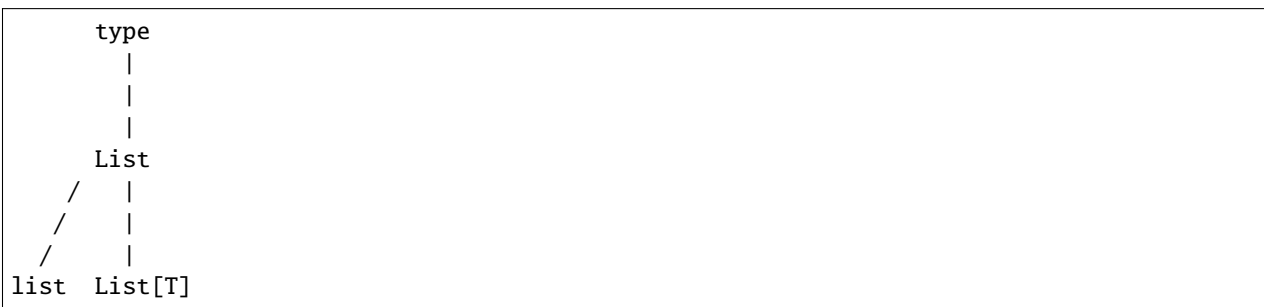
In the above example `T` may refer to any type, so that `List[List[int]]` is an instance of `List`, but not an instance of `List[int]`.

class `reframe.utility.typecheck.`**`Bool`**(*args*, ***kwargs*)

Bases: `object`

A boolean type accepting implicit conversions from strings.

This type represents a boolean value but allows implicit conversions from `str`. More specifically, the following conversions are supported:

- The strings `'yes'`, `'true'` and `'1'` are converted to `True`.

- The strings `'no'`, `'false'` and `'0'` are converted to `False`.

The built-in `bool` type is registered as a subclass of this type.

Boolean test variables that are meant to be set properly from the command line must be declared of this type and not `bool`.

class reframe.utility.typecheck.**ConvertibleType**(*name*, *bases*, *namespace*, ***kwargs*)

Bases: `ABCMeta`

A type that support conversions from other types.

This is a metaclass that allows classes that use it to support arbitrary conversions from other types using a cast-like syntax without having to change their constructor:

```
new_obj = convertible_type(another_type)
```

For example, a class whose constructor accepts and `int` may need to support a cast-from-string conversion. This is particular useful if you want a custom-typed test `variable` to be able to be set from the command line using the `-S` option.

In order to support such conversions, a class must use this metaclass and define a class method, named as `__rfm_cast_<type>__`, for each of the type conversion that needs to support .

The following is an example of a class `X` that its normal constructor accepts two arguments but it also allows conversions from string:

```
class X(metaclass=ConvertibleType):
    def __init__(self, x, y):
        self.data = (x, y)

    @classmethod
    def __rfm_cast_str__(cls, s):
        return X(*(int(x) for x in s.split(',', maxsplit=1)))

 assert X(2, 3).data == X('2,3').data
```

New in version 3.8.0.

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: *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: *ReframeError*
>
> Raised when a build fails.

exception `reframe.core.exceptions.`**`BuildSystemError`**(**args*)

> Bases: *ReframeError*
>
> Raised when a build system is not configured properly.

exception `reframe.core.exceptions.`**`CommandLineError`**(**args*)

> Bases: *ReframeError*
>
> Raised when an error in command-line arguments occurs.

exception `reframe.core.exceptions.`**`ConfigError`**(**args*)

> Bases: *ReframeError*
>
> Raised when a configuration error occurs.

exception `reframe.core.exceptions.`**`ContainerError`**(**args*)

> Bases: *ReframeError*
>
> Raised when a container platform is not configured properly.

exception `reframe.core.exceptions.`**`DependencyError`**(**args*)

> Bases: *ReframeError*
>
> Raised when a dependency problem is encountered.

exception `reframe.core.exceptions.`**`EnvironError`**(**args*)

