Table of Contents

Gaudi Testing Infrastructure

Introduction

The importance of tests in software development is known to everybody... or it should, so I'll not talk about it.

To effectively run tests for a piece of software, it is important to have a testing infrastructure. Such an infrastructure should allow the developer to run the tests and check the results in the simplest possible way (if not by clicking on a button, it should be by executing a simple command). Adding new tests should be easy as well, so the introduction of tests for newly discovered bugs will not require much time, that can be devoted, instead, to fix them.

LCG provides tools for testing in its collection of libraries and tools. One of these tools is QMTest, which is easily extensible to cover the special needs that may come along in big projects like Gaudi. This is the tool I used to prepare a testing infrastructure for Gaudi.

GaudiTest.GaudiExeTest: the Gaudi extension to QMTest

Although QMTest already comes with a good set of classes to run the tests, they are not flexible enough for the special cases that may be encountered in the tests for Gaudi, e.g. the need to compare the standard output with a reference file that my be different for different platforms, or to check for the presence (or absence) of a specific message in the output.

To overcome the limitations of QMTest standard tests, I wrote a Gaudi specific test class that inherits from the standard ExecTestBase class (provided by QMTest). The new test class, as it is for its base class, is meant to run an executable and to check its return code and standard output and error streams. The special features added are:

- keep the standard output and standard error if the program is terminated by a signal (QMTest usually discards them, but we can profit from the automatic stack trace that is printed in case of a SEGFAULT)
- allow a comparison of stdout and stderr with reference files, which may be overridden on specific platforms (by a convention)
- if the command is a python script, it is executed through the python interpreter (for portability)
- allow to specify in the test description a piece of code to validate the output of a test
- define a working directory different from the one from which qmtest is run
- allow to specify options directly in the test description

The parameters that can be defined for the test are:

Program (`program`)
: This field indicates the path to the program. If it is not an absolute path, the value of the 'PATH' environment variable will be used to search for the program. If not specified, $GAUDIEXE or Gaudi.exe are used.

Argument List (`args`)
: If this field is left blank, the program is run without any arguments.
Use this field to specify the option files.
An implicit 0th argument (the path to the program) is added automatically.

Options (`options`)
: This field allows to pass a list of options to the main program without the need of a separate option file.
The content of the field is written to a temporary file which name is passed the the application as last argument (appended to the field "Argument List").

Working Directory (`workdir`)
: If this field is left blank, the program will be run from the qmtest directory, otherwise from the directory specified.

Reference Output (`reference`)
: Path to the reference file for standard output.
: If this field is left blank, any standard output will be considered valid.
: If the reference file is specified, any output on standard error is ignored.

Reference for standard error (`error_reference`)
: Path to the reference file for the standard error.
: If left blank, any output on the standard error will be considered a failure.

Unsupported Platforms (`unsupported_platforms`)
: Platform on which the test must not be run.
: List of regular expressions identifying the platforms on which the test is not run and the result is set to UNTESTED.

Validator (`validator`)
: Code to validate the output of the test.
: If defined, the function is used to validate the products of the test. If specified, overrides standard output, standard error and reference files. The code will have access to the following variables
    ◊ self: the test class instance
    ◊ stdout: the standard output of the executed test
    ◊ stderr: the standard error of the executed test
    ◊ result: the Result objects to fill with messages
    ◊ causes: a list of causes of failure

Standard Error (`stderr`)
: Text to which the standard error has to be compared (overridden by `error_reference`)

Prerequisite Tests
: List of tests that should be run before the current test and should produce the specified outcome in order to allow the execution

Standard Input
: Optional text to be passed to the executable as standard input

Time Out (`timeout`)
: Maximum time allowed to the process to complete. The default is at least 600s, but you can specify a bigger integer number, or set the environment variable `QMTEST_IGNORE_TIMEOUT` to ignore the time-out (Gaudi specific extension).

Temporary directory (`use_temp_dir`)
: Boolean value to specify if the test has to be executed in a temporary directory, false by default. The temporary directory is shared among the tests executed but a single instance of QMTest (useful in case of dependencies between output of a test and input of anohter).

Expected Exit Code (`exit_code`)
: Integer value representing exit code value expected from the program. If the program produces an exit code value different from this one, the test fails.

