GYP

Testing

Contents

Introduction

This document describes the GYP testing infrastructure, as provided by the `TestGyp.py` module.

These tests emphasize testing the *behavior* of the various GYP-generated build configurations: Visual Studio, Xcode, SCons, Make, etc. The goal is *not* to test the output of the GYP generators by, for example, comparing a GYP-generated Makefile against a set of known “golden” Makefiles (although the testing infrastructure could be used to write those kinds of tests). The idea is that the generated build configuration files could be completely written to add a feature or fix a bug so long as they continue to support the functional behaviors defined by the tests: building programs, shared libraries, etc.

“Hello, world!” GYP test configuration

Here is an actual test configuration, a simple build of a C program to print `"Hello, world!"`.

```
$ ls -l test/hello
total 20
-rw-r--r-- 1 knight knight 312 Jul 30 20:22 gyptest-all.py
-rw-r--r-- 1 knight knight 307 Jul 30 20:22 gyptest-default.py
-rwxr-xr-x 1 knight knight 326 Jul 30 20:22 gyptest-target.py
-rw-r--r-- 1 knight knight  98 Jul 30 20:22 hello.c
-rw-r--r-- 1 knight knight 142 Jul 30 20:22 hello.gyp
$
```

The `gyptest-*.py` files are three separate tests (test scripts) that use this configuration. The first one, `gyptest-all.py`, looks like this:

```
#!/usr/bin/env python

"""
Verifies simplest-possible build of a "Hello, world!" program
using an explicit build target of 'all'.
"""

import TestGyp

test = TestGyp.TestGyp()

test.run_gyp('hello.gyp')

test.build_all('hello.gyp')

test.run_built_executable('hello', stdout="Hello, world!\n")

test.pass_test()
```

The test script above runs GYP against the specified input file (`hello.gyp`) to generate a build configuration. It then tries to build the `'all'` target (or its equivalent) using the generated build configuration. Last, it verifies that the build worked as expected by running the executable program (`hello`) that was just presumably built by the generated configuration, and verifies that the output from the program matches the expected `stdout` string (`"Hello, world!\n"`).

Which configuration is generated (i.e., which build tool to test) is specified when the test is run; see the next section.

Surrounding the functional parts of the test described above are the header, which should be basically the same for each test (modulo a different description in the docstring):

```
#!/usr/bin/env python

"""
```

```
Verifies simplest-possible build of a "Hello, world!" program
using an explicit build target of 'all'.
"""

import TestGyp

test = TestGyp.TestGyp()
```

Similarly, the footer should be the same in every test:

```
test.pass_test()
```

Running tests

Test scripts are run by the `gyptest.py` script. You can specify (an) explicit test script(s) to run:

```
$ python gyptest.py test/hello/gyptest-all.py
PYTHONPATH=/home/knight/src/gyp/trunk/test/lib
TESTGYP_FORMAT=scons
/usr/bin/python test/hello/gyptest-all.py
PASSED
$
```

If you specify a directory, all test scripts (scripts prefixed with `gyptest-`) underneath the directory will be run:

```
$ python gyptest.py test/hello
PYTHONPATH=/home/knight/src/gyp/trunk/test/lib
TESTGYP_FORMAT=scons
/usr/bin/python test/hello/gyptest-all.py
PASSED
/usr/bin/python test/hello/gyptest-default.py
PASSED
/usr/bin/python test/hello/gyptest-target.py
PASSED
$
```

Or you can specify the `-a` option to run all scripts in the tree:

```
$ python gyptest.py -a
PYTHONPATH=/home/knight/src/gyp/trunk/test/lib
TESTGYP_FORMAT=scons
/usr/bin/python test/configurations/gyptest-configurations.py
PASSED
/usr/bin/python test/defines/gyptest-defines.py
PASSED
```

```
    .
    .
    .
    .
/usr/bin/python test/variables/gyptest-commands.py
PASSED
$
```

If any tests fail during the run, the `gyptest.py` script will report them in a summary at the end.

Debugging tests

Tests that create intermediate output do so under the gyp/out/testworkarea directory. On test completion, intermediate output is cleaned up. To preserve this output, set the environment variable PRESERVE=1. This can be handy to inspect intermediate data when debugging a test.

You can also set PRESERVE_PASS=1, PRESERVE_FAIL=1 or PRESERVE_NO_RESULT=1 to preserve output for tests that fall into one of those categories.

Specifying the format (build tool) to use

By default, the `gyptest.py` script will generate configurations for the “primary” supported build tool for the platform you’re on: Visual Studio on Windows, Xcode on Mac, and (currently) SCons on Linux. An alternate format (build tool) may be specified using the `-f` option:

```
$ python gyptest.py -f make test/hello/gyptest-all.py
PYTHONPATH=/home/knight/src/gyp/trunk/test/lib
TESTGYP_FORMAT=make
/usr/bin/python test/hello/gyptest-all.py
PASSED
$
```

Multiple tools may be specified in a single pass as a comma-separated list:

```
$ python gyptest.py -f make,scons test/hello/gyptest-all.py
PYTHONPATH=/home/knight/src/gyp/trunk/test/lib
TESTGYP_FORMAT=make
/usr/bin/python test/hello/gyptest-all.py
PASSED
TESTGYP_FORMAT=scons
/usr/bin/python test/hello/gyptest-all.py
PASSED
$
```

Test script functions and methods

The `TestGyp` class contains a lot of functionality intended to make it easy to write tests. This section describes the most useful pieces for GYP testing.