> Bases: *ReframeError*
>
> Raised when an error related to an environment occurs.

exception `reframe.core.exceptions.`**`FailureLimitError`**(**args*)

> Bases: *ReframeError*
>
> Raised when the limit of test failures has been reached.

exception `reframe.core.exceptions.`**`ForceExitError`**(**args*)

> Bases: *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: *JobError*
>
> Raised by job schedulers when a job is blocked indefinitely.

exception `reframe.core.exceptions.`**`JobError`**(*msg=None*, *jobid=None*)

> Bases: *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: *JobError*
>
> Raised when trying an operation on a unstarted job.

exception `reframe.core.exceptions.`**`JobSchedulerError`**(**args*)

> Bases: *ReframeError*
>
> Raised when a job scheduler encounters an error condition.

exception reframe.core.exceptions.**LoggingError**(**args*)

> Bases: *ReframeError*
>
> Raised when an error related to logging has occurred.

exception reframe.core.exceptions.**NameConflictError**(**args*)

> Bases: *RegressionTestLoadError*
>
> Raised when there is a name clash in the test suite.

exception reframe.core.exceptions.**PerformanceError**(**args*)

> Bases: *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: *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()
    except Exception as e:
        print(e)
```

exception reframe.core.exceptions.**ReframeError**(**args*)

> Bases: *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: *ReframeBaseError*
>
> A fatal framework error.

Execution must be aborted.

exception reframe.core.exceptions.**RefRameSyntaxError**(**args*)

Bases: *RefRameError*

Raised when the syntax of regression tests is incorrect.

exception reframe.core.exceptions.**RegressionTestLoadError**(**args*)

Bases: *RefRameError*

Raised when the regression test cannot be loaded.

exception reframe.core.exceptions.**SanityError**(**args*)

Bases: *RefRameError*

Raised to denote an error in sanity checking.

exception reframe.core.exceptions.**SkipTestError**(**args*)

Bases: *RefRameError*

Raised when a test needs to be skipped.

exception reframe.core.exceptions.**SpawnedProcessError**(*args*, *stdout*, *stderr*, *exitcode*)

Bases: *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: *SpawnedProcessError*

Raised when a spawned OS command has timed out.

property timeout

The timeout of the process.

exception reframe.core.exceptions.**StatisticsError**(**args*)

Bases: *RefRameError*

Raised to denote an error in dealing with statistics.

exception reframe.core.exceptions.**TaskDependencyError**(**args*)

Bases: *RefRameError*

Raised inside a regression task by the runtime when one of its dependencies has failed.

exception reframe.core.exceptions.**TaskExit**(**args*)

Bases: *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.**is_user_error**(*exc_type*, *exc_value*, *tb*)

> Check if error is a user programming error.
>
> A user error is any of `AttributeError`, `NameError`, `ModuleNotFoundError`, `TypeError` or `ValueError` and the exception isthrown from user context.

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.

2.7 ReFrame Test Library (experimental)

This is a collection of generic tests that you can either run out-of-the-box by specializing them for your system using the *-S* option or create your site-specific tests by building upon them.

2.7.1 Microbenchmarks

OSU microbenchmarks

There are two final parameterized tests that represent the various OSU benchmarks:

- The `osu_run` test that runs the benchmarks only. This assumes that the OSU microbenchmarks are installed and available.
- The `osu_build_run` test that builds and runs the benchmarks. This test uses two fixtures in total: one to build the tests and one to fetch them.

Depending on your setup you can select the most appropriate final test. The benchmarks define various variables with a reasonable default value that affect the execution of the benchmark. For collective communication benchmarks, setting the `num_tasks` is required. All tests set `num_tasks_per_node` to 1 by default.

Examples

Run the run-only version of the point to point bandwidth benchmark:

```
reframe -n 'osu_run.*benchmark_info=mpi.pt2pt.osu_bw' -S modules=my-osu-benchmarks -S␣
↪valid_systems=mysystem -S valid_prog_environs=myenv -l
```

Build and run the CUDA-aware version of the allreduce benchmark.