(Note: few parameters are missing, but they are not very relevant)

How to add tests to a package

Including a package in the testing procedure is simple. The first thing is to add the line

```
apply_pattern QMTest
```

at the end of the `requirements` file, then the directory `tests/qmtest` has to be created in the package (e.g. GaudiExamples/v19r5/tests/qmtest). With these two simple steps, the package will be included in the run of the tests described below.

Of course this is not enough to actually run any test: we need to explain to QMTest which tests to run and how. This is simple if you know how to write QMTest test description files: it is enough to add them into the directory `tests/qmtest`.

One needs to run

```
make <project>/configure
```

to make the infrastructure aware of the new test.

Best Practices

When adding tests to a package is a good practice to group them in a directory with the same name of the package, all lower case, with extension `.qms` (see the sections about grouping tests). For example, for the package GaudiKernel the tests description files should go in `tests/qmtest/gaudikernel.qms`. This will make it easier to find where a test is coming from when the results of a whole project are merged. Moreover, if such a directory exists, by default only the tests there specified will be run, giving the possibility of using another `.qms` directory for tests that should be run by hand.

When writing a custom validator, it is important to take into account that the values added to the list `causes` must be a few short hints on the type of failure, while the `results` must contain enough details to (hopefully) understand what was the actual problem. For example, in a test that checks if some Python modules can be imported, for a failure we should add something like "import error" to `causes` and the full list of modules that could not be imported to an entry in the `results`, possibly with the Python stack trace.

Test description for QMTest

In QMTest, each test is described with an XML file ending with the extension `.qmt` and with **only lowercase letters in the name**. I'm not going to explain all the possible variants of this file, I simply provide few examples to start with for Gaudi.

Run Gaudi with an option file

The test is successful if nothing is printed on the standard error and if the return code is 0.

```
<?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesour
<extension class="GaudiTest.GaudiExeTest" kind="test">
  <argument name="program"><text>gaudirun.py</text></argument>
  <argument name="args"><set><text>path/to/options/myjob.opts</text></set></argument>
</extension>
```

To use more then one option file:

```
<?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesour
<extension class="GaudiTest.GaudiExeTest" kind="test">
  <argument name="program"><text>gaudirun.py</text></argument>
  <argument name="args"><set>
    <text>path/to/options/myjob1.opts</text>
    <text>path/to/options/myjob2.opts</text>
  </set></argument>
</extension>
```

Run Gaudi with options (without an option file)

The test is successful if nothing is printed on the standard error and if the return code is 0.

```
<?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesour
```

```
<extension class="GaudiTest.GaudiExeTest" kind="test">
  <argument name="program"><text>gaudirun.py</text></argument>
  <argument name="options"><text>
from Gaudi.Configuration import *
from Configurables import MyAlg
alg = MyAlg(Cut = 10)
ApplicationMgr().TopAlg.append(alg)
</text></argument>
</extension>
```

Run Gaudi with an option file and a reference output

The test is successful if the standard output is equivalent to the reference file and if the return code is 0.

```
<?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesour
<extension class="GaudiTest.GaudiExeTest" kind="test">
  <argument name="program"><text>gaudirun.py</text></argument>
  <argument name="args"><set><text>path/to/options/myjob.opts</text></set></argument>
  <argument name="reference"><text>path/to/reference/file.ref</text></argument>
</extension>
```

If a platform specific reference file is needed it must have the name `path/to/reference/file.ref.XXX` where `XXX` has to be replaced by the first 3 letters of the value of CMTCONFIG. (This will probably change in the future)

Run a python script (GaudiPython)

Almost identical to the two previous example with only one extra line:

```
<?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesour
<extension class="GaudiTest.GaudiExeTest" kind="test">
  <argument name="program"><text>path/to/myscript.py</text></argument>
  <argument name="args"><set><text>path/to/options/myjob.opts</text></set></argument>
  <argument name="reference"><text>path/to/reference/file.ref</text></argument>
</extension>
```

Exclude a platform from a test

Starting from the basic example, let's skip the execution of the test on Windows (which is usually the one creating problems)

```
<?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesour
<extension class="GaudiTest.GaudiExeTest" kind="test">
  <argument name="program"><text>gaudirun.py</text></argument>
  <argument name="args"><set><text>path/to/options/myjob.opts</text></set></argument>
  <argument name="unsupported_platforms"><set><text>win.*</text></set></argument>
</extension>
```

Note: the regular expression `win.*` means "every string that contains 'win' followed by any character (see the Regular Expression Syntax page in the Python documentation).