(The `TestGyp` class is actually a subclass of more generic `TestCommon` and `TestCmd` base classes that contain even more functionality than is described here.)

Initialization

The standard initialization formula is:

```
import TestGyp
test = TestGyp.TestGyp()
```

This copies the contents of the directory tree in which the test script lives to a temporary directory for execution, and arranges for the temporary directory's removal on exit.

By default, any comparisons of output or file contents must be exact matches for the test to pass. If you need to use regular expressions for matches, a useful alternative initialization is:

```
import TestGyp
test = TestGyp.TestGyp(match = TestGyp.match_re,
                       diff = TestGyp.diff_re)`
```

Running GYP

The canonical invocation is to simply specify the `.gyp` file to be executed:

```
test.run_gyp('file.gyp')
```

Additional GYP arguments may be specified:

```
test.run_gyp('file.gyp', arguments=['arg1', 'arg2', ...])
```

To execute GYP from a subdirectory (where, presumably, the specified file lives):

```
test.run_gyp('file.gyp', chdir='subdir')
```

Running the build tool

Running the build tool requires passing in a `.gyp` file, which may be used to calculate the name of a specific build configuration file (such as a MSVS solution file corresponding to the `.gyp` file).

There are several different `.build_*()` methods for invoking different types of builds.

To invoke a build tool with an explicit `all` target (or equivalent):

```
test.build_all('file.gyp')
```

To invoke a build tool with its default behavior (for example, executing `make` with no targets specified):

```
test.build_default('file.gyp')
```

To invoke a build tool with an explicit specified target:

```
test.build_target('file.gyp', 'target')
```

Running executables

The most useful method executes a program built by the GYP-generated configuration:

```
test.run_built_executable('program')
```

The `.run_built_executable()` method will account for the actual built target output location for the build tool being tested, as well as tack on any necessary executable file suffix for the platform (for example `.exe` on Windows).

`stdout=` and `stderr=` keyword arguments specify expected standard output and error output, respectively. Failure to match these (if specified) will cause the test to fail. An explicit `None` value will suppress that verification:

```
test.run_built_executable('program',
                          stdout="expect this output\n",
                                                    stderr=None)
```

Note that the default values are `stdout=None` and `stderr=''` (that is, no check for standard output, and error output must be empty).

Arbitrary executables (not necessarily those built by GYP) can be executed with the lower-level `.run()` method:

```
test.run('program')
```

The program must be in the local directory (that is, the temporary directory for test execution) or be an absolute path name.

Fetching command output

```
test.stdout()
```

Returns the standard output from the most recent executed command (including `.run_gyp()`, `.build_*()`, or `.run*()` methods).

```
test.stderr()
```

Returns the error output from the most recent executed command (including `.run_gyp()`, `.build_*()`, or `.run*()` methods).

Verifying existence or non-existence of files or directories

```
test.must_exist('file_or_dir')
```

Verifies that the specified file or directory exists, and fails the test if it doesn't.

```
test.must_not_exist('file_or_dir')
```

Verifies that the specified file or directory does not exist, and fails the test if it does.

Verifying file contents

```
test.must_match('file', 'expected content\n')
```

Verifies that the content of the specified file match the expected string, and fails the test if it does not. By default, the match must be exact, but line-by-line regular expressions may be used if the `TestGyp` object was initialized with `TestGyp.match_re`.

```
test.must_not_match('file', 'expected content\n')
```

Verifies that the content of the specified file does *not* match the expected string, and fails the test if it does. By default, the match must be exact, but line-by-line regular expressions may be used if the `TestGyp` object was initialized with `TestGyp.match_re`.

```
test.must_contain('file', 'substring')
```

Verifies that the specified file contains the specified substring, and fails the test if it does not.

```
test.must_not_contain('file', 'substring')
```

Verifies that the specified file does not contain the specified substring, and fails the test if it does.

```
test.must_contain_all_lines(output, lines)
```

Verifies that the output string contains all of the "lines" in the specified list of lines. In practice, the lines can be any substring and need not be `\n` -terminaed lines per se. If any line is missing, the test fails.

```
test.must_not_contain_any_lines(output, lines)
```

Verifies that the output string does *not* contain any of the “lines” in the specified list of lines. In practice, the lines can be any substring and need not be `\n` -terminaed lines per se. If any line exists in the output string, the test fails.

```
test.must_contain_any_line(output, lines)
```

Verifies that the output string contains at least one of the “lines” in the specified list of lines. In practice, the lines can be any substring and need not be `\n` -terminaed lines per se. If none of the specified lines is present, the test fails.

Reading file contents

```
test.read('file')
```

Returns the contents of the specified file. Directory elements contained in a list will be joined:

```
test.read(['subdir', 'file'])
```

Test success or failure

```
test.fail_test()
```

Fails the test, reporting `FAILED` on standard output and exiting with an exit status of `1` .

```
test.pass_test()
```

Passes the test, reporting `PASSED` on standard output and exiting with an exit status of `0` .

```
test.no_result()
```

Indicates the test had no valid result (i.e., the conditions could not be tested because of an external factor like a full file system). Reports `NO RESULT` on standard output and exits with a status of `2` .

Powered by Gitiles