```
reframe -n 'osu_build_run.*benchmark_info=mpi.collective.osu_allreduce.*build_type=cuda'␣
↪-S device_buffers=cuda -S num_tasks=16 -S valid_systems=sys:part -S valid_prog_
↪environs=myenv -l
```

class hpctestlib.microbenchmarks.mpi.osu.**build_osu_benchmarks**(**args*, ***kwargs*)

> Bases: *CompileOnlyRegressionTest*
>
> Fixture for building the OSU benchmarks
>
> **build_type = <reframe.core.parameters.TestParam object>**
>
> > Build variant parameter.
> >
> > > **Type**
> > > > str
> > >
> > > **Values**
> > > > 'cpu', 'cuda', 'rocm', 'openacc'
>
> **osu_benchmarks = <reframe.core.fixtures.TestFixture object>**
>
> > The fixture object that retrieves the benchmarks
> >
> > > **Type**
> > > > *fetch_osu_benchmarks*
> > >
> > > **Scope**
> > > > *session*

class hpctestlib.microbenchmarks.mpi.osu.**fetch_osu_benchmarks**(**args*, ***kwargs*)

> Bases: *RunOnlyRegressionTest*
>
> Fixture for fetching the OSU benchmarks.
>
> **version = 5.9**
>
> > The version of the benchmarks to fetch.
> >
> > > **Type**
> > > > str
> > >
> > > **Default**
> > > > '5.9'

class hpctestlib.microbenchmarks.mpi.osu.**osu_benchmark**(**args*, ***kwargs*)

> Bases: *RunOnlyRegressionTest*
>
> OSU benchmark test base class.
>
> **benchmark_info = <reframe.core.parameters.TestParam object>**
>
> > Parameter indicating the available benchmark to execute.
> >
> > > **Type**
> > > > 2-element tuple containing the benchmark name and whether latency or bandwidth is to be measured.
> > >
> > > **Values**
> > > > mpi.collective.osu_alltoall, mpi.collective.osu_allreduce, mpi.pt2pt.
> > > > osu_bw, mpi.pt2pt.osu_latency
>
> **device_buffers = cpu**
>
> > Device buffers.
> >
> > Use accelerator device buffers. Valid values are cpu, cuda, openacc or rocm.

> **Type**
>> str
>
> **Default**
>> 'cpu'

message_size

> Maximum message size.
>
> Both the performance and the sanity checks will be done for this message size.
>
> This value is set to 8 for latency benchmarks and to 4194304 for bandwidth benchmarks.
>
> > **Type**
> >> int

num_iters = 1000

> Number of iterations.
>
> This value is passed to the excutable through the -i option.
>
> > **Type**
> >> int
> >
> > **Default**
> >> 1000

num_tasks

> Number of tasks to use.
>
> This variable is required. It is set to 2 for point to point benchmarks, but it is undefined for collective benchmarks
>
> > **Required**
> >> Yes

num_warmup_iters = 10

> Number of warmup iterations.
>
> This value is passed to the excutable through the -x option.
>
> > **Type**
> >> int
> >
> > **Default**
> >> 10

class hpctestlib.microbenchmarks.mpi.osu.**osu_build_run**(*args*, ***kwargs*)

> Bases: *osu_benchmark*
>
> OSU benchmark test (build and run)
>
> **osu_binaries = <reframe.core.fixtures.TestFixture object>**
>
> > The fixture object that builds the OSU binaries
> >
> > > **Type**
> > >> *build_osu_benchmarks*
> > >
> > > **Scope**
> > >> *environment*

---

class hpctestlib.microbenchmarks.mpi.osu.**osu_run**(*args*, ***kwargs*)

 Bases: *osu_benchmark*

 Run-only OSU benchmark test

GPU benchmarks

class hpctestlib.microbenchmarks.gpu.gpu_burn.**gpu_burn_build**(*args*, ***kwargs*)

 Bases: *CompileOnlyRegressionTest*

 Fixture for building the GPU burn benchmark.

Table 2: Summary

Variables	Parameters	Fixtures
 • *gpu_arch* • *gpu_build*	*None*	*None*

 gpu_arch = None

  Set the GPU architecture.

  This variable will be passed to the compiler to generate the arch-specific code.

   Type
     str or None

   Default
     None

 gpu_build = None

  Set the build option to either `'cuda'` or `'hip'`.

   Type
     str

   Default
     'cuda'

class hpctestlib.microbenchmarks.gpu.gpu_burn.**gpu_burn_check**(*args*, ***kwargs*)

 Bases: *RunOnlyRegressionTest*