Run a test and check if a string is present in the standard output

This is a bit complicated because it requires a custom validator for the test.

Starting from the simple Gaudi test with option file:

```
<?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesour
<extension class="GaudiTest.GaudiExeTest" kind="test">
  <argument name="program"><text>gaudirun.py</text></argument>
```

```
  <argument name="args"><set><text>path/to/options/myjob.opts</text></set></argument>
  <argument name="validator"><text>
expected_string = &quot;The message I expect&quot;
if stdout.find(expected_string) == -1:
    causes.append('missing string')
    result['GaudiTest.expected_string'] = result.Quote(expected_string)
</text></argument>
</extension>
```

Notes:

- since the file is XML, the special characters have to be escaped a-la XML (e.g. `&quot;` for `"`)
- the messages added to `causes` will be printed in case of failure as "Unexpected `message`.", so, in this example, "Unexpected missing string."
- the object `result` is used to pass/record useful information, in this example, if the test fails, we add the string we were expecting (the standard output is already added by default)

Run a test and check if chunk of lines is present in the standard output

A function to find and compare a block in the standard output is available to the user-defined validator.

Starting from the simple Gaudi test with option file:

```
<?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesour
<extension class="GaudiTest.GaudiExeTest" kind="test">
  <argument name="program"><text>gaudirun.py</text></argument>
  <argument name="args"><set><text>path/to/options/myjob.opts</text></set></argument>
<argument name="validator"><text>
block = """
MyAlg                INFO Event 1
MyAlg                INFO Event 2
MyAlg                INFO Event 3
MyAlg                INFO ==> Good Event
MyAlg                INFO Event 4
"""
findReferenceBlock(block, signature_offset = 3)
</text></argument>
</extension>
```

Notes:

- `signature_offset` tells the function which line to use to recognize the presence of the block in the standard output (in the example it is the 4th non-empty line of the block: the one saying "Good Event").

Use a different time-out for the test

A test should not take a long time to be able to run of all the tests in a reasonable time. Unfortunately, sometime it is not possible, so, when the 10 min. default time-out is not enough, you can change that value in the .qmt file adding a line like:

```
<argument name="timeout"><integer>1200</integer></argument>
```

specifying the desired time-out period in seconds. The .qmt file could look like this:

```
<?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesour
<extension class="GaudiTest.GaudiExeTest" kind="test">
  <argument name="program"><text>gaudirun.py</text></argument>
  <argument name="args"><set><text>path/to/options/myjob.opts</text></set></argument>
  <argument name="timeout"><integer>1200</integer></argument>
```

```
</extension>
```

Test that have to be executed in a temporary directory

When a test has to be executed in a temporary directory, it is enough to add the following line to the .qmt file.

```
<argument name="use_temp_dir"><enumeral>true</enumeral></argument>
```

Note that all the tests executed by an instance of QMTest are sharing the same temporary directory.

It is possible to force the temporary directory to be a specific directory (for debugging) setting the environment variable `QMTEST_TMPDIR`.

Check that a non-zero exit code is produced by the test

Add a line like

```
  <argument name="exit_code"><integer>1</integer></argument>
```

to the `.qmt` file. The number is the expected return code, of course.

You can find a list of valid exit codes in GaudiKernel/AppReturnCode.h.

Common examples of validators

All the following examples must be surrounded by the lines

```
<argument name="validator"><text>
```

and

```
</text></argument>
```

Check for presence of a reference block

```
block = """
MyAlg                INFO Event 1
MyAlg                INFO Event 2
MyAlg                INFO Event 3
MyAlg                INFO ==> Good Event
MyAlg                INFO Event 4
"""
findReferenceBlock(block, signature_offset = 3)
```

In case of more than one reference blocks, one call to findReferenceBlock is needed for each of them, passing always a different `id`)

Check for presence of WARNING, ERROR and FATAL messages

```
countErrorLines()
```

The function `countErrorLines` accepts a dictionary as argument, to declare how many error messages to expect. E.g.

```
countErrorLines({"FATAL":2,
                 "ERROR":7,
                 })
```

Use a different time-out for the test

How to define which tests to run automatically and/or group of tests

In QMTest, it is possible to define what is called a "suite" of tests. Of course it requires an XML file with an all lower-case name, but this time with the extension `.qms`.