 GPU burn benchmark.

 This benchmark runs continuously GEMM, either single or double precision, on a selected set of GPUs on the node where the benchmark runs.

 The floating point precision of the computations, the duration of the benchmark as well as the list of GPU devices that the benchmark will run on can be controlled through test variables.

 This benchmark tries to build the benchmark code through the *gpu_burn_build* fixture.

 This benchmark sets the *num_gpus_per_node* test attribute, if not already set, based on the number of devices with `type == 'gpu'` defined in the corresponding partition configuration. Similarly, this benchmark will use the `arch` device configuration attribute to set the `gpu_arch` variable, if this is not already set by the user.

Table 3: Summary

Vari-ables	Pa-rame-ters	Metrics	Fixtures	System features	Environment features

•
  _use_dp_
•
  duration
•
  devices
	None	
		•
		  _gpu_perf_min_
		•
		  _gpu_temp_max_
			
			•
			  _gpu_burn_build_
			  [E]
				+gpu	+cuda OR +hip

devices = []

　　List of device IDs to run the benchmark on.

　　If empty, the benchmark will run on all the available devices.

　　　　Type
　　　　　　List[int]

　　　　Default
　　　　　　[]

duration = 10

　　Duration of the benchmark in seconds.

　　　　Type
　　　　　　int

　　　　Default
　　　　　　10

gpu_perf_min()

　　Lowest performance recorded among all the selected devices.

gpu_temp_max()

　　Maximum temperature recorded among all the selected devices.

use_dp = True

　　Use double-precision arithmetic when running the benchmark.

　　　　Type
　　　　　　bool

　　　　Default
　　　　　　True

2.7.2 Scientific Applications

class hpctestlib.sciapps.amber.nve.**amber_nve_check**(**args*, ***kwargs*)

> Bases: *RunOnlyRegressionTest*
>
> Amber NVE test.
>
> Amber is a suite of biomolecular simulation programs. It began in the late 1970's, and is maintained by an active development community.
>
> This test is parametrized over the benchmark type (see *benchmark_info*) and the variant of the code (see variant). Each test instance executes the benchmark, validates numerically its output and extracts and reports a performance metric.
>
> **assert_energy_readout**()
>
> > Assert that the obtained energy meets the required tolerance.
>
> **property bench_name**
>
> > The benchmark name.
> >
> > > **Type**
> > > > str
>
> **benchmark_info = <reframe.core.parameters.TestParam object>**
>
> > Parameter pack encoding the benchmark information.
> >
> > The first element of the tuple refers to the benchmark name, the second is the energy reference and the third is the tolerance threshold.
> >
> > > **Type**
> > > > *Tuple[str, float, float]*
> >
> > **Values**
> >
> > ```
> > [
> >     ('Cellulose_production_NVE', -443246.0, 5.0E-05),
> >     ('FactorIX_production_NVE', -234188.0, 1.0E-04),
> >     ('JAC_production_NVE_4fs', -44810.0, 1.0E-03),
> >     ('JAC_production_NVE', -58138.0, 5.0E-04)
> > ]
> > ```
>
> **property energy_ref**
>
> > The energy reference value for this benchmark.
> >
> > > **Type**
> > > > str
>
> **property energy_tol**
>
> > The energy tolerance value for this benchmark.
> >
> > > **Type**
> > > > str
>
> **input_file**
>
> > The input file to use.
> >
> > This is set to mdin.CPU or mdin.GPU depending on the test variant during initialization.
> >
> > > **Type**
> > > > str

> > **Type**
> >
> > > str
> >
> > **Default**
> >
> > > `'1.0.0'`

> **property energy_ref**
>
> > The energy reference value for this benchmark.
> >
> > > **Type**
> > >
> > > > str

> **property energy_tol**
>
> > The energy tolerance value for this benchmark.
> >
> > > **Type**
> > >
> > > > str

> **nb_impl = <reframe.core.parameters.TestParam object>**
>
> > Parameter encoding the implementation of the non-bonded calculations
> >
> > > **Type**
> > >
> > > > str
> > >
> > > **Values**
> > >
> > > > `['cpu', 'gpu']`

2.7.3 Data Analytics

class hpctestlib.data_analytics.spark.spark_checks.**compute_pi_check**(**args*, ***kwargs*)

> Bases: *RunOnlyRegressionTest*
>
> Test Apache Spark by computing PI.
>
> Apache Spark is a unified analytics engine for large-scale data processing. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for incremental computation and stream processing (see spark.apache.org).
>
> This test checks that Spark is functioning correctly. To do this, it is necessary to define the tolerance of acceptable deviation. The tolerance is used to check that the computations are executed correctly, by comparing the value of pi calculated to the one obtained from the math library. The default assumption is that Spark is already installed on the system under test.
>
> **assert_pi_readout()**
>
> > Assert that the obtained pi value meets the specified tolerances.
>
> **exec_cores = 1**
>
> > The number of cores per each Spark executor
> >
> > > **Type**
> > >
> > > > int
> > >
> > > **Required**
> > >
> > > > No
> > >
> > > **Default**
> > >
> > > > *1*

executor_memory

Amount of memory to use per executor process, following the JVM memory strings convention, i.e a number with a size unit suffix ("k", "m", "g" or "t") (e.g. 512m, 2g)

> **Type**
>> str
>
> **Required**
>> Yes

num_workers = 1

The number of Spark workers per node

> **Type**
>> int
>
> **Required**
>> No
>
> **Default**
>> *1*

spark_local_dirs = /tmp

The local directories used by Spark

> **Type**
>> str
>
> **Required**
>> No
>
> **Default**
>> *'/tmp'*

spark_prefix

The Spark installation prefix path

> **Type**
>> str
>
> **Required**
>> Yes

tolerance = 0.01

The absolute tolerance of the computed value of PI

> **Type**
>> float
>
> **Required**
>> No
>
> **Default**
>> *0.01*

variant = <reframe.core.parameters.TestParam object>

Parameter encoding the variant of the test.

> **Type**
>> str
>
> **Values**
>> ['spark', 'pyspark']

2.7.4 Python

class hpctestlib.python.numpy.numpy_ops.**numpy_ops_check**(*args*, ***kwargs*)

> Bases: *RunOnlyRegressionTest*
>
> NumPy basic operations test.
>
> NumPy is the fundamental package for scientific computing in Python. It provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.
>
> This test test performs some fundamental NumPy linear algebra operations (matrix product, SVD, Cholesky decomposition, eigendecomposition, and inverse matrix calculation) and users the execution time as a performance metric. The default assumption is that NumPy is already installed on the currest system.
>
> > **time_cholesky**()
> >
> > > Time of the `cholesky` kernel in seconds.
> >
> > **time_dot**()
> >
> > > Time of the `dot` kernel in seconds.
> >
> > **time_eigendec**()
> >
> > > Time of the `eigendec` kernel in seconds.
> >
> > **time_inv**()
> >
> > > Time of the `inv` kernel in seconds.
> >
> > **time_svd**()
> >
> > > Time of the `svd` kernel in seconds.

2.7.5 Interactive Computing

class hpctestlib.interactive.jupyter.ipcmagic.**ipcmagic_check**(*args*, ***kwargs*)

> Bases: *RunOnlyRegressionTest*
>
> Test ipcmagic via a distributed TensorFlow training with ipyparallel.
>
> ipcmagic is a Python package and collection of CLI scripts for controlling clusters for Jupyter. For more information, please have a look here.
>
> This test checks the ipcmagic performance. To do this, a single-layer neural network is trained against a noisy linear function. The parameters of the fitted linear function are returned in the end along with the resulting loss function. The default assumption is that ipcmagic is already installed on the system under test.
>
> > **assert_successful_execution**()
> >