XML file for a suite of tests:

```
<?xml version="1.0" ?>
<!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesourcery.com/qm/dtds/2.3/-
<extension class="explicit_suite.ExplicitSuite" kind="suite">
  <argument name="test_ids"><set>
    <text>mytest1</text>
    <text>mytest2</text>
    <text>another_test</text>
  </set></argument>
</extension>
```

XML file for a suite of suites:

```
<?xml version="1.0" ?>
<!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesourcery.com/qm/dtds/2.3/-
<extension class="explicit_suite.ExplicitSuite" kind="suite">
  <argument name="suite_ids"><set>
    <text>mysuite1</text>
    <text>mysuite2</text>
    <text>another_suite</text>
  </set></argument>
</extension>
```

XML file for a mixed suite:

```
<?xml version="1.0" ?>
<!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www.codesourcery.com/qm/dtds/2.3/-
<extension class="explicit_suite.ExplicitSuite" kind="suite">
  <argument name="suite_ids"><set>
    <text>mysuite1</text>
  </set></argument>
  <argument name="test_ids"><set>
    <text>mytest1</text>
    <text>mytest2</text>
  </set></argument>
</extension>
```

An alternative way of defining a suite of tests is to create a directory with a lower-case name ending with `.qms` and put the `.qmt` (or other `.qms` files or directories) into it. In this case, the tests will be identified with `suitename.testsname` (without extensions).

If a suite with the same name of the package (all lower-case) is found in the `tests/qmtest` directory, only the tests in it will be executed automatically.

How to run the tests

WARNING: This section is outdated as CMT is not in use anymore.

To execute the tests of a package, you have to go into the `cmt` directory of that package and execute the following command:

```
cmt TestPackage
```

This will give an output like:

```
#-------------------------------------------------------------
# Now trying [cmt qmtest_run] in .../Gaudi/vXrY/cmt (18/18)
#-------------------------------------------------------------
-tag_add=QMTest
Execute action qmtest_run => python .../GaudiPolicy/vXrY/cmt/fragments/run_qmtest.py Gaudi
==========> Running tests for package Gaudi
==========> Entering '../tests/qmtest'
==========> Running 'qmtest run -o ../../slc4_ia32_gcc34_dbg/results.qmr'
--- TEST RESULTS ------------------------------------------------------------

  import_opts                                 : PASS

  import_py                                   : PASS

  import_py_err                               : PASS

--- TESTS THAT DID NOT PASS -------------------------------------------------

  None.


--- STATISTICS --------------------------------------------------------------

       3        tests total
       3 (100%) tests PASS
```

To run a specific test (or a set of them) it is enough to pass them as arguments to the action:

```
cmt TestPackage mytest1 mytest2
```

Note that if the tests are inside a directory test suite, you should specify the name of the test as it is printed by QMTest, i.e. with all the directory names preceding the test name, without extensions and replacing `/` (slash) with `.` (dot). For example, the test `mypackage.qms/test_group1.qms/a_test.qmt` can be run with:

```
cmt TestPackage mypackage.test_group1.a_test
```

If you have the pattern `QMTestSummarize` in your requirements, you can execute all the test of the project and collect the results in one go with

```
cmt TestProject
```

Running the tests from a read/only location

It is possible to run the tests using a read/only (shared) installation of Gaudi.

Just set the environment variable `QMTESTRESULTSDIR` to point to a writable directory, before going to a package cmt directory (e.g. `GaudiRelease/cmt`) and run

```
cmt TestPackage
```

or

```
cmt TestProject
```

Conclusion

I didn't cover all possible aspects of the testing infrastructure for Gaudi, but what is here should be enough to start and go a rather long way before the need of more details. If you reach that point, drop me and e-mail

Note: Updated for Gaudi v21r11

-- MarcoClemencic - 2009-10-12

This topic: Gaudi > GaudiTestingInfrastructure
Topic revision: r19 - 2019-10-01 - unknown