> > > Checks that the program is running on 2 different nodes (hostnames are different), that IPCMagic is configured and returns the correct end-of-program message (returns the slope parameter in the end).

2.7.6 Machine Learning

class hpctestlib.ml.tensorflow.horovod.**tensorflow_cnn_check**(*args*, ***kwargs*)

    Bases: *RunOnlyRegressionTest*

    Run a synthetic CNN benchmark with TensorFlow2 and Horovod.

    TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications. For more information, refer to https://www.tensorflow.org/.

    Horovod is a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet. The goal of Horovod is to make distributed deep learning fast and easy to use. For more information refer to https://github.com/horovod/horovod.

    This test runs the Horovod `tensorflow2_synthentic_benchmark.py` example, checks its sanity and extracts the GPU performance.

    batch_size = 32

        The size of the batch used during the learning of models.

            Type

                int

            Default

                32

    benchmark_version = v0.21.0

        The version of Horovod to use.

            Type

                str

            Default

                'v0.21.0'

    model = InceptionV3

        The name of the model to use for this benchmark.

            Type

                str

            Default

                'InceptionV3'

    num_batches_per_iter = 5

        The number of batches per iteration.

            Type

                int

            Default

                5

    num_iters = 5

        The number of iterations.

            Type

                int

            Default

                5

num_warmup_batches = 5

> The number of warmup batches
>
> > **Type**
> > > [int](#)
> >
> > **Default**
> > > 5

throughput_iteration()

> The average GPU throughput per iteration in `images/s`.

throughput_total()

> The total GPU throughput of the benchmark in `images/s`.

class hpctestlib.ml.pytorch.horovod.**pytorch_cnn_check**(**args*, ***kwargs*)

> Bases: *RunOnlyRegressionTest*
>
> Run a synthetic CNN benchmark with PyTorch and Horovod.
>
> PyTorch is a Python package that provides tensor computation like NumPy with strong GPU acceleration and deep neural networks built on a tape-based autograd system. For more information, refer to https://pytorch.org/.
>
> Horovod is a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet. The goal of Horovod is to make distributed deep learning fast and easy to use. For more information refer to https://github.com/horovod/horovod.
>
> This test runs the Horovod `pytorch_synthentic_benchmark.py` example, checks its sanity and extracts the GPU performance.
>
> **batch_size = 64**
>
> > The size of the batch used during the learning of models.
> >
> > > **Type**
> > > > [int](#)
> > >
> > > **Default**
> > > > 64
>
> **benchmark_version = v0.21.0**
>
> > The version of Horovod to use.
> >
> > > **Type**
> > > > [str](#)
> > >
> > > **Default**
> > > > 'v0.21.0'
>
> **model = inception_v3**
>
> > The name of the model to use for this benchmark.
> >
> > > **Type**
> > > > [str](#)
> > >
> > > **Default**
> > > > 'inception_v3'
>
> **num_batches_per_iter = 5**
>
> > The number of batches per iteration.
> >
> > > **Type**
> > > > [int](#)

> > **Default**
> >
> > > 5

num_iters = 5

> The number of iterations.
>
> > **Type**
> >
> > > int
> >
> > **Default**
> >
> > > 5

num_warmup_batches = 5

> The number of warmup batches
>
> > **Type**
> >
> > > int
> >
> > **Default**
> >
> > > 5

throughput_iteration()

> The average GPU throughput per iteration in `images/s`.

throughput_total()

> The total GPU throughput of the benchmark in `images/s`.

PYTHON MODULE INDEX

h

r

Symbols

A

V

W

